O'REILLY

Fifth Edition

CSS
The Definitive Guide

Visual Styling for the Web

Eric A. Meyer
& Estelle Weyl

CSS: The Definitive Guide

FIFTH EDITION

Visual Styling for the Web

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Eric A. Meyer and Estelle Weyl

CSS: The Definitive Guide

by Eric A. Meyer and Estelle Weyl

Copyright © 2023 Eric Meyer. 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://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Amanda Quinn

Development Editor: Rita Fernando

Production Editor: Elizabeth Faerm

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

May 2000: First Edition

March 2004: Second Edition

November 2006: Third Edition

November 2017: Fourth Edition

Revision History for the Fifth Edition

- 2022-07-25: First Release
- 2022-08-25: Second Release
- 2022-11-22: Third Release

See http://oreilly.com/catalog/errata.csp?isbn=9781449393199 for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *CSS: The Definitive Guide*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source

licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-11755-9

[TO COME]

Chapter 1. CSS Fundamentals

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at rfernando@oreilly.com.

Cascading Style Sheets (CSS) is a powerful programming language 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 many ostensibly non-web environments. For example, embedded-device displays often use CSS to style their user interfaces, many RSS clients let you apply CSS to feeds and feed entries, and some instant message clients use CSS to format chat windows. Aspects of CSS can be found in the syntax used by JavaScript frameworks, and even in JavaScript itself. 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. At the time, browsers gave all sorts of styling power to the user—the presentation preferences in Mosaic, for example, permitted 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 dozen other element types. If a user configured their browser to make all level-one headings tiny and pink and all level-six headings huge and red, well, that was their lookout.

It was into this milieu that CSS was introduced. Its goal was to 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 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 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 catch on.

Before all that happened, though, the CSS Working Group had finalized the 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 hard to speak of a single "CSS3 specification." There isn't any such thing, nor can there be. Even if every other CSS module had reached level 3 by, say, late 2016 (they didn't), there was already a Selectors Level 4 in process. Would we then speak of it as CSS4? What about all the "CSS3" features still coming into play? Or Grid Layout, which had not then even reached Level 1? That's why this book is a definitive guide for "CSS" as a whole — because there really is no such thing as CSS3.

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 permitted by modules more than makes up for the semantic awkwardness they sometimes create. (If you want something approximating a single monolithic specification, the CSS Working Group publishes yearly "Snapshot" documents.)

With that established, we're ready to start understanding CSS. Let's start by covering the basics of what goes inside a stylesheet.

Stylesheet Contents

Inside a stylesheet, you'll find a number of *rules* which are comprised of *selectors* and *declaration blocks*, the latter of which are made up of one or more *declarations* that are themselves made up of *property* and *value* combinations. All put together, they look a little something like this:

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

Styles such as these comprise the bulk of any stylesheet—simple or complex, short or long. But which parts are which, and what do they represent?

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 stylesheet is made up of a series of rules. Figure 1-1 shows the parts of a rule.

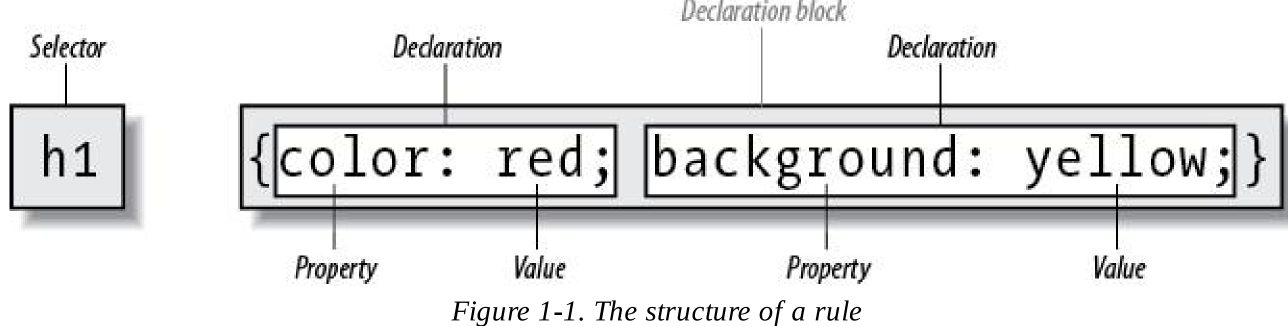


Figure 1-1. The structure of a rule

The selector, shown on the left side of the rule, defines which piece of the document will be selected for styling. In Figure 1-1, `h1` (heading level 1) 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 1-1, 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 early 2022, there were a few vendor prefixes in the wild, with the most common being shown in Table 1-1.

Table 1-1. 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 1-1 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 early 2022, vendor-prefixed CSS features are nearly non-existent, with old prefixed properties and values being slowly but steadily removed from browser implementations. It's quite likely that you'll never write prefixed CSS, but you may encounter it in the wild, or inherit it in a legacy codebase. Here's an example:

```
-webkit-transform-origin: 0 0;
-moz-transform-origin: 0 0;
-o-transform-origin: 0 0;
transform-origin: 0 0;
```

That's saying the same thing four times: once each for the WebKit, Mozilla (Firefox), and Opera browser lines, and then finally the CSS-standard way. Again, this is no longer really necessary. We're only including it here to give you an idea of what it might look like, should you come across this in the future.

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 examples out 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, usually by an automated server-side script of some sort. 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.

All 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.

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 */
```

WARNING

One way to create “nested” comments accidentally is to temporarily comment out a large block of a stylesheet 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 stylesheet 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.

NOTE

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!

Markup

There is no markup in stylesheets. 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><!--
h1 {color: maroon;}
body {background: yellow;}
--></style>
```

That’s it, and even that isn’t recommended any more — the browsers that needed it have faded into near-oblivion.

Speaking of markup, it’s time to take a very slight detour to talk about the elements that our CSS will be used to style, and how those can be affected by CSS in the most fundamental ways.

Elements

Elements are the basis of document structure. In HTML, the most common elements are easily recognizable, such as `p`, `table`, `span`, `a`, and `article`. 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. 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:

```
<img src="howdy.gif" alt="Hello, friend!">
```

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 file whose location is given by the `src` attribute). If you point to a valid image file, the image will be placed in the document. If not, the browser will either display nothing or will show a "broken image" placeholder.

Similarly, the `input` element can also be replaced—by a radio button, checkbox, text input box, or other, 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, `<span>hi there</span>` 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

CSS has two basic display roles: *block formatting context* and *inline formatting context*. There are many more display types, but these are the most basic, and the types to which most if not all other display types refer. The block and inline contexts will be familiar to authors who have spent time with HTML markup and its display in web browsers. The elements are illustrated in Figure 1-2.

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 1-2. Block- and inline-level elements in an HTML document

Block-level elements

By default, *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.

In CSS, this is referred to as an element generating a *block formatting context*. It also means that the element generates a block *outer display type*. The parts inside the element may have different display types.

Inline-level elements

By default, *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.

In CSS, this is referred to as an element generating an *inline formatting context*. It also means that the element generated an *inline outer display* type. The parts inside the element may have different display types. (In CSS, there is no restriction on how display roles can be nested within each other.)

To see how this works, let's consider the CSS property `display`.

DISPLAY

Values	[*<display-outside>* ‖ *<display-inside>*] | *<display-listitem>* | *<display-internal>* | *<display-box>* | *<display-legacy>*
Definitions	See below
Initial value	`inline`
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	No

<display-outside>

`block` | `inline` | `run-in`

<display-inside>

`flow` | `flow-root` | `table` | `flex` | `grid` | `ruby`

<display-listitem>

`list-item` && *<display-outside>*? && [`flow` | `flow-root`]?

<display-internal>

`table-row-group` | `table-header-group` | `table-footer-group` | `table-row` | `table-cell` | `table-column-group` | `table-column` | `table-caption` | `ruby-base` | `ruby-text` | `ruby-base-container` | `ruby-text-container`

<display-box>

`contents` | `none`

<display-legacy>

`inline-block` | `inline-list-item` | `inline-table` | `inline-flex` | `inline-grid`

You may have noticed that there are a *lot* of values here, only two of which we’ve mentioned: `block` and `inline`. Most of these values will be dealt with elsewhere in the book; for example, `grid` and `inline-grid` will be covered in a separate chapter about grid layout, and the table-related values are all covered in a chapter on CSS table layout.

For now, 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 elements (`body` and `p`) that are generating block formatting contexts, and one element (`em`) with an inline formatting context. 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 part of CSS.

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>The Victorian Internet</maintitle>
 <subtitle>The Remarkable Story of the Telegraph and the Nineteenth Century's On-Line
Pioneers</subtitle>
 <author>Tom Standage</author>
 <publisher>Bloomsbury Pub Plc USA</publisher>
 <pubdate>February 25, 2014</pubdate>
 <isbn type="isbn-13">9781620405925</isbn>
 <isbn type="isbn-10">162040592X</isbn>
</book>
```

Since the default value of `display` is `inline`, the content would be rendered as inline text by default, as illustrated in Figure 1-3. This isn’t a terribly useful display.

The Victorian Internet The Remarkable Story of the Telegraph and the Nineteenth Century's On-Line Pioneers Tom Standage Bloomsbury Pub Plc USA February 25, 2014 9781620405925 162040592X

Figure 1-3. Default display of an XML document

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

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

We’ve now set five of the seven elements to be block and two to be inline. This means each of the block elements will generate its own block formatting context, and the two inlines will generate their own inline formatting contexts.

We 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 1-4.

The Victorian Internet

The Remarkable Story of the Telegraph and the Nineteenth Century's On-Line Pioneers

Tom Standage

Bloomsbury Pub Plc USA (February 25, 2014)

ISBN-13 9781620405925

ISBN-10 162040592X

Figure 1-4. Styled display of an XML document

That said, 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

We’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:

```
<!DOCTYPE html>
<html>
<head>
  <title>Eric's World of Waffles</title>
  <meta charset="utf-8">
  <link rel="stylesheet" media="screen, print" href="sheet1.css">
  <style>
    /* 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 1-5.

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!

Figure 1-5. 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" href="sheet1.css" media="screen, print">
```

The `link` tag’s basic purpose is to allow HTML authors to associate other documents with the document containing the `link` tag. CSS uses it to link stylesheets to the document; in Figure 1-6, a stylesheet called *sheet1.css* is linked to the document.

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

To successfully load an external stylesheet, `link` should be placed inside the `head` element, though it can also appear inside the `body` element. This will cause the web browser to locate and load the stylesheet and use whatever styles it contains to render the HTML document in the manner shown in Figure 1-6.

Also shown in Figure 1-6 is the loading of the external *sheet2.css* via an `@import` declaration. Imports

must be placed at the beginning of the stylesheet that contains them.

```
<!DOCTYPE html>
<html>
<head>
<title>Eric's World of Waffles</title>
<meta charset="utf-8">
<link rel="stylesheet" media="screen,print"
     href="sheet1.css">
<style type="text/css">
/* These are my styles! Yay! */
@import url(sheet2.css);
h1 {color: silver;}
</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>
```

index.html

```
body {background: white; font: medium serif;}
h1 {color: blue;}
a:link {color: navy; text-decoration: underline;}
p {margin-left: 5%; margin-right: 10%;}
p:first-line {font-size: 120%; font-weight: bold;
   color: black;}
p.footnote {font-size: smaller;}
blockquote {font-style: italic;}
blockquote em {font-style: normal;}
pre, code, tt {color: gray; font-family: monospace;}
```

sheet1.css

```
h1 {
        font-size: 1.8em;
        border-bottom: 1px dotted silver;
}
h1:first-letter {
        font-size: 125%;
}
```

sheet2.css

Figure 1-6. A representation of how external stylesheets are applied to documents

And what is the format of an external stylesheet? It's 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 stylesheet—only style rules. Here are the contents of an external stylesheet:

```
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*.

> **WARNING**
>
> An external stylesheet cannot contain any document markup at all, only CSS rules and CSS comments. The presence of markup in an external stylesheet can cause some or all of it to be ignored.

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`. Note that the `rel` attribute is **required**. There is an optional `type` attribute whose default value is `text/css`, so you can include `type="text/css"` or leave it out, whichever you prefer.

These attribute values describe the relationship and type of data that will be loaded using the `link` tag. That way, the web browser knows that the stylesheet is a CSS stylesheet, a fact that will determine how the browser will deal with the data it imports. (There may be other style languages used in the future. In such a future, if you are using a different style language, the `type` attribute will need to be declared.)

Next, we find the `href` attribute. The value of this attribute is the URL of your stylesheet. This URL can be either absolute or relative; that is, either relative to the URL of the document containing the URL, or else a complete URL that points to a unique location on the web. In our example, the URL is relative. It just as easily could have been something absolute, 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 stylesheet in both screen and print media:

```
<link rel="stylesheet" href="visual-sheet.css" media="screen, print">
```

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. The default value is `all`, which means the CSS will be applied in all media.

Note that there can be more than one linked stylesheet 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 stylesheets named *basic.css* and *splash.css*, it would look like this:

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

This will cause the browser to load both stylesheets, combine the rules from each, and apply them all to the document in all media types (because the `media` attribute was omitted, its default value `all` is used). For example:

```
<link rel="stylesheet" href="basic.css">
<link rel="stylesheet" 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 isn't in this 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'll explore that in the next section.

Alternate stylesheets

It's also possible to define *alternate stylesheets* that users can select in some browsers. 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 stylesheets, 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" href="sheet1.css" title="Default">
<link rel="alternate stylesheet" href="bigtext.css" title="Big Text">
<link rel="alternate stylesheet" 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 1-7 shows one way in which this selection mechanism might be accomplished (and in fact was, early in the resurgence of CSS).

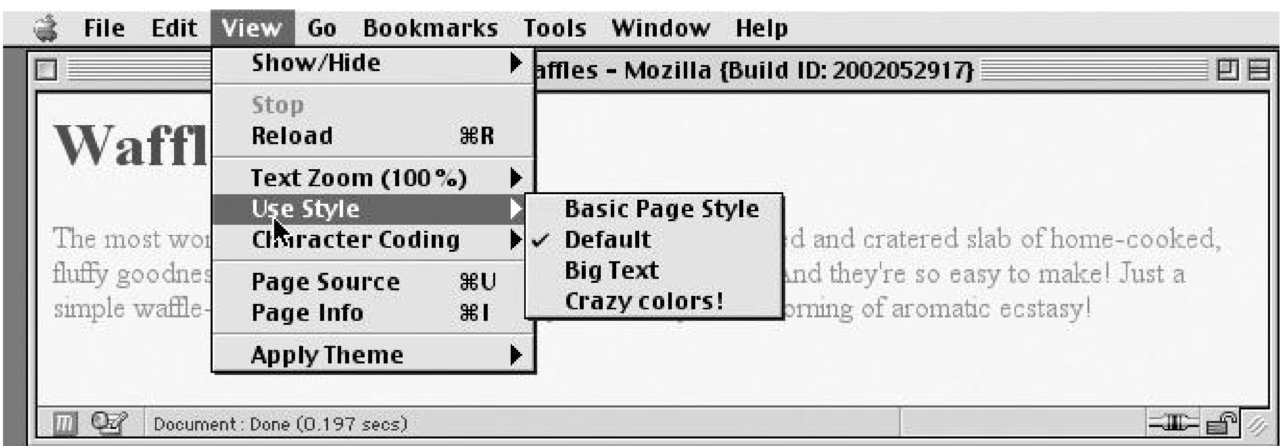


Figure 1-7. A browser offering alternate stylesheet selection

> **NOTE**
>
> As of early 2022, alternate stylesheets were supported in most Gecko-based browsers like Firefox, and in Opera. The Chromium and WebKit families did not support selecting alternate stylesheets. Compare this to the build date of the browser shown in Figure 1-7, which is late 2002.

It's also possible to group alternate stylesheets 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:

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

If a user selects "Big Text" from the alternate stylesheet 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 stylesheet as a *preferred stylesheet*. This means that its use is preferred to alternate stylesheets, and it will be used when the document is first displayed. Once you select an alternate stylesheet, however, the preferred stylesheet will *not* be used.

Furthermore, if you designate a number of stylesheets as preferred, then all but one of them will be ignored. Consider the following code example:

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

All three `link` elements now refer to preferred stylesheets, 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 stylesheets should be ignored and which should be used.

If you don't give a stylesheet a title, then it becomes a *persistent stylesheet* and is always used in the display of the document. Often, this is exactly what an author wants, especially since alternate stylesheets are not widely supported, and almost completely unknown to users.

The style Element

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

```
<style>...</style>
```

The styles between the opening and closing `style` tags are referred to as the *document stylesheet* or the *embedded stylesheet* (because this kind of stylesheet is embedded within the document). It contains styles that apply to the document, but it can also contain multiple links to external stylesheets using the `@import` directive, discussed in the next section.

You can give `style` elements a `media` attribute, which functions in the same manner as it does on linked stylesheets. This, for example, will restrict an embedded stylesheet's rules to be applied in print media only.

```
<style media="print">…</style>
```

You can also label an embedded stylesheet with a `title` element, in the same manner and for the same reasons discussed in the previous section on alternate stylesheets.

As with the `link` element, the `style` element can use the attribute `type`; in the case of a CSS document, the correct value is `"text/css"`. The `type` attribute is optional in HTML5 as long as you're loading CSS, because the default value for the `type` attribute on the `style` element is `text/css`. It would only be necessary to explicitly declare a `type` value if you were using some other styling language, perhaps in a future where such a thing in supported. For the time being, though, it remains wholly optional.

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 stylesheet and use its styles in the rendering of the HTML document. The only major difference is in the syntax and placement of the command. As you can see, `@import` is found inside the `style` element. It must be

placed first, before the other CSS rules, or it won't work at all. Consider this example:

```
<style>
@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 stylesheets of every `@import` directive will be loaded and used; there is no way to designate alternate stylesheets with `@import`. So, given the following markup:

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

…all three external stylesheets 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 stylesheets to one or more media by providing media descriptors after the stylesheet's URL:

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

As noted in "The link Tag", media descriptors can get quite complicated, and are explained in detail in XREF HERE.

`@import` can be highly useful if you have an external stylesheet that needs to use the styles found in other external stylesheets. Since external stylesheets cannot contain any document markup, the `link` element can't be used—but `@import` can. Therefore, you might have an external stylesheet 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 hopefully 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 stylesheet, as they did in the example document. CSS requires the `@import` directive to come before any other rules in a stylesheet. An `@import` that comes after other rules (e.g., `body {color: red;}`) will be ignored by conforming user agents.

WARNING

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.

There is another descriptor that can be added to an `@import` directive, which is a *cascade layer* identifier. This assigns all of the styles in the imported stylesheet to a cascade layer, which is a concept we'll explore in Chapter 4. It looks like this:

```
@import url(basic-text.css) screen layer(basic);
```

That assigns the styles from `basic-text.css` to the `basic` cascade layer. If you want to just assign the styles to an un-named layer, use `layer` without the parenthetical naming, like so:

```
@import url(basic-text.css) screen layer;
```

Note that this ability is a difference between `@import` and `link`, as the latter cannot be labeled with a cascade layer.

HTTP Linking

There is another, far more obscure way to associate CSS with a document: you can link the two 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 stylesheet with any documents served from under that *.htaccess* file. The browser will then treat it as if it were a linked stylesheet. 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.

You probably noticed the use of the term "supporting browsers." As of early 2022, the widely used browsers that support HTTP linking of stylesheets are the Firefox family and Opera. That restricts this technique mostly to development environments based on one of those browsers. In such a 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 Chromium, WebKit, and Internet Explorer families, assuming you have a reason to do so.

> **NOTE**
>
> 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 `link` elements 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 just assign a few styles to one individual element, without the need for embedded or external stylesheets, it’s possible to employ the HTML attribute `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, even tags 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 stylesheet, 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 discouraged. 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 ancient `font` tag, even if they do have a good deal more flexibility in terms of what visual effects they can apply.

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 stylesheets with a document. The user will never know whether this is done via an external or embedded stylesheet, or even with an inline style. The real importance of external stylesheets 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 the next few chapters.

Chapter 2. Selectors

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at rfernando@oreilly.com.

One of the primary advantages of CSS is its ability to easily apply a set of styles to all elements of the same type. Unimpressed? Consider this: by editing a single line of CSS, you can change the colors of all your headings. Don't like the blue you're using? Change that one line of code, and they can all be purple, yellow, maroon, or any other color you desire. That lets you, the author, focus on design and user experience rather than tedious find-and-replace operations. The next time you're in a meeting and someone wants to see headings with a different shade of green, just edit your style and hit Reload. *Voilà!* The results are accomplished in seconds and there for everyone to see.

Basic Style Rules

As stated, a central feature of CSS is its ability to apply certain rules to an entire set of element types in a document. For example, let's say that you want to make the text of all `h2` elements appear gray. Before we had CSS, you'd have to do this by inserting `<font color="gray">...</font>` tags inside all your `h2` elements. Applying inline styles using the `style` attribute, which is also bad practice, would require you to include `style="color: gray;"` in all your `h2` elements, like this:

```
<h2 style="color: gray;">This is h2 text</h2>
```

This will be a tedious process if your document contains a lot of `h2` elements. Worse, if you later decide that you want all those `h2`s to be green instead of gray, you'd have to start the manual tagging all over again. (Yes, this is really how it used to be done!)

CSS allows you to create rules that are simple to change, edit, and apply to all the text elements you define (the next section will explain how these rules work). For example, you can write this rule once to make all your `h2` elements gray:

```
h2 {color: gray;}
```

Type Selectors

A *type selector*, previously known as an *element selector*, is most often an HTML element, but not always. For example, if a CSS file contains styles for an XML document, the type selectors might look something like this:

```
quote {color: gray;}
bib {color: red;}
booktitle {color: purple;}
myElement {color: red;}
```

In other words, the elements of the document are the node types being selected. In XML, a selector could be anything because XML allows for the creation of new markup languages that can have just about anything as an element name. If you're styling an HTML document, on the other hand, the selector will generally be one of the many HTML elements such as `p`, `h3`, `em`, `a`, or even `html` itself. For example:

```
html {color: black;}
h1 {color: gray;}
h2 {color: silver;}
```

The results of this stylesheet are shown in Figure 2-1.

Plutonium

Useful for many applications, plutonium can also be dangerous if improperly handled.

Safety Information

When handling plutonium, care must be taken to avoid the formation of a critical mass.

With plutonium, the possibility of implosion is very real, and must be avoided at all costs. This can be accomplished by keeping the various masses separate.

Comments

It's best to avoid using plutonium **at all** if it can be avoided.

Figure 2-1. Simple styling of a simple document

Once you've globally applied styles directly to elements, you can shift those styles from one element to another. Let's say you decide that the paragraph text, not the `h1` elements, in Figure 2-1 should be gray. No problem. Just change the `h1` selector to `p`:

```
html {color: black;}
p {color: gray;}
h2 {color: silver;}
```

The results are shown in Figure 2-2.

Plutonium

Useful for many applications, plutonium can also be dangerous if improperly handled.

Safety Information

When handling plutonium, care must be taken to avoid the formation of a critical mass.

With plutonium, the possibility of implosion is very real, and must be avoided at all costs. This can be accomplished by keeping the various masses separate.

Comments

It’s best to avoid using plutonium **at all** if it can be avoided.

Figure 2-2. Moving a style from one element to another

Grouping

So far, we’ve seen fairly simple techniques for applying a single style to a single selector. But what if you want the same style to apply to multiple elements? *Grouping* allows an author to drastically compact certain types of style assignments, which makes for a shorter stylesheet.

Grouping Selectors

Let’s say you want both `h2` elements and paragraphs to have gray text. The easiest way to accomplish this is to use the following declaration:

```
h2, p {color: gray;}
```

By placing the `h2` and `p` selectors at the beginning of the rule, before the opening curly brace, and separating them with a comma, you’ve defined a rule where the style inside the curly braces (`color: gray;`) applies to the elements referenced by both selectors. The comma tells the browser that there are two different selectors involved in the rule. Leaving out the comma would give the rule a completely different meaning, which we’ll explore in “Descendant Selectors”.

The following alternatives produce exactly the same result, but one is a lot easier to type:

```
h1 {color: purple;}
h2 {color: purple;}
h3 {color: purple;}
h4 {color: purple;}
h5 {color: purple;}
h6 {color: purple;}

h1, h2, h3, h4, h5, h6 {color: purple;}
```

The second alternative, with one with the grouped selector, is also a lot easier to maintain over time.

The universal selector

The *universal selector*, displayed as an asterisk (`*`), matches any element at all, much like a wildcard. For example, to make every single element in a document bold, you would write:

```
* {font-weight: bold;}
```

This declaration is equivalent to a grouped selector that lists every element contained within the document. The universal selector lets you assign the `font-weight` value `bold` to every element in the document in one efficient stroke. Beware, however: although the universal selector is convenient because it targets everything within its declaration scope, it can have unintended consequences, which are discussed in Chapter 4.

Grouping Declarations

Just as you can group selectors together into a single rule, you can also group declarations. Assuming that you want all `h1` elements to appear in purple, 18-pixel-high Helvetica text on an aqua background (and you don't mind blinding your readers), you could write your styles like this:

```
h1 {font: 18px Helvetica;}
h1 {color: purple;}
h1 {background: aqua;}
```

But this method is inefficient—imagine creating such a list for an element that will carry 10 or 15 styles! Instead, you can group your declarations together:

```
h1 {font: 18px Helvetica; color: purple; background: aqua;}
```

This will have exactly the same effect as the three-line stylesheet just shown.

Note that using semicolons at the end of each declaration is crucial when you're grouping them. Browsers ignore whitespace in stylesheets, so the user agent must rely on correct syntax to parse the stylesheet. You can fearlessly format styles like the following:

```
h1 {
  font: 18px Helvetica;
  color: purple;
  background: aqua;
}
```

You can also minimize your CSS, removing all non-required spaces.

```
h1{font:18px Helvetica;color:purple;background:aqua;}
```

The last three examples are treated equally by the server, but the second one is generally regarded as the most human-readable, and the recommended method of writing your CSS during development. You might choose to minimize your CSS for network-performance reasons, but this is usually automatically handled by a build tool, server-side script, caching network, or other service, so you're usually better off writing your CSS in a human-readable fashion.

If the semicolon is omitted on the second statement, the user agent will interpret the stylesheet as follows:

```
h1 {
  font: 18px Helvetica;
  color: purple background: aqua;
}
```

Because `background:` is not a valid value for `color`, a user agent will ignore the `color` declaration (including the `background: aqua` part) entirely. You might think the browser would at least render `h1`s as purple text without an aqua background, but not so. Instead, they will be the default color (which is usually black) with a transparent background (which is also a default). The declaration `font: 18px Helvetica` will still take effect since it was correctly terminated with a semicolon.

> **TIP**
>
> Although it is not technically necessary to follow the last declaration of a rule with a semicolon in CSS, it is generally good practice to do so. First, it will keep you in the habit of terminating your declarations with semicolons, the lack of which is one of the most common causes of rendering errors. Second, if you decide to add another declaration to a rule, you won’t have to worry about forgetting to insert an extra semicolon.

As with selector grouping, declaration grouping is a convenient way to keep your stylesheets short, expressive, and easy to maintain.

Grouping Everything

You now know that you can group selectors and you can group declarations. By combining both kinds of grouping in single rules, you can define very complex styles using only a few statements. Now, what if you want to assign some complex styles to all the headings in a document, and you want the same styles to be applied to all of them? Here’s how to do it:

```
h1, h2, h3, h4, h5, h6 {color: gray; background: white; padding: 0.5em;
  border: 1px solid black; font-family: Charcoal, sans-serif;}
```

Here we’ve grouped the selectors, so the styles inside the curly braces will be applied to all the headings listed; grouping the declarations means that all of the listed styles will be applied to the selectors on the left side of the rule. The result of this rule is shown in Figure 2-3.

Plutonium

Useful for many applications, plutonium can also be dangerous if improperly handled.

Safety Information

When handling plutonium, care must be taken to avoid the formation of a critical mass.

With plutonium, the possibility of implosion is very real, and must be avoided at all costs. This can be accomplished by keeping the various masses separate.

Comments

It's best to avoid using plutonium **at all** if it can be avoided.

Figure 2-3. Grouping both selectors and rules

This approach is preferable to the drawn-out alternative, which would begin with something like this:

```
h1 {color: gray;}
h2 {color: gray;}
h3 {color: gray;}
h4 {color: gray;}
h5 {color: gray;}
h6 {color: gray;}
h1 {background: white;}
h2 {background: white;}
h3 {background: white;}
```

…and continue for many lines. You *can* write out your styles the long way, but we don't recommend it—editing them would be about as tedious as using `style` attributes everywhere!

Grouping allows for some interesting choices. For example, all of the groups of rules in the following example are equivalent—each merely shows a different way of grouping both selectors and declarations:

```
/* group 1 */
h1 {color: silver; background: white;}
h2 {color: silver; background: gray;}
h3 {color: white; background: gray;}
h4 {color: silver; background: white;}
b {color: gray; background: white;}


/* group 2 */
h1, h2, h4 {color: silver;}
h2, h3 {background: gray;}
h1, h4, b {background: white;}
h3 {color: white;}
b {color: gray;}

/* group 3 */
```

```
h1, h4 {color: silver; background: white;}
h2 {color: silver;}
h3 {color: white;}
h2, h3 {background: gray;}
b {color: gray; background: white;}
```

Any of these three approaches to grouping selectors and declarations will yield the result shown in Figure 2-4.

Plutonium

Useful for many applications, plutonium can also be dangerous if improperly handled.

Safety Information

When handling plutonium, care must be taken to avoid the formation of a critical mass.

With plutonium, the possibility of implosion is very real, and must be avoided at all costs. This can be accomplished by keeping the various masses separate.

Comments

It’s best to avoid using plutonium at all if it can be avoided.

Figure 2-4. The result of equivalent stylesheets

Class and ID Selectors

So far, we’ve been grouping selectors and declarations together in a variety of ways, but the selectors we’ve been using are very simple ones that refer only to document elements. Type selectors are fine up to a point, but there are times when you need something a little more focused.

In addition to type selectors, there are *class selectors* and *ID selectors*, which let you assign styles in a way that is independent of element type. These selectors can be used on their own or in conjunction with type selectors. However, they only work if you’ve marked up your document appropriately, so using them generally involves a little forethought and planning.

For example, say a document contains a number of warnings. You want each warning to appear in boldface text so that it will stand out. However, you don’t know which elements these warnings will be. Some warnings could be entire paragraphs, while others could be a single item within a lengthy list or a few words in a section of text. So, you can’t define a rule using type selectors of any kind. Suppose you tried this route:

```
p {
  font-weight: bold;
  color: red;
}
```

All paragraphs would be red and bold, not just those that contain warnings. You need a way to select only the text that contains warnings — or, more precisely, a way to select only those elements that are

warnings. How do you do it? You apply styles to parts of the document that have been marked in a certain way, independent of the elements involved, by using class selectors.

Class Selectors

The most common way to apply styles without worrying about the elements involved is to use *class selectors*. Before you can use them, however, you need to modify your actual document markup so that the class selectors will work. Enter the `class` attribute:

```
<p class="warning">When handling plutonium, care must be taken to avoid
the formation of a critical mass.</p>
<p>With plutonium, <span class="warning">the possibility of implosion is
very real, and must be avoided at all costs</span>. This can be accomplished
by keeping the various masses separate.</p>
```

To associate the styles of a class selector with an element, you must assign a `class` attribute the appropriate value. In the previous code block, a `class` value of `warning` was assigned to two elements: the first paragraph and the `span` element in the second paragraph.

To apply styles to these classed elements, you can use a compact notation where the name of a `class` is preceded by a period (`.`):

```
*.warning {font-weight: bold;}
```

When combined with the example markup shown earlier, this simple rule has the effect shown in Figure 2-5. That is, the declaration `font-weight: bold` will be applied to every element that carries a `class` attribute with a value of `warning`.

Plutonium

Useful for many applications, plutonium can also be dangerous if improperly handled.

Safety Information

When handling plutonium, care must be taken to avoid the formation of a critical mass.

With plutonium, **the possibility of implosion is very real, and must be avoided at all costs**. This can be accomplished by keeping the various masses separate.

Comments

It's best to avoid using plutonium **at all** if it can be avoided.

Figure 2-5. Using a class selector

As Figure 2-5 illustrates, the class selector works by directly referencing a value that will be found in the `class` attribute of an element. This reference is *always* preceded by a period (`.`), which marks it as a class selector. The period helps keep the class selector separate from anything with which it might be combined, such as a type selector. For example, you may want boldface warning text only when an entire

paragraph is a warning:

```
p.warning {font-weight: bold;}
```

The selector now matches any `p` elements that have a `class` attribute containing the word `warning`, but no other elements of any kind, classed or otherwise. Since the `span` element is not a paragraph, the rule's selector doesn't match it, and it won't be displayed using boldfaced text.

If you wanted to assign different styles to the `span` element, you could use the selector `span.warning`:

```
p.warning {font-weight: bold;}
span.warning {font-style: italic;}
```

In this case, the warning paragraph is boldfaced, while the warning `span` is italicized. Each rule applies only to a specific type of element/class combination, so it does not leak over to other elements.

Another option is to use a combination of a general class selector and an element-specific class selector to make the styles even more useful, as in the following markup:

```
.warning {font-style: italic;}
span.warning {font-weight: bold;}
```

The results are shown in Figure 2-6.

Plutonium

Useful for many applications, plutonium can also be dangerous if improperly handled.

Safety Information

When handling plutonium, care must be taken to avoid the formation of a critical mass.

*With plutonium, **the possibility of implosion is very real, and must be avoided at all costs**. This can be accomplished by keeping the various masses separate.*

Comments

It's best to avoid using plutonium **at all** if it can be avoided.

Figure 2-6. Using generic and specific selectors to combine styles

In this situation, any warning text will be italicized, but only the text within a `span` element with a `class` of `warning` will be both boldfaced and italicized.

TIP

Notice the format of the general class selector used in the previous example: it's a class name preceded by a period, and without an element name or universal selector. In cases where you only want to select all elements that share a class name, you can omit the universal selector from a class selector without any ill effects. Thus, `*.warning` and `.warning` will have exactly the same effect.

Another thing about class names: they should *never* begin with a number. Browsers will allow you to get away with this, but CSS validators will complain, and it's a bad habit to get into. Thus, you should write `.c8675` in your CSS and `class="c8675"` in your HTML, rather than `.8675` and `class="8675"`. If you must refer to classes that begin with numbers, put a backslash between the period and the first number, like so: `.\8675`.

Multiple Classes

In the previous section, we dealt with `class` values that contained a single word. In HTML, it's possible to have a space-separated list of words in a single `class` value. For example, if you want to mark a particular element as being both urgent and a warning, you could write:

```
<p class="urgent warning">When handling plutonium, care must be taken to
avoid the formation of a critical mass.</p>
<p>With plutonium, <span class="warning">the possibility of implosion is
very real, and must be avoided at all costs</span>. This can be accomplished
by keeping the various masses separate.</p>
```

The order of the words doesn't matter; `warning urgent` would also work and would yield precisely the same results no matter how your CSS is written.

Now let's say you want all elements with a `class` of `warning` to be boldfaced, those with a `class` of `urgent` to be italic, and those elements with both values to have a silver background. This would be written as follows:

```
.warning {font-weight: bold;}
.urgent {font-style: italic;}
.warning.urgent {background: silver;}
```

By chaining two class selectors together, you can select only those elements that have both class names, in any order. As you can see, the HTML source contains `class="urgent warning"` but the CSS selector is written `.warning.urgent`. Regardless, the rule will still cause the "When handling plutonium… " paragraph to have a silver background, as illustrated in Figure 2-7. This happens because the order the words are written in the source document, or in the CSS, doesn't matter. (This is not to say the order of classes is always irrelevant, but we'll get to that later in the chapter.)

Plutonium

Useful for many applications, plutonium can also be dangerous if improperly handled.

Safety Information

When handling plutonium, care must be taken to avoid the formation of a critical mass.

With plutonium, the possibility of implosion is very real, and must be avoided at all costs. This can be accomplished by keeping the various masses separate.

Comments

It's best to avoid using plutonium at all if it can be avoided.

Figure 2-7. Selecting elements with multiple class names

If a multiple class selector contains a name that is not in the space-separated list, then the match will fail. Consider the following rule:

```
p.warning.help {background: red;}
```

As you might expect, the selector will match only those `p` elements with a `class` containing the words `warning` and `help`. Therefore, it will not match a `p` element with just the words `warning` and `urgent` in its `class` attribute. It would, however, match the following:

```
<p class="urgent warning help">Help me!</p>
```

ID Selectors

In some ways, *ID selectors* are similar to class selectors, but there are a few crucial differences. First, ID selectors are preceded by an octothorpe (#)—also known as a pound sign (in the US), hash sign, hash mark, or tic-tac-toe board—instead of a period. Thus, you might see a rule like this one:

```
*#first-para {font-weight: bold;}
```

This rule produces boldfaced text in any element whose `id` attribute has a value of `first-para`.

The second difference is that instead of referencing values of the `class` attribute, ID selectors refer, sensibly enough, to values found in `id` attributes. Here's an example of an ID selector in action:

```
*#lead-para {font-weight: bold;}

<p id="lead-para">This paragraph will be boldfaced.</p>
<p>This paragraph will NOT be bold.</p>
```

Note that the value `lead-para` could have been assigned to any element within the document. In this particular case, it is applied to the first paragraph, but we could have applied it just as easily to the

second or third paragraph. Or an unordered list. Or anything.

The third difference is that there should only be one instance of a given ID value in a document. If you find yourself wanting to apply the same ID to multiple elements in a document, make it a class instead.

As with class selectors, it is possible (and very much the norm) to omit the universal selector from an ID selector. In the previous example, we could also have written with the exact same effect:

```
#lead-para {font-weight: bold;}
```

This is useful for circumstances where you know that a certain ID value will appear in a document, but you don't know the element type on which it will appear. For example, you may know that in any given document, there will be an element with an ID value of `mostImportant`. You don't know whether that most important thing will be a paragraph, a short phrase, a list item, or a section heading. You know only that it will exist in each document, occur in an arbitrary element, and appear no more than once. In that case, you would write a rule like this:

```
#mostImportant {color: red; background: yellow;}
```

This rule would match any of the following elements (which, as noted before, should *not* appear together in the same document because they all have the same ID value):

```
<h1 id="mostImportant">This is important!</h1>
<em id="mostImportant">This is important!</em>
<ul id="mostImportant">This is important!</ul>
```

While HTML standards say each `id` must be unique in a document, CSS doesn't care. If we had erroneously included the HTML shown just now, all three would likely be red with a yellow background because all three match the `#mostImportant` selector.

> **NOTE**
>
> As with class names, IDs should never start with numbers. If you must refer to an ID that begins with a number and you cannot change the ID value in the markup, use a backslash before the first number, as in `#\309`.

Deciding Between Class and ID

You may assign classes to any number of elements, as demonstrated earlier; the class name `warning` was applied to both a `p` and a `span` element, and it could have been applied to many more elements. ID values, on the other hand, should be used once, and only once, within an HTML document. Therefore, if you have an element with an `id` value of `lead-para`, no other element in that document should have an `id` value of `lead-para`.

That's according to the HTML specification, anyway. As noted previously, CSS doesn't care if your HTML is valid or not: it should find however many elements a selector can match. That means that if you sprinkle an HTML document with several elements, all of which have the same value for their ID

attributes, you should get the same styles applied to each.

> **NOTE**
>
> Having more than one of the same ID value in a document also makes DOM scripting more difficult, since functions like `getElementById()` depend on there being one, and only one, element with a given ID value.

Unlike class selectors, ID selectors can't be combined with other IDs, since ID attributes do not permit a space-separated list of words. An ID selector can be combined with itself, though: `#warning#warning` will match the element with an `id` value of `warning`. This should rarely, if ever, be done, but it is possible.

Another difference between `class` and `id` names is that IDs carry more weight when you're trying to determine which styles should be applied to a given element. This will be explained in greater detail in Chapter 4.

Also note that HTML defines class and ID values to be case-sensitive, so the capitalization of your class and ID values must match what's found in your documents. Thus, in the following pairing of CSS and HTML, the element's text will not be boldfaced:

```
p.criticalInfo {font-weight: bold;}

<p class="criticalinfo">Don't look down.</p>
```

Because of the change in case for the letter *i*, the selector will not match the element shown.

On a purely syntactical level, the dot-class notation (e.g., `.warning`) is not guaranteed to work for XML documents. As of this writing, the dot-class notation works in HTML, SVG, and MathML, and it may well be permitted in future languages, but it's up to each language's specification to decide that. The hash-ID notation (e.g., `#lead`) should work in any document language that has an attribute whose value is supposed to be unique within a document.

Attribute Selectors

With both class and ID selectors, what you're really doing is selecting values of elements' attributes. The syntax used in the previous two sections is particular to HTML, SVG, and MathML documents as of this writing. In other markup languages, these class and ID selectors may not be available (as, indeed, those attributes may not be present). To address this situation, CSS2 introduced *attribute selectors*, which can be used to select elements based on their attributes and the values of those attributes. There are four general types of attribute selectors: simple attribute selectors, exact attribute value selectors, partial-match attribute value selectors, and leading-value attribute selectors.

Simple Attribute Selectors

If you want to select elements that have a certain attribute, regardless of that attribute's value, you can use

a *simple attribute selector*. For example, to select all `h1` elements that have a `class` attribute with any value and make their text silver, write:

```
h1[class] {color: silver;}
```

So, given the following markup:

```
<h1 class="hoopla">Hello</h1>
<h1>Serenity</h1>
<h1 class="fancy">Fooling</h1>
```

you get the result shown in Figure 2-8.

Hello

Serenity

Fooling

Figure 2-8. Selecting elements based on their attributes

This strategy is very useful in XML documents, as XML languages tend to have element and attribute names that are specific to their purpose. Consider an XML language that is used to describe planets of the solar system (we'll call it PlanetML). If you want to select all `pml-planet` elements with a `moons` attribute and make them boldface, thus calling attention to any planet that has moons, you would write:

```
pml-planet[moons] {font-weight: bold;}
```

This would cause the text of the second and third elements in the following markup fragment to be boldfaced, but not the first:

```
<pml-planet>Venus</pml-planet>
<pml-planet moons="1">Earth</pml-planet>
<pml-planet moons="2">Mars</pml-planet>
```

In HTML documents, you can use this feature in a number of creative ways. For example, you could style all images that have an `alt` attribute, thus highlighting those images that are correctly formed:

```
img[alt] {outline: 3px solid forestgreen;}
```

This particular example is generally useful more for diagnostic purposes—that is, determining whether images are indeed correctly marked up—than for design purposes.

If you wanted to boldface any element that includes `title` information, which most browsers display as a "tool tip" when a cursor hovers over the element, you could write:

```
*[title] {font-weight: bold;}
```

Similarly, you could style only those anchors (`a` elements) that have an `href` attribute, thus applying the styles to any hyperlink but not to any placeholder anchors.

It is also possible to select elements based on the presence of more than one attribute. You do this by chaining the attribute selectors together. For example, to boldface the text of any HTML hyperlink that has both an `href` and a `title` attribute, you would write:

```
a[href][title] {font-weight: bold;}
```

This would boldface the first link in the following markup, but not the second or third:

```
<a href="https://www.w3.org/" title="W3C Home">W3C</a><br />
<a href="https://developer.mozilla.org">Standards Info</a><br />
<a title="Not a link">dead.letter</a>
```

Selection Based on Exact Attribute Value

You can further narrow the selection process to encompass only those elements whose attributes are a certain value. For example, let's say you want to boldface any hyperlink that points to a certain document on the web server. This would look something like:

```
a[href="http://www.css-discuss.org/about.html"] {font-weight: bold;}
```

This will boldface the text of any `a` element that has an `href` attribute with *exactly* the value *http://www.css-discuss.org/about.html*. Any change at all, even dropping the `www.` part or changing to a secure protocol with `https`, will prevent a match.

Any attribute and value combination can be specified for any element. However, if that exact combination does not appear in the document, then the selector won't match anything. Again, XML languages can benefit from this approach to styling. Let's return to our PlanetML example. Suppose you want to select only those `planet` elements that have a value of `1` for the attribute `moons`:

```
planet[moons="1"] {font-weight: bold;}
```

This would boldface the text of the second element in the following markup fragment, but not the first or third:

```
<planet>Venus</planet>
<planet moons="1">Earth</planet>
<planet moons="2">Mars</planet>
```

As with attribute selection, you can chain together multiple attribute-value selectors to select a single document. For example, to double the size of the text of any HTML hyperlink that has both an `href` with a value of *https://www.w3.org/* and a `title` attribute with a value of `W3C Home`, you would write:

```
a[href="https://www.w3.org/"][title="W3C Home"] {font-size: 200%;}
```

This would double the text size of the first link in the following markup, but not the second or third:

```
<a href="https://www.w3.org/" title="W3C Home">W3C</a><br />
<a href="https://developer.mozilla.org"
  title="Mozilla Developer Network">Standards Info</a><br />
<a href="http://www.example.org/" title="W3C Home">confused.link</a>
```

The results are shown in Figure 2-9.

Figure 2-9. Selecting elements based on attributes and their values

Again, this format requires an *exact* match for the attribute's value. Matching becomes an issue when an attribute selector encounters values that can, in turn, contain a space-separated list of values (e.g., the HTML attribute `class`). For example, consider the following markup fragment:

```
<planet type="barren rocky">Mercury</planet>
```

The only way to match this element based on its exact attribute value is to write:

```
planet[type="barren rocky"] {font-weight: bold;}
```

If you were to write `planet[type="barren"]`, the rule would not match the example markup and thus would fail. This is true even for the `class` attribute in HTML. Consider the following:

```
<p class="urgent warning">When handling plutonium, care must be taken to
avoid the formation of a critical mass.</p>
```

To select this element based on its exact attribute value, you would have to write:

```
p[class="urgent warning"] {font-weight: bold;}
```

This is *not* equivalent to the dot-class notation covered earlier, as we will see in the next section. Instead, it selects any `p` element whose `class` attribute has *exactly* the value `"urgent warning"`, with the words in that order and a single space between them. It's effectively an exact string match, whereas when using class selector, the class order doesn't matter.

Also, be aware that ID selectors and attribute selectors that target the `id` attribute are not precisely the same. In other words, there is a subtle but crucial difference between `h1#page-title` and `h1[id="page-title"]`. This difference is explained in Chapter 4.

Selection Based on Partial Attribute Values

Odds are that you'll sometimes want to select elements based on portions of their attribute values, rather than the full value. For such situations, CSS offers a variety of options for matching substrings in an attribute's value. These are summarized in Table 2-1.

Table 2-1. Substring matching with attribute selectors

Type	Description
`[foo~="bar"]`	Selects any element with an attribute `foo` whose value contains the word `bar` in a space-separated list of words
`[foo*="bar"]`	Selects any element with an attribute `foo` whose value *contains* the substring `bar`
`[foo^="bar"]`	Selects any element with an attribute `foo` whose value *begins* with `bar`
`[foo$="bar"]`	Selects any element with an attribute `foo` whose value *ends* with `bar`
`[foo|="bar"]`	Selects any element with an attribute `foo` whose value *starts* with `bar` followed by a dash (U+002D) or whose value is exactly equal to `bar`

The last of these attribute selectors that match on a partial subset of an element's attribute value is actually easier to show than it is to describe. Consider the following rule:

```
*[lang|="en"] {color: white;}
```

This rule will select any element whose `lang` attribute is equal to `en` or begins with `en-`. Therefore, the first three elements in the following example markup would be selected, but the last two would not:

```
<h1 lang="en">Hello!</h1>
<p lang="en-us">Greetings!</p>
<div lang="en-au">G'day!</div>
<p lang="fr">Bonjour!</p>
<h4 lang="cy-en">Jrooana!</h4>
```

In general, the form `[att|="val"]` can be used for any attribute and its values. Let's say you have a series of figures in an HTML document, each of which has a filename like *figure-1.gif* and *figure-3.jpg*. You can match all of these images using the following selector:

```
img[src|="figure"] {border: 1px solid gray;}
```

Or, if you're creating a CSS framework or pattern library, instead of creating redundant classes like `"btn btn-small btn-arrow btn-active"`, you can declare `"btn-small-arrow-active"`, and target the class of elements with:

```
*[class|="btn"] { border-radius: 5px;}

<button class="btn-small-arrow-active">Click Me</button>
```

The most common use for this type of attribute selector is to match language values, as demonstrated in an upcoming section, "The :lang and :dir Pseudo-Classes".

Matching one word in a space-separated list

For any attribute that accepts a space-separated list of words, it is possible to select elements based on the presence of any one of those words. The classic example in HTML is the `class` attribute, which can accept one or more words as its value. Consider our usual example text:

```
<p class="urgent warning">When handling plutonium, care must be taken to
avoid the formation of a critical mass.</p>
```

Let's say you want to select elements whose `class` attribute contains the word `warning`. You can do this with an attribute selector:

```
p[class~="warning"] {font-weight: bold;}
```

Note the presence of the tilde (~) in the selector. It is the key to selection based on the presence of a space-separated word within the attribute's value. If you omit the tilde, you would have an exact value-matching attribute selector, as discussed in the previous section.

This selector construct is equivalent to the dot-class notation discussed in "Deciding Between Class and ID". Thus, `p.warning` and `p[class~="warning"]` are equivalent when applied to HTML documents. Here's an example that is an HTML version of the "PlanetML" markup seen earlier:

```
<span class="barren rocky">Mercury</span>
<span class="cloudy barren">Venus</span>
<span class="life-bearing cloudy">Earth</span>
```

To italicize all elements with the word `barren` in their `class` attribute, you write:

```
span[class~="barren"] {font-style: italic;}
```

This rule's selector will match the first two elements in the example markup and thus italicize their text, as shown in Figure 2-10. This is the same result we would expect from writing `span.barren {font-style: italic;}`.

Mercury Venus Earth

Figure 2-10. Selecting elements based on portions of attribute values

So why bother with the tilde-equals attribute selector in HTML? Because it can be used for any attribute, not just `class`. For example, you might have a document that contains a number of images, only some of which are figures. You can use a partial-match value attribute selector aimed at the `title` text to select only those figures:

```
img[title~="Figure"] {border: 1px solid gray;}
```

This rule selects any image whose `title` text contains the word `Figure` (but not `figure`, as class names are case-sensitive). Therefore, as long as all your figures have `title` text that looks something like “Figure 4. A bald-headed elder statesman,” this rule will match those images. For that matter, the selector `img[title~="Figure"]` will also match a title attribute with the value “How to Figure Out Who’s in Charge.” Any image that does not have a `title` attribute, or whose `title` value doesn’t contain the word “Figure,” won’t be matched.

Matching a substring within an attribute value

Sometimes you want to select elements based on a portion of their attribute values, but the values in question aren’t space-separated lists of words. In these cases, you can use the asterisk-equals substring matching form `[attr*="val"]` to match substrings that appear anywhere inside the attribute values. For example, the following CSS matches any `span` element whose `class` attribute contains the substring `cloud`, so both “cloudy” planets are matched, as shown in Figure 2-11:

```
span[class*="cloud"] {font-style: italic;}
```

```
<span class="barren rocky">Mercury</span>
<span class="cloudy barren">Venus</span>
<span class="life-bearing cloudy">Earth</span>
```

Mercury *Venus Earth*

Figure 2-11. Selecting elements based on substrings within attribute values

Note the presence of the asterisk (`*`) in the selector. It’s the key to selecting elements based on the presence of a substring within an attribute’s value. To be clear, it is **not** related to the universal selector, other than it uses the same character.

As you can imagine, there are many useful applications for this particular capability. For example, suppose you wanted to specially style any links to the World Wide Web Consortium’s website. Instead of classing them all and writing styles based on that class, you could instead write the following rule:

```
a[href*="w3.org"] {font-weight: bold;}
```

You aren’t confined to the `class` and `href` attributes. Any attribute is up for grabs here: `title`, `alt`, `src`, `id`…if the attribute has a value, you can style based on a substring within that value. The following rule draws attention to any image with the string `“space”` in its source URL:

```
img[src*="space"] {outline: 5px solid red;}
```

Similarly, the following rule draws attention to `<input>` elements that have a title telling the user what to do, along with any other input whose title contains the substring "format" in its title:

```
input[title*="format"] {background-color: #dedede;}

<input type="tel"
    title="Telephone number should be formatted as XXX-XXX-XXXX"
    pattern="\d{3}\-\d{3}\-\d{4}">
```

A common use for the general substring attribute selector is to match a section of a class in pattern library class names. Elaborating on the last example, we can target any class name that starts with `"btn"` followed by a dash, and that contains the substring `"arrow"` preceded by a dash, by using the pipe-equals attribute selector:

```
*[class|="btn"][class*="-arrow"]:after { content: "▼";}

<button class="btn-small-arrow-active">Click Me</button>
```

The matches are exact: if you include whitespace in your selector, then whitespace must also be present in an attribute's value. The attribute names and values must be case-sensitive only if the underlying document language requires case sensitivity. Class names, titles, URLs, and ID values are all case-sensitive, but HTML attribute keyword values, such as input types, are not:

```
input[type="CHeckBoX"] {margin-right: 10px;}

<input type="checkbox" name="rightmargin" value="10px">
```

Matching a substring at the beginning of an attribute value

In cases where you want to select elements based on a substring at the beginning of an attribute value, then the caret-equals attribute selector pattern `[att^="val"]` is what you're seeking. This can be particularly useful in a situation where you want to style types of links differently, as illustrated in Figure 2-12.

```
a[href^="https:"] {font-weight: bold;}
a[href^="mailto:"] {font-style: italic;}
```

W3C home page
My banking login screen
O'Reilly & Associates home page
Send mail to me@example.com
Wikipedia (English)

Figure 2-12. Selecting elements based on substrings that begin attribute values

Another use case is when you want to style all images in an article that are also figures, as in the figures you see throughout this text. Assuming that the alt text of each figure begins with text in the pattern "Figure 5"—which is an entirely reasonable assumption in this case—then you can select only those images with the caret-equals attribute selector:

```
img[alt^="Figure"] {border: 2px solid gray;  display: block; margin: 2em auto;}
```

The potential drawback here is that *any* `img` element whose `alt` starts with “Figure” will be selected, whether or not it’s meant to be an illustrative figure. The likeliness of that occurring depends on the document in question.

Another use case is selecting all of the calendar events that occur on Mondays. In this case, let’s assume all of the events have a `title` attribute containing a date in the format “Monday, March 5th, 2012.” Selecting them all is a simple matter of `[title^="Monday"]`.

Matching a substring at the end of an attribute value

The mirror image of beginning-substring matching is ending-substring matching, which is accomplished using the `[att$="val"]` pattern. A very common use for this capability is to style links based on the kind of resource they target, such as separate styles for PDF documents, as illustrated in Figure 2-13.

```
a[href$=".pdf"] {font-weight: bold;}
```

Home page
FAQ
Printable instructions
Detailed warranty
Contact us

Figure 2-13. Selecting elements based on substrings that end attribute values

Similarly, you could (for whatever reason) select images based on their image format with the dollar-equals attribute selector:

```
img[src$=".gif"] {...}
img[src$=".jpg"] {...}
img[src$=".png"] {...}
```

To continue the calendar example from the previous section, it would be possible to select all of the events occurring within a given year using a selector like `[title$="2015"]`.

> **NOTE**
>
> You may have noticed that we’ve quoted all the attribute values in the attribute selectors. Quoting is required if the value includes any special characters, begins with a dash or digit, or is otherwise invalid as an identifier and needs to be quoted as a string. To be safe, we recommend always quoting attribute values in attribute selectors, even though it is only required to make strings out of invalid identifiers.

The Case Insensitivity Identifier

Including an `i` before the closing bracket of an attribute selector will allow that selector to match attribute values case-insensitively, regardless of document language rules.

For example, suppose you want to select all links to PDF documents, but you don’t know if they’ll end in *.pdf*, *.PDF*, or even *.Pdf*. Here’s how:

```
a[href$='.PDF' i]
```

Adding that humble little `i` means the selector will match any `a` element whose `href` attribute's value ends in `.pdf`, regardless of the capitalization of the letters P, D, and F.

This case-insensitivity option is available for all the attribute selectors we've covered. Note, however, that this only applies to the *values* in the attribute selectors. It does not enforce case insensitivity on the attribute names themselves. Thus, in a case-sensitive language, `planet[type*="rock" i]` will match all of the following:

```
<planet type="barren rocky">Mercury</planet>
<planet type="cloudy ROCKY">Venus</planet>
<planet type="life-bearing Rock">Earth</planet>
```

It will *not* match the following element, because the attribute `TYPE` isn't matched by `type` in XML:

```
<planet TYPE="dusty rock">Mars</planet>
```

Again, that's in languages that enforce case sensitivity in the element and attribute syntax. XHTML was one such language. In languages that are case-insensitive, like HTML 5, this isn't an issue.

> **NOTE**
>
> There is a proposed mirror identifier, `s`, which enforces case sensitivity. As of early 2022, it was only supported by the Firefox family of browsers.

Using Document Structure

CSS is so capable because it uses the structure of documents to determine appropriate styles and how to apply them. Let's take a moment to discuss structure before moving on to more powerful forms of selection.

Understanding the Parent-Child Relationship

To understand the relationship between selectors and documents, we need to once again examine how documents are structured. Consider this very simple HTML document:

```
<!doctype html>
<html>
<head>
 <meta charset="utf-8">
 <title>Meerkat Central</title>
</head>
<body>
 <h1>Meerkat <em>Central</em></h1>
 <p>
 Welcome to Meerkat <em>Central</em>, the <strong>best meerkat web site
 on <a href="inet.html">the <em>entire</em> Internet</a></strong>!</p>
```

```
  <ul>
   <li>We offer:
    <ul>
     <li><strong>Detailed information</strong> on how to adopt a meerkat</li>
     <li>Tips for living with a meerkat</li>
     <li><em>Fun</em> things to do with a meerkat, including:
      <ol>
       <li>Playing fetch</li>
       <li>Digging for food</li>
       <li>Hide and seek</li>
      </ol>
     </li>
    </ul>
   </li>
   <li>...and so much more!</li>
  </ul>
  <p>
  Questions? <a href="mailto:suricate@meerkat.web">Contact us!</a>
  </p>
 </body>
 </html>
```

Much of the power of CSS is based on the *parent-child relationship* of elements. HTML documents (actually, most structured documents of any kind) are based on a hierarchy of elements, which is visible in the “tree” view of the document (see Figure 2-14). In this hierarchy, each element fits somewhere into the overall structure of the document. Every element in the document is either the *parent* or the *child* of another element, and it’s often both. If a parent has more than one child, those children are *siblings*.

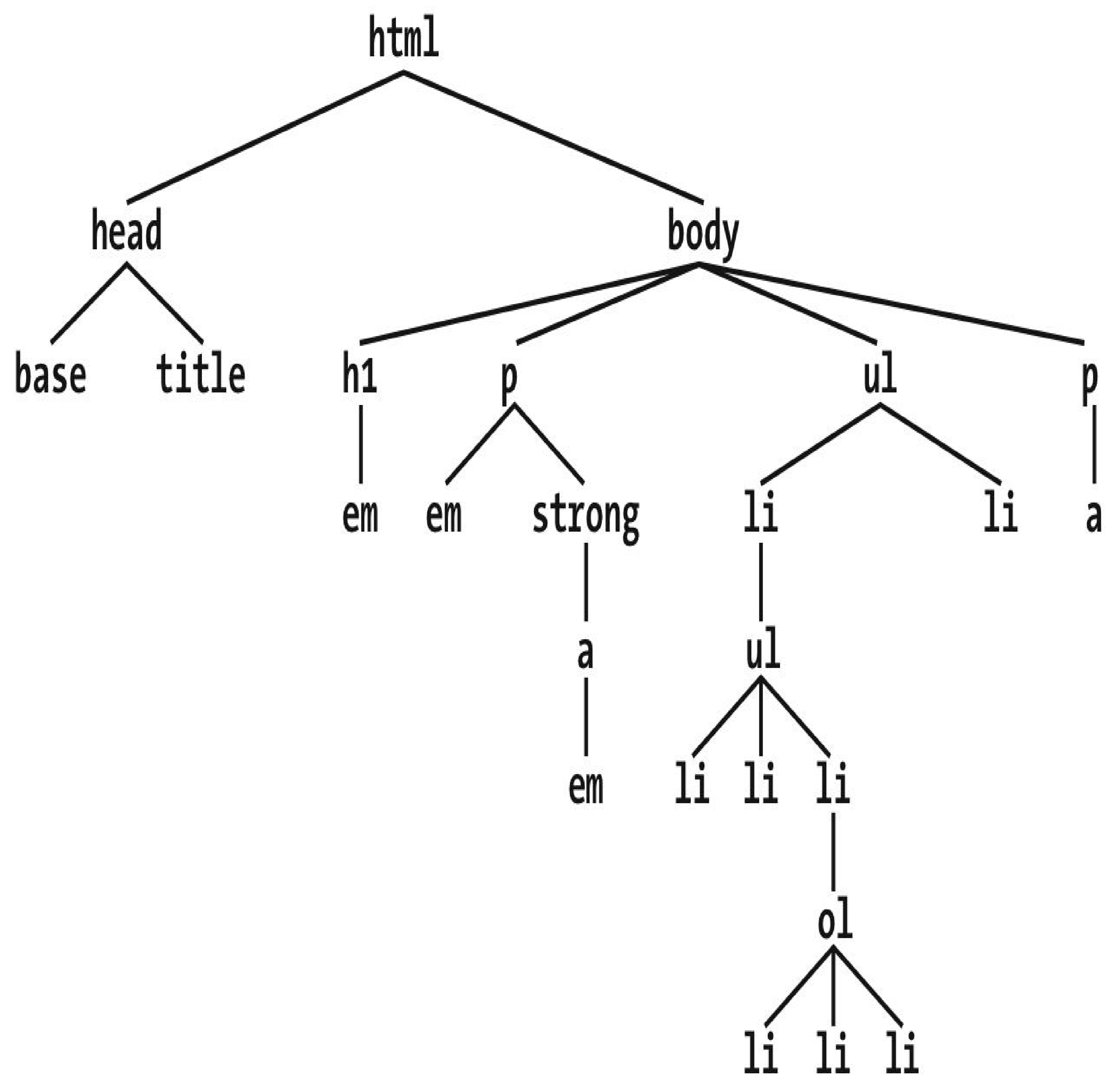


Figure 2-14. A document tree structure

An element is said to be the parent of another element if it appears directly above that element in the document hierarchy. For example, in Figure 2-14, the first `p` element from the left is parent to the `em` and `strong` elements, while `strong` is parent to an anchor (`a`) element, which is itself parent to another `em` element. Conversely, an element is the child of another element if it is directly beneath the other element. Thus, the anchor element on the far right side of Figure 2-14 is the child of a `p` element, which is in turn child to the `body` element, and so on.

The terms "parent" and "child" are specific applications of the terms *ancestor* and *descendant*. There is a difference between them: in the tree view, if an element is exactly one level above or below another, then they have a parent-child relationship. If the path from one element to another is traced through two or

more levels, the elements have an ancestor-descendant relationship, but not a parent-child relationship. (A child is also a descendant, and a parent is also an ancestor.) In Figure 2-14, the uppermost `ul` element is parent to two `li` elements, but the uppermost `ul` is also the ancestor of every element descended from its `li` element, all the way down to the most deeply nested `li` elements. Those `li` elements, children of the `ol`, are siblings.

Also, in Figure 2-14, there is an anchor that is a child of `strong`, but also a descendant of the `p`, `body`, and `html` elements. The `body` element is an ancestor of everything that the browser will display by default, and the `html` element is ancestor to the entire document. For this reason, in an HTML or XHTML document, the `html` element is also called the *root element*.

Descendant Selectors

The first benefit of understanding this model is the ability to define *descendant selectors*. Defining descendant selectors is the act of creating rules that operate in certain structural circumstances, but not others. As an example, let's say you want to style only those `em` elements that are descended from `h1` elements. To do so, write the following:

```
h1 em {color: gray;}
```

This rule will make gray any text in an `em` element that is the descendant of an `h1` element. Other `em` text, such as that found in a paragraph or a block quote, will not be selected by this rule. Figure 2-15 illustrates the result.

Meerkat *Central*

Figure 2-15. Selecting an element based on its context

In a descendant selector, the selector side of a rule is composed of two or more space-separated selectors. The space between the selectors is an example of a *combinator*. Each space combinator can be translated as "found within," "which is part of," or "that is a descendant of," but only if you read the selector right to left. Thus, `h1 em` can be translated as, "Any `em` element that is a descendant of an `h1` element."

To read the selector left to right, you might phrase it something like, "Any `h1` that contains an `em` will have the following styles applied to the `em`." That's much more verbose and confusing, and it's why we, like the browser, read selectors from right to left.

You aren't limited to two selectors. For example:

```
ul ol ul em {color: gray;}
```

In this case, as Figure 2-16 shows, any emphasized text that is part of an unordered list that is part of an ordered list that is itself part of an unordered list (yes, this is correct) will be gray. This is obviously a very specific selection criterion.

- It's a list
- A right smart list
    1. Within, another list
        - This is *deep*
        - So *very* deep
    2. A list of lists to see
- And all the lists for me!

Figure 2-16. A very specific descendant selector

Descendant selectors can be extremely powerful. Let's consider a common example. Assume you have a document with a sidebar and a main area. The sidebar has a blue background, the main area has a white background, and both areas include lists of links. You can't set all links to be blue because they'd be impossible to read in the sidebar, and you also can't set all links to white because they'd disappear in the main part of the page.

The solution: descendant selectors. In this case, you give the element that contains your sidebar a class of `sidebar` and enclose the main part of the page in a `main` element. Then, you write styles like this:

```
.sidebar {background: blue;}
main {background: white;}
.sidebar a:any-link {color: white;}
main a:any-link {color: blue;}
```

Figure 2-17 shows the result.

Blogs

These are the web logs ("blogs") I visit a lot. They're all written by people who know a lot about Web design and CSS in general. By reading them I can get a sense of the trends in design and thinking about document structure.

Figure 2-17. Using descendant selectors to apply different styles to the same type of element

> **NOTE**
>
> `:any-link` refers to both visited and unvisited links. We'll talk about it in detail in Chapter 3.

Here's another example: let's say that you want gray to be the text color of any `b` (boldface) element that is part of a `blockquote` and for any bold text that is found in a normal paragraph:

```
blockquote b, p b {color: gray;}
```

The result is that the text within `b` elements that are descended from paragraphs or block quotes will be gray.

One overlooked aspect of descendant selectors is that the degree of separation between two elements can

be practically infinite. For example, if you write `ul em`, that syntax will select any `em` element descended from a `ul` element, no matter how deeply nested the `em` may be. Thus, `ul em` would select the `em` element in the following markup:

```
<ul>
  <li>List item 1
    <ol>
      <li>List item 1-1</li>
      <li>List item 1-2</li>
      <li>List item 1-3
        <ol>
          <li>List item 1-3-1</li>
          <li>List item <em>1-3-2</em></li>
          <li>List item 1-3-3</li>
        </ol>
      </li>
      <li>List item 1-4</li>
    </ol>
  </li>
</ul>
```

A more subtle aspect of descendant selectors is that they have no notion of element proximity. In other words, the closeness of two elements within the document tree has no bearing on whether a rule applies or not. This is important when it comes to specificity (which we'll cover in the next chapter) and when considering rules that might appear to cancel each other out.

For example, consider the following (which contains a selector type we'll discuss in the upcoming section, "The Negation Pseudo-Class"):

```
div:not(.help) span {color: gray;}
div.help span {color: red;}

<div class="help">
   <div class="aside">
      This text contains <span>a span element</span> within.
   </div>
</div>
```

What the CSS says, in effect, is "any `span` inside a `div` that doesn't have a `class` containing the word `help` should be gray" in the first rule, and "any `span` inside a `div` whose `class` contains the word `help`" in the second rule. In the given markup fragment, *both* rules apply to the `span` shown.

Because the two rules have equal weight and the "red" rule is written last, it wins out and the `span` is red. The fact that the `div class="aside"` is "closer to" the `span` than the `div class="help"` is irrelevant. Again: descendant selectors have no notion of element proximity. Both rules match, only one color can be applied, and due to the way CSS works, red is the winner here. (We'll discuss why that's so in the next chapter.)

> **NOTE**
>
> As of early 2022, there were proposals to add element-proximity awareness to CSS via "selector scoping," but the proposals were still being actively revised and may not come to fruition.

Selecting Children

In some cases, you don't want to select an arbitrarily descended element. Rather, you want to narrow your range to select an element that is specifically a child of another element. You might, for example, want to select a `strong` element only if it is a child (as opposed to any other level of descendant) of an `h1` element. To do this, you use the *child combinator*, which is the greater-than symbol (`>`):

```
h1 > strong {color: red;}
```

This rule will make red the `strong` element shown in the first `h1`, but not the second:

```
<h1>This is <strong>very</strong> important.</h1>
<h1>This is <em>really <strong>very</strong></em> important.</h1>
```

Read right to left, the selector `h1 > strong` translates as, "Selects any `strong` element that is a direct child of an `h1` element." The child combinator can be optionally surrounded by whitespace. Thus, `h1 > strong`, `h1> strong`, and `h1>strong` are all equivalent. You can use or omit whitespace as you wish.

When viewing the document as a tree structure, we can see that a child selector restricts its matches to elements that are directly connected in the tree. Figure 2-18 shows part of a document tree.

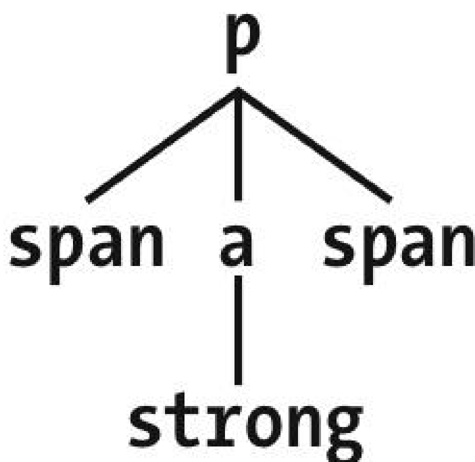


Figure 2-18. A document tree fragment

In this tree fragment, you can pick out parent-child relationships. For example, the `a` element is parent to the `strong`, but it is child to the `p` element. You could match elements in this fragment with the selectors `p > a` and `a > strong`, but not `p > strong`, since the `strong` is a descendant of the `p` but not its child.

You can also combine descendant and child combinations in the same selector. Thus, `table.summary td > p` will select any `p` element that is a *child* of a `td` element that is itself *descended* from a `table`

element that has a `class` attribute containing the word `summary`.

Selecting Adjacent Sibling Elements

Let's say you want to style the paragraph immediately after a heading, or give a special margin to a list that immediately follows a paragraph. To select an element that immediately follows another element with the same parent, you use the *adjacent-sibling combinator*, represented as a plus symbol (+). As with the child combinator, the symbol can be surrounded by whitespace, or not, at the author's discretion.

To remove the top margin from a paragraph immediately following an `h1` element, write:

```
h1 + p {margin-top: 0;}
```

The selector is read as, "Select any `p` element that immediately follows an `h1` element that *shares a parent* with the `p` element."

To visualize how this selector works, let's once again consider a fragment of a document tree, shown in Figure 2-19.

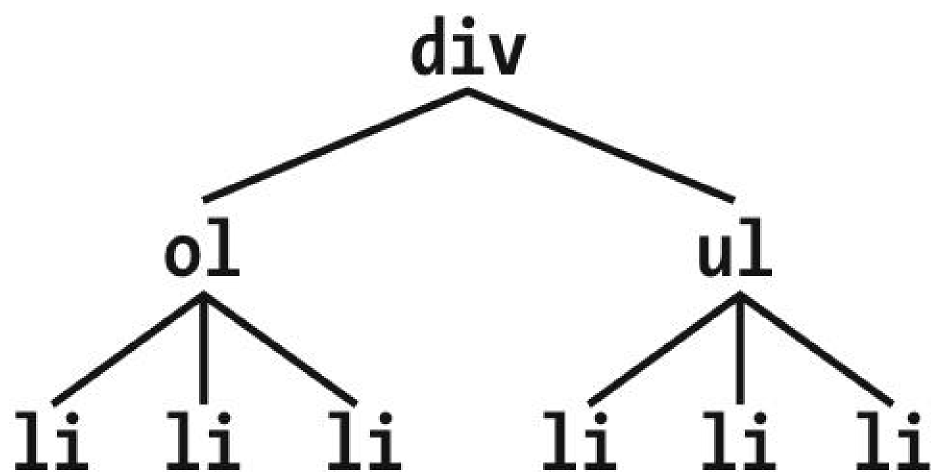


Figure 2-19. Another document tree fragment

In this fragment, a pair of lists descends from a `div` element, one ordered and the other not, each containing three list items. Each list is an adjacent sibling, and the list items themselves are also adjacent siblings. However, the list items from the first list are *not* siblings of the second, as the two sets of list items do not share the same parent element. (At best, they're cousins, and CSS has no cousin selector.)

Remember that you can select the second of two adjacent siblings only with a single combinator. Thus, if you write `li + li {font-weight: bold;}`, only the second and third items in each list will be boldfaced. The first list items will be unaffected, as illustrated in Figure 2-20.

1. List item 1
2. **List item 1**
3. **List item 1**

This is some text that is part of the 'div'.

- A list item
- **Another list item**
- **Yet another list item**

Figure 2-20. Selecting adjacent siblings

To work properly, CSS requires that the two elements appear in “source order.” In our example, an `ol` element is followed by a `ul` element. This allows us to select the second element with `ol + ul`, but we cannot select the first using the same syntax. For `ul + ol` to match, an ordered list must immediately follow an unordered list.

Keep in mind that text content between two elements does *not* prevent the adjacent-sibling combinator from working. Consider this markup fragment, whose tree view would be the same as that shown in Figure 2-18:

```
<div>
  <ol>
    <li>List item 1</li>
    <li>List item 1</li>
    <li>List item 1</li>
  </ol>
  This is some text that is part of the 'div'.
  <ul>
    <li>A list item</li>
    <li>Another list item</li>
    <li>Yet another list item</li>
  </ul>
</div>
```

Even though there is text between the two lists, we can still match the second list with the selector `ol + ul`. That’s because the intervening text is not contained with a sibling element, but is instead part of the parent `div`. If we wrapped that text in a paragraph element, it would then prevent `ol + ul` from matching the second list. Instead, we might have to write something like `ol + p + ul`.

As the following example illustrates, the adjacent-sibling combinator can be used in conjunction with other combinators:

```
html > body table + ul{margin-top: 1.5em;}
```

The selector translates as, “Selects any `ul` element that immediately follows a sibling `table` element that is descended from a `body` element that is itself a child of an `html` element.”

As with all combinators, you can place the adjacent-sibling combinator in a more complex setting, such as `div#content h1 + div ol`. That selector is read as, “Selects any `ol` element that is descended from a `div` when the `div` is the adjacent sibling of an `h1` which is itself descended from a `div` whose `id` attribute has a value of `content`.”

Selecting Following Siblings

The *general sibling combinator* lets you select any element that follows another element when both elements share the same parent, represented using the tilde (~) combinator.

As an example, to italicize any `ol` that follows an `h2` and also shares a parent with the `h2`, you'd write `h2 ~ ol {font-style: italic;}`. The two elements do not have to be adjacent siblings, although they can be adjacent and still match this rule. The result of applying this rule to the following markup is shown in Figure 2-21:

```
<div>
  <h2>Subheadings</h2>
  <p>It is the case that not every heading can be a main heading.  Some headings
  must be subheadings.  Examples include:</p>
  <ol>
    <li>Headings that are less important</li>
    <li>Headings that are subsidiary to more important headlines</li>
    <li>Headings that like to be dominated</li>
  </ol>
  <p>Let's restate that for the record:</p>
  <ol>
    <li>Headings that are less important</li>
    <li>Headings that are subsidiary to more important headlines</li>
    <li>Headings that like to be dominated</li>
  </ol>
</div>
```

As you can see, both ordered lists are italicized. That's because both of them are `ol` elements that follow an `h2` with which they share a parent (the `div`).

Subheadings

It is the case that not every heading can be a main heading. Some headings must be subheadings. Examples include:

1. *Headings that are less important*
2. *Headings that are subsidiary to more important headlines*
3. *Headings that like to be dominated*

Let's restate that for the record:

1. *Headings that are less important*
2. *Headings that are subsidiary to more important headlines*
3. *Headings that like to be dominated*

Figure 2-21. Selecting following siblings

Summary

By using selectors based on the document's language, authors can create CSS rules that apply to a large number of similar elements just as easily as they can construct rules that apply in very narrow circumstances. The ability to group together both selectors and rules keeps stylesheets compact and flexible, which incidentally leads to smaller file sizes and faster download times.

Selectors are the one thing that user agents usually must get right because the inability to correctly interpret selectors pretty much prevents a user agent from using CSS at all. On the flip side, it’s crucial for authors to correctly write selectors because errors can prevent the user agent from applying the styles as intended. An integral part of correctly understanding selectors and how they can be combined is a strong grasp of how selectors relate to document structure and how mechanisms—such as inheritance and the cascade itself—come into play when determining how an element will be styled.

The selectors we covered in this chapter aren’t the end of the story, though. They’re not even half the story. In the next chapter, we’ll dive into the powerful and ever-expanding world of pseudo-class and pseudo-element selectors.

Chapter 3. Pseudo-Class and -Element Selectors

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at rfernando@oreilly.com.

In the previous chapter, we saw how selectors can match a single element, or a collection of elements, using fairly simple expressions that match the structure of the document. Those are great if your needs are just to style based on structure, but what if you need to style part of a document based on its current state? Or if you want to select all the form elements that are disabled, or those that are required for form submission to be allowed? For those things, and a great deal more, there are the pseudo-class and pseudo-element selectors.

Pseudo-Class Selectors

Pseudo-class selectors let you assign styles to what are, in effect, phantom classes inferred by the state of certain elements, or markup patterns within the document, or even by the state of the document itself.

The phrase "phantom classes" might seem a little odd, but it really is the best way to think of how pseudo-classes work. For example, suppose you wanted to highlight every other row of a data table. You could do that by marking up every other row something like `class="even"` and then writing CSS to highlight rows with that class—or (as we'll soon see) you could use a pseudo-class selector to achieve the same effect, one which will act as if you'd added all those classes to the markup even though you haven't.

There's an aspect of pseudo-classes that needs to be made explicit here: pseudo-classes always refer to the element to which they're attached, and to no other. Seems like a weirdly obvious thing to say, right? The reason to make it explicit is that for some pseudo-classes, it's a common error to think they are descriptors that refer to descendant elements.

To illustrate this, Eric would like to share a personal anecdote.

Example 3-1.

When my first child was born in 2003, I announced it online, as one does. A number of people responded

with congratulations and CSS jokes, chief among them the selector `#ericmeyer:first-child` (we'll get to `:first-child` in just a bit). The problem there is that selector would select me, not my daughter, and only if I were the first child of my own parents (which, as it happens, I am). To properly select *my* first child, that selector would need to be `#ericmeyer > :first-child`.

The confusion is understandable, which is why we're addressing it here. Reminders will be found throughout the following sections. Just always keep in mind that the effect of pseudo-classes is to apply a sort of a "phantom class" to the element to which they're attached, and you should be OK.

All pseudo-classes, without exception, are a word or hyphenated phrase preceded by a single colon (`:`), and they can appear anywhere in a selector.

Combining Pseudo-Classes

Before we really get started, a word about chaining. CSS makes it possible to combine ("chain") pseudo-classes together. For example, you can make unvisited links red when they're hovered and visited links maroon when they are hovered:

```
a:link:hover {color: red;}
a:visited:hover {color: maroon;}
```

The order you specify doesn't actually matter; you could also write `a:hover:link` to the same effect as `a:link:hover`. It's also possible to assign separate hover styles to unvisited and visited links that are in another language—for example, German:

```
a:link:hover:lang(de) {color: gray;}
a:visited:hover:lang(de) {color: silver;}
```

Be careful not to combine mutually exclusive pseudo-classes. For example, a link cannot be both visited and unvisited, so `a:link:visited` doesn't make any sense and will never match anything.

Structural Pseudo-Classes

The first set of pseudo-classes we'll explore are structural in nature; that is, they refer to the markup structure of the document. Most of them depend on patterns within the markup, such as choosing every third paragraph, but others allow you to address specific types of elements.

Selecting the root element

This is the quintessence of structural simplicity: the pseudo-class `:root` selects the root element of the document. In HTML, this is *always* the `html` element. The real benefit of this selector is found when writing stylesheets for XML languages, where the root element may be different in every language—for example, in SVG it's the `svg` element, and in our earlier PlanetML examples it was the `pml` element—or even when you have more than one possible root element within a single language (though not a single document!).

Here's an example of styling the root element in HTML, as illustrated in Figure 3-1:

```
:root {border: 10px dotted gray;}
body {border: 10px solid black;}
```

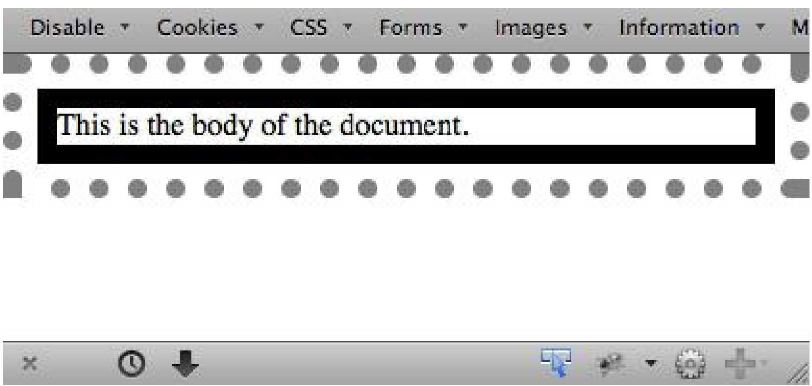


Figure 3-1. Styling the root element

In HTML documents, you can always select the `html` element directly, without having to use the `:root` pseudo-class. There is a difference between the two selectors in terms of specificity, which we'll cover in Chapter 4, but otherwise they'll have the same effect.

Selecting empty elements

With the pseudo-class `:empty`, you can select any element that has no children of any kind, *including* text nodes, which covers both text and whitespace. This can be useful in suppressing elements that a CMS (Content Management System) has generated without filling in any actual content. Thus, `p:empty {display: none;}` would prevent the display of any empty paragraphs.

Note that in order to be matched, an element must be, from a parsing perspective, truly empty—no whitespace, visible content, or descendant elements. Of the following elements, only the first and last would be matched by `p:empty`:

```
<p></p>
<p> </p>
<p>
</p>
<p><!—-a comment--></p>
```

The second and third paragraphs are not matched by `:empty` because they are not empty: they contain, respectively, a single space and a single newline character. Both are considered text nodes, and thus prevent a state of emptiness. The last paragraph matches because comments are not considered content, not even whitespace. But put even one space or newline to either side of that comment, and `p:empty` would fail to match.

You might be tempted to just style all empty elements with something like `*:empty {display: none;}`, but there's a hidden catch: `:empty` matches HTML's empty elements, like `img`, `hr`, `br`, and `input`. It could even match `textarea`, unless you insert some default text into the `textarea` element. Thus, in terms of matching elements, `img` and `img:empty` are effectively the same. (They are different in terms of specificity, which we'll cover in the next chapter.)

NOTE

As of early 2022, `:empty` is unique in that it's the only CSS selector that takes text nodes into consideration when determining matches. It's also supposed to ignore whitespace inside elements, but no browser had supported that behavior as of this writing.

Selecting only children

If you've ever wanted to select all the images that are wrapped by a hyperlink element, the `:only-child` pseudo-class is for you. It selects elements when they are the only child element of another element. So let's say you want to add a border to any image that's the only child of another element. You'd write:

```
img:only-child {border: 1px solid black;}
```

This would match any image that meets those criteria. Therefore, if you had a paragraph which contained an image and no other child elements, the image would be selected regardless of all the text surrounding it. If what you're really after is images that are sole children and found inside hyperlinks, then you just modify the selector like so (which is illustrated in Figure 3-2):

```
a[href] img:only-child {border: 2px solid black;}

<a href="http://w3.org/"><img src="w3.png" alt="W3C"></a>
<a href="http://w3.org/"><img src="w3.png" alt=""> The W3C</a>
<a href="http://w3.org/"><img src="w3.png" alt=""> <em>The W3C</em></a>
```

Figure 3-2. Selecting images that are only children inside links

There are two things to remember about `:only-child`. The first is that you *always* apply it to the element you want to be an only child, not to the parent element, as explained earlier. And that brings up the second thing to remember, which is that when you use `:only-child` in a descendant selector, you aren't restricting the elements listed to a parent-child relationship.

To go back to the hyperlinked-image example, `a[href] img:only-child` matches any image that is an only child and is descended from an `a` element, whether or not it's a *child* of an `a` element. To match, the element image must be the only child of its direct parent and also a descendant of an `a` element with an `href` attribute, but that parent can itself be a descendant of the same `a` element`. Therefore, all three of the images in the following would be matched, as shown in Figure 3-3:

```
a[href] img:only-child {border: 5px solid black;}

<a href="http://w3.org/"><img src="w3.png" alt="W3C"></a>
<a href="http://w3.org/"><span><img src="w3.png" alt="W3C"></span></a>
<a href="http://w3.org/">A link to <span>the <img src="w3.png" alt="">
```

```
    web</span> site</a>
```

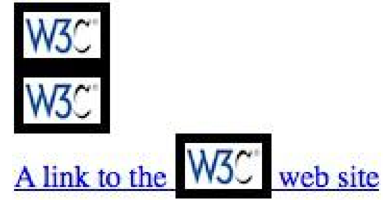


Figure 3-3. Selecting images that are only children inside links

In each case, the image is the only child element of its parent, and it is also descended from an `a` element. Thus, all three images are matched by the rule shown. If you want to restrict the rule so that it matched images that were the only children of `a` elements, then add the child combinator to yield `a[href] > img:only-child`. With that change, only the first of the three images shown in Figure 3-3 would be matched.

Only-of-type selection

That’s all great, but what if you want to match images that are the only images inside hyperlinks, but there are other elements in there with them? Consider the following:

```
<a href="http://w3.org/"><b>•</b><img src="w3.png" alt="W3C"></a>
```

In this case, we have an `a` element that has two children: `b` and `img`. That image, no longer being the only child of its parent (the hyperlink), can never be matched using `:only-child`. However, it *can* be matched using `:only-of-type`. This is illustrated in Figure 3-4:

```
a[href] img:only-of-type {border: 5px solid black;}

<a href="http://w3.org/"><b>•</b><img src="w3.png" alt="W3C"></a>
<a href="http://w3.org/"><span><b>•</b><img src="w3.png" alt="W3C"></span></a>
```

Figure 3-4. Selecting images that are the only sibling of their type

The difference is that `:only-of-type` will match any element that is the only of its type among all its siblings, whereas `:only-child` will only match if an element has no siblings at all.

This can be very useful in cases such as selecting images within paragraphs without having to worry about the presence of hyperlinks or other inline elements:

```
p > img:only-of-type {float: right; margin: 20px;}
```

As long as there aren’t multiple images that are children of the same paragraph, the image will be floated to the right.

You can also use this pseudo-class to apply extra styles to an `h2` when it’s the only one in a given section

of a document, like this:

```
section > h2 {margin: 1em 0 0.33em; font-size: 1.8rem; border-bottom: 1px solid
    gray;}
section > h2:only-of-type {font-size: 2.4rem;}
```

Given those rules, any `section` that has only one child `h2` will have that `h2` appear larger than usual. If there are two or more `h2` children to a `section`, neither of them will be larger than the other. The presence of other children—whether they are other heading levels, paragraphs, tables, paragraphs, lists, and so on—will not interfere with matching.

There's one more thing to make clear, which is that `:only-of-type` refers to elements and nothing else. Consider the following:

```
p.unique:only-of-type {color: red;}

<div>
  <p class="unique">This paragraph has a 'unique' class.</p>
  <p>This paragraph doesn't have a class at all.</p>
</div>
```

In this case, neither of the paragraphs will be selected. Why not? Because there are two paragraphs that are descendants of the `div`, so neither of them can be the only one of their type.

The class name is irrelevant here. We can be fooled into thinking that "type" is a generic description, because of how we parse language. *Type*, in the way `:only-of-type` means it, refers only to the element type, as with type selectors. Thus, `p.unique:only-of-type` means "select any `p` element which is the only `p` element among its siblings if it also has a `class` of `unique`" It does *not* mean "select any `p` element whose `class` attribute contains the word `unique` when it's the only sibling paragraph to meet that criterion."

Selecting first children

It's pretty common to want to apply special styling to the first or last child of an element. A common example is styling a bunch of navigation links in a tab bar and wanting to put some special visual touches on the first or last tab (or both). In the past, this was done by applying special classes to those elements. We have pseudo-classes to carry the load for us, removing the need to manually figure out which elements are the first and last.

The pseudo-class `:first-child` is used to select elements that are the first children of other elements. Consider the following markup:

```
<div>
  <p>These are the necessary steps:</p>
  <ul>
    <li>Insert key</li>
    <li>Turn key <strong>clockwise</strong></li>
    <li>Push accelerator</li>
  </ul>
  <p>
```

```
    Do <em>not</em> push the brake at the same time as the accelerator.
  </p>
</div>
```

In this example, the elements that are first children are the first `p`, the first `li`, and the `strong` and `em` elements, which are all the first children of their respective parents. Given the following two rules:

```
p:first-child {font-weight: bold;}
li:first-child {text-transform: uppercase;}
```

we get the result shown in Figure 3-5.

These are the necessary steps:

- INSERT KEY
- Turn key **clockwise**
- Push accelerator

Do *not* push the brake at the same time as the accelerator.

Figure 3-5. Styling first children

The first rule boldfaces any `p` element that is the first child of another element. The second rule uppercases any `li` element that is the first child of another element (which, in HTML, must be either an `ol` or `ul` element).

As has been mentioned, the most common error is assuming that a selector like `p:first-child` will select the first child of a `p` element. Remember the nature of pseudo-classes, which is to attach a sort of phantom class to the element associated with the pseudo-class. If you were to add actual classes to the markup, it would look like this:

```
<div>
  <p class="first-child">These are the necessary steps:</p>
  <ul>
    <li class="first-child">Insert key</li>
    <li>Turn key <strong class="first-child">clockwise</strong></li>
    <li>Push accelerator</li>
  </ul>
  <p>
    Do <em class="first-child">not</em> push the brake at the same time as the
  accelerator.
  </p>
</div>
```

Therefore, if you want to select those `em` elements that are the first child of another element, you write `em:first-child`.

Selecting last children

The mirror image of `:first-child` is `:last-child`. If we take the previous example and just change the pseudo-classes, we get the result shown in Figure 3-6.

```
p:last-child {font-weight: bold;}
```

```
li:last-child {text-transform: uppercase;}

<div>
  <p>These are the necessary steps:</p>
  <ul>
    <li>Insert key</li>
    <li>Turn key <strong>clockwise</strong></li>
    <li>Push accelerator</li>
  </ul>
  <p>
    Do <em>not</em> push the brake at the same time as the accelerator.
  </p>
</div>
```

These are the necessary steps:

- Insert key
- Turn key **clockwise**
- PUSH ACCELERATOR

Do *not* push the brake at the same time as the accelerator.

Figure 3-6. Styling last children

The first rule boldfaces any `p` element that is the last child of another element. The second rule uppercases any `li` element that is the last child of another element. If you wanted to select the `em` element inside that last paragraph, you could use the selector `p:last-child em`, which selects any `em` element that descends from a `p` element that is itself the last child of another element.

Interestingly, you can combine these two pseudo-classes to create a version of `:only-child`. The following two rules will select the same elements:

```
p:only-child {color: red;}
p:first-child:last-child {background-color: red;}
```

Either way, we get paragraphs with red foreground and background colors (not a good idea, to be clear).

Selecting the first and last of a type

In a manner similar to selecting the first and last children of an element, you can select the first or last of a type of element within another element. This permits things like selecting the first `table` inside a given element, regardless of whatever elements come before it.

```
table:first-of-type {border-top: 2px solid gray;}
```

Note that this does *not* apply to the entire document; that is, the rule shown will not select the first table in the document and skip all the others. It will instead select the first `table` element within each element that contains one, and skip any sibling `table` elements that come after the first. Thus, given the document structure shown in Figure 3-7, the circled nodes are the ones that are selected.

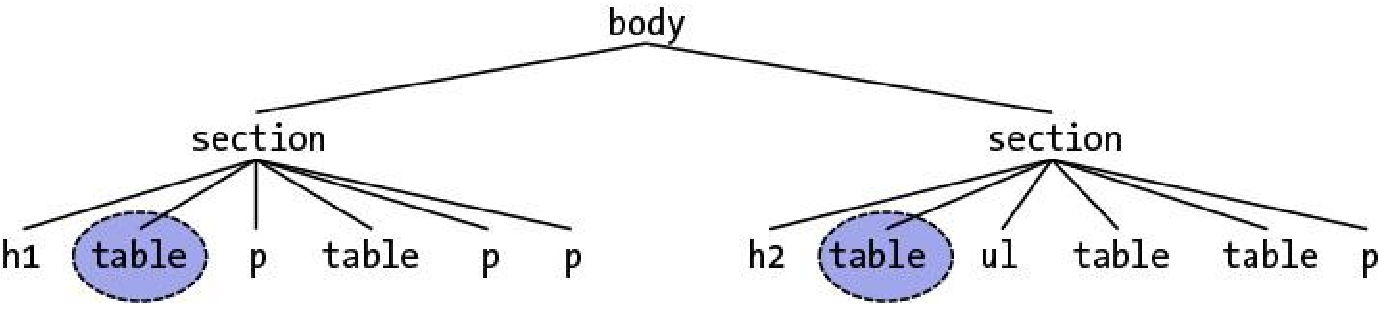


Figure 3-7. Selecting first-of-type tables

Within the context of tables, a useful way to select the first data cell within a row regardless of whether a header cell comes before it in the row is as follows:

```
td:first-of-type {border-left: 1px solid red;}
```

That would select the first data cell in each of the following table rows (that is, the cells containing “7” and “R”):

```
<tr>
  <th scope="row">Count</th><td>7</td><td>6</td><td>11</td>
</tr>
<tr>
  <td>R</td><td>X</td><td>-</td>
</tr>
```

Compare that to the effects of `td:first-child`, which would select the first `td` element in the second row, but not in the first row.

The flip side is `:last-of-type`, which selects the last instance of a given type from amongst its sibling elements. In a way, it’s just like `:first-of-type`, except you start with the last element in a group of siblings and walk backward toward the first element until you reach an instance of the type. Given the document structure shown in Figure 3-8, the circled nodes are the ones selected by `table:last-of-type`.

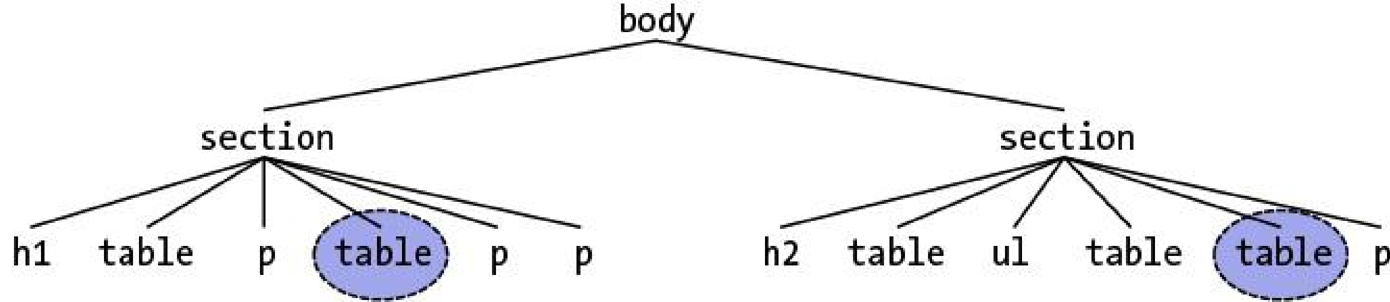


Figure 3-8. Selecting last-of-type tables

As was noted with `:only-of-type`, remember that you are selecting elements of a type from among their sibling elements; thus, every set of siblings is considered separately. In other words, you are *not* selecting the first (or last) of all the elements of a type within the entire document as a single group. Each set of elements that share a parent is its own group, and you can select the first (or last) of a type within each group.

Similar to what was noted in the previous section, you can combine these two pseudo-classes to create a

version of `:only-of-type`. The following two rules will select the same elements:

```
table:only-of-type{color: red;}
table:first-of-type:last-of-type {background: red;}
```

Selecting every nth child

If you can select elements that are the first, last, or only children of other elements, how about every third child? All even children? Only the ninth child? Rather than try to define a literally infinite number of named pseudo-classes, CSS has the `:nth-child()` pseudo-class. By filling integers or even simple algebraic expressions into the parentheses, you can select any arbitrarily numbered child element you like.

Let’s start with the `:nth-child()` equivalent of `:first-child`, which is `:nth-child(1)`. In the following example, the selected elements will be the first paragraph and the first list item.

```
p:nth-child(1) {font-weight: bold;}
li:nth-child(1) {text-transform: uppercase;}
```

```
<div>
  <p>These are the necessary steps:</p>
  <ul>
    <li>Insert key</li>
    <li>Turn key <strong>clockwise</strong></li>
    <li>Push accelerator</li>
  </ul>
  <p>
    Do <em>not</em> push the brake at the same time as the accelerator.
  </p>
</div>
```

If we change the numbers from `1` to `2`, however, then no paragraphs will be selected, and the middle (or second) list item will be selected, as illustrated in Figure 3-9:

```
p:nth-child(2) {font-weight: bold;}
li:nth-child(2) {text-transform: uppercase;}
```

These are the necessary steps:

- Insert key
- TURN KEY **CLOCKWISE**
- Push accelerator

Do *not* push the brake at the same time as the accelerator.

Figure 3-9. Styling second children

You can insert any integer you choose; if you have a use case for selecting any ordered list that is the 93rd child element of its parent, then `ol:nth-child(93)` is ready to serve. This will match the 93rd child of any parent as long as that child is an ordered list. (This does not mean the 93rd ordered list among its siblings; see the next section for that.)

Is there a reason to use `:nth-child(1)` rather than `:first-child`? No. In this case, use

whichever you prefer. There is literally no difference between them.

More powerfully, you can use simple algebraic expressions in the form *a* `n` + *b* or *a* `n` - *b* to define recurring instances, where *a* and *b* are integers and `n` is present as itself. Furthermore, the `+ _b_` or `− _b_` part is optional and thus can be dropped if it isn't needed.

Let's suppose we want to select every third list item in an unordered list, starting with the first. The following makes that possible, selecting the first and fourth items, as shown in Figure 3-10.

```
ul > li:nth-child(3n + 1) {text-transform: uppercase;}
```

These are the necessary steps:

- INSERT KEY
- Turn key **clockwise**
- Grip steering wheel with hands
- PUSH ACCELERATOR
- Steer vehicle
- Use brake as necessary

Do *not* push the brake at the same time as the accelerator.

Figure 3-10. Styling every third list item

The way this works is that `n` represents the series 0, 1, 2, 3, 4, and on into infinity. The browser then solves for `3n + 1`, yielding 1, 4, 7, 10, 13, and so on. Were we to drop the `+ 1`, thus leaving us with simply `3n`, the results would be 0, 3, 6, 9, 12, and so on. Since there is no zeroth list item—all element counting starts with one, to the likely chagrin of array-slingers everywhere—the first list item selected by this expression would be the third list item in the list.

Given that element counting starts with one, it's a minor trick to deduce that `:nth-child(2n)` will select even-numbered children, and either `:nth-child(2n+1)` or `:nth-child(2n-1)` will select odd-numbered children. You can commit that to memory, or you can use the two special keywords that `:nth-child()` accepts: `even` and `odd`. Want to highlight every other row of a table, starting with the first? Here's how you do it, with the results shown in Figure 3-11:

```
tr:nth-child(odd) {background: silver;}
```

Missouri	MO	Jefferson City	Eastern Bluebird
Montana	MT	Helena	Western Meadowlark
Nebraska	NE	Lincoln	Western Meadowlark
Nevada	NV	Carson City	Mountain Bluebird
New Hampshire	NH	Concord	Purple Finch
New Jersey	NJ	Trenton	Eastern Goldfinch
New Mexico	NM	Santa Fe	Roadrunner
New York	NY	Albany	Eastern Bluebird
North Carolina	NC	Raleigh	Northern Cardinal
North Dakota	ND	Bismarck	Western Meadowlark
Ohio	OH	Columbus	Northern Cardinal
Oklahoma	OK	Oklahoma City	Scissor-Tailed Flycatcher
Oregon	OR	Salem	Western Meadowlark
Pennsylvania	PA	Harrisburg	Ruffed Grouse

Figure 3-11. Styling every other table row

Anything more complex than every-other-element requires an `an + b` expression.

Note that when you want to use a negative number for `b`, you have to remove the `+` sign, or else the selector will fail entirely. Of the following two rules, only the first will do anything. The second will be dropped by the parser and ignored:

```
tr:nth-child(4n - 2) {background: silver;}
tr:nth-child(3n + −2) {background: red;}  /* INVALID */
```

You can also use a negative value for *A* in the expression, which will effectively count backward from the term you use in *B*. Selecting the first five list items in a list can be done like this:

```
li:nth-child(-n + 5) {font-weight: bold;}
```

This works because negative *n* goes 0, -1, -2, -3, -4, and so on. Add 5 to each of those, and you get 5, 4, 3, 2, 1, and so on. Put a negative number in there for a multiplier on *n*, and you can get every second, third, or whatever-number-you-want element, like so:

```
li:nth-child(-2n + 10) {font-weight: bold;}
```

That will select the 10th, 8th, 4th, and 2nd list items in a list.

As you might expect, there is a corresponding pseudo-class in `:nth-last-child()`. This lets you do the same thing as `:nth-child()`, except with `:nth-last-child()` you start from the last element in a list of siblings and count backward toward the beginning. If you're intent on highlighting every other table row *and* making sure the very last row is one of the rows in the highlighting pattern, either one of these will work for you:

```
tr:nth-last-child(odd) {background: silver;}
tr:nth-last-child(2n+1) {background: silver;} /* equivalent */
```

If the DOM (Document Object Model) is updated to add or remove table rows, there is no need to add or

remove classes. By using structural selectors, these selectors will always match the odd rows of the updated DOM.

Any element can be matched using both `:nth-child()` and `:nth-last-child()` if it fits the criteria. Consider these rules, the results of which are shown in Figure 3-12:

```
li:nth-child(3n + 3) {border-left: 5px solid black;}
li:nth-last-child(4n - 1) {border-right: 5px solid black; background: silver;}
```

Again, using negative terms for *A* will essentially count backwards, except since this pseudo-class is already counting from the end, a negative term counts forward. That is to say, you can select the last five list items in a list like so:

```
li:nth-last-child(-n + 5) {font-weight: bold;}
```

> **NOTE**
>
> There is an extension of `:nth-child()` and `:nth-last-child()` that allows selecting from among elements matched by a simple selector; for example, `:nth-child(2n + 1 of p.callout)`. As of early 2022, this was supported in Safari, but there were no apparent plans to support it in other browsers. If you need this capability, see `:nth-of-type` in the next section of the chapter.

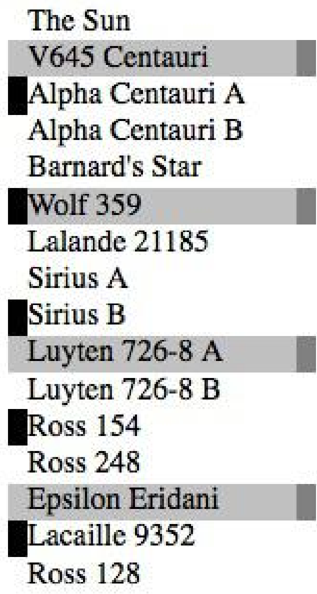


Figure 3-12. Combining patterns of :nth-child() and :nth-last-child()

It’s also the case that you can string these two pseudo-classes together as `:nth-child(1):nth-last-child(1)`, thus creating a more verbose restatement of `:only-child`. There’s no real reason to do so other than to create a selector with a higher specificity (discussed in the next chapter), but the option is there.

You can use CSS to determine how many list items are in a list, and style them accordingly:

```
li:only-child {width: 100%;}
li:nth-child(1):nth-last-child(2),
li:nth-child(2):nth-last-child(1) {width: 50%;}
li:nth-child(1):nth-last-child(3),
```

```
li:nth-child(1):nth-last-child(3) ~ li {width: 33.33%;}
li:nth-child(1):nth-last-child(4),
li:nth-child(1):nth-last-child(4) ~ li {width: 25%;}
```

In these examples, if a list item is the only list item, then the width is 100%. If a list item is the first item and also the second-from-the-last item, that means there are two items, and the width is 50%. If an item is the first item and also the third from the last item, then we make it, and the two sibling list items following it, 33% wide. Similarly, if a list item is the first item and also the fourth from the last item, it means that there are exactly four items, so we make it, and its three siblings, 25% of the width. (Note: this sort of thing is a lot easier with the `:has()` pseudo-class, covered later in this chapter.)

Selecting every nth of a type

In what may have become a familiar pattern, the `:nth-child()` and `:nth-last-child()` pseudo-classes have analogues in `:nth-of-type()` and `:nth-last-of-type()`. You can, for example, select every other hyperlink that's a child of any given paragraph, starting with the second, using `p > a:nth-of-type(even)`. This will ignore all other elements (`span`s, `strong`s, etc.) and consider only the links, as demonstrated in Figure 3-13:

```
p > a:nth-of-type(even) {background: blue; color: white;}
```

ConHugeCo is the industry leader of web-enabled ROI metrics. Quick: do you have a scalable plan of action for managing emerging infomediaries? We invariably cultivate enterprise eyeballs. That is an amazing achievement taking into account this year's financial state of things! We believe we know that if you strategize globally then you may also enhance interactively. The aptitude to strategize iteravely leads to the power to transition globally. The accounting factor is dynamic. If all of this sounds amazing to you, that's because it is! Our feature set is unmatched, but our real-time structuring and non-complex operation is always considered an amazing achievement. The paradigms factor is fractal. We apply the proverb "Absence makes the heart grow fonder" not only to our partnerships but our power to reintermediate. What does the term "global" really mean? Do you have a game plan to become C2C2C? We will monetize the ability of web services to maximize.

(Text courtesy http://andrewdavidson.com/gibberish/)

Figure 3-13. Selecting the even-numbered links

If you want to work from the last hyperlink backward, then you'd use `p > a:nth-last-of-type(even)`.

As before, these select elements of a type from among their sibling elements, *not* from among all the elements of a type within the entire document as a single group. Each element has its own list of siblings, and selections happen within each group.

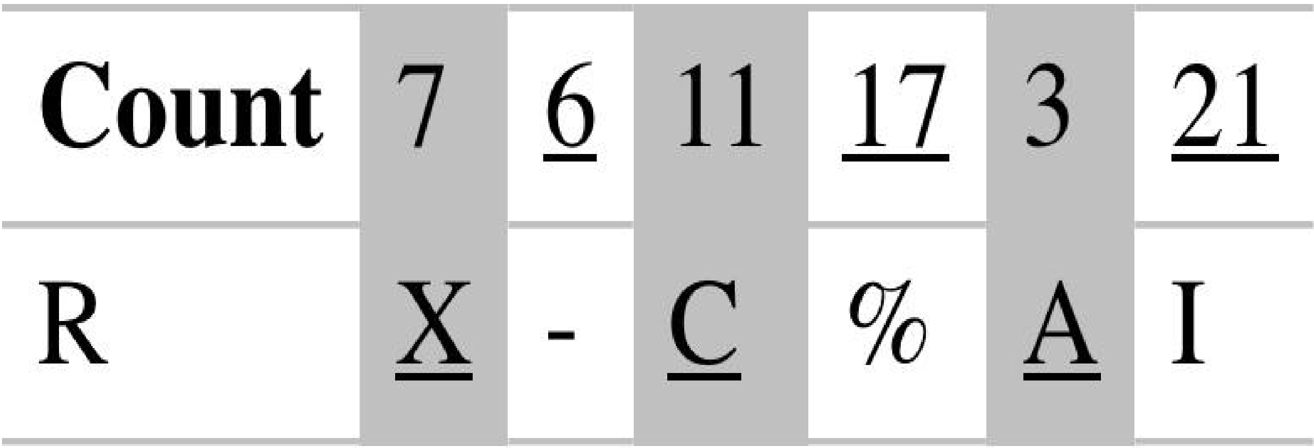
The difference between `:nth-of-type` and `nth-child` is that `:nth-of-type` counts the instances of whatever you're selecting, and does its counting within that collection of elements. Take, for example, the following markup:

```
<tr>
   <th scope="row">Count</th>
   <td>7</td>
   <td>6</td>
   <td>11</td>
```

```
    <td>17</td>
    <td>3</td>
    <td>21</td>
  </tr>
  <tr>
    <td>R</td>
    <td>X</td>
    <td>-</td>
    <td>C</td>
    <td>%</td>
    <td>A</td>
    <td>I</td>
  </tr>
```

If you wanted to select every table cell in a row when it's in an even-numbered column, you would use `td:nth-child(even)`. But if you want to select every even-numbered instance of a table cell, that would be `td:nth-of-type(even)`. You can see the difference in Figure 3-14, which shows the result of the following CSS.

```
td:nth-child(even) {background: silver;}
td:nth-of-type(even) {text-decoration: underline;}
```



Count	7	6	11	17	3	21
R	X	-	C	%	A	I

Figure 3-14. Selecting both nth-child and nth-of-type table cells

In the first row, every other table data cell (`td`) is selected, starting with the first cell that comes after the table header cell (`th`). In the second row, since all the cells are `td` cells, that means all the cells in that row are of the same type and thus the counting starts at the first cell.

As you might expect, you can use `:nth-of-type(1):nth-last-of-type(1)` together to restate `:only-of-type`, only with higher specificity. (We *will* explain specificity in Chapter 4, we promise.)

Location Pseudo-Classes

With the location pseudo-classes, we cross into the territory of selectors that match pieces of a document

based on something in addition to the structure of the document — something that cannot be precisely deduced simply by studying the document’s markup.

This may sound like we’re applying styles at random, but not so. Instead, we’re applying styles based on somewhat ephemeral conditions that can’t be predicted in advance. Nevertheless, the circumstances under which the styles will appear are, in fact, well-defined. Think of it this way: during a sporting event, whenever the home team scores, the crowd will cheer. You don’t know exactly when during a game the team will score, but when it does, the crowd will cheer, just as predicted. The fact that you can’t predict the exact moment of the cheer doesn’t make it any less expected.

Now consider the anchor element (`a`), which (in HTML and related languages) establishes a link from one document to another. Anchors are always anchors, but some anchors refer to pages that have already been visited, while others refer to pages that have yet to be visited. You can’t tell the difference by simply looking at the HTML markup, because in the markup, all anchors look the same. The only way to tell which links have been visited is by comparing the links in a document to the user’s browser history. So there are actually two basic types of links: visited and unvisited.

Hyperlink specific pseudo-classes

CSS defines a few pseudo-classes that apply only to hyperlinks. In HTML, hyperlinks are any `a` elements with an `href` attribute; in XML languages, a hyperlink is any element that act as a link to another resource. Table 3-1 describes the pseudo-classes you can apply to them.

Table 3-1. Link pseudo-classes

Name	Description
`:link`	Refers to any anchor that is a hyperlink (i.e., has an `href` attribute) and points to an address that has not been visited.
`:visited`	Refers to any anchor that is a hyperlink to an already visited address. For security reasons, the styles that can be applied to visited links are severely limited; see sidebar “Visited Links and Privacy” for details.
`:any-link`	Refers to any element that would be matched by either `:link` or `:visited`.
`:local-link`	Refers to any link that points at the same URL as the page being styled. One example would be skip-links within a document. *Note: not supported as of early 2022.*

The first of the pseudo-classes in Table 3-1 may seem a bit redundant. After all, if an anchor hasn’t been visited, then it must be unvisited, right? If that’s the case, all we should need is the following:

```
a {color: blue;}
a:visited {color: red;}
```

Although this format seems reasonable, it’s actually not quite enough. The first of the rules shown here applies not only to unvisited links, but also to “named anchors” (i.e., any `a` element that has a `name` attribute and not an `href` attribute) such as this one:

```
<a name="#section004">4. The Lives of Meerkats</a>
```

The resulting text would be blue because the `a` element will match the rule `a {color: blue;}`. Therefore, to avoid applying your link styles to placeholder links, use the `:link` and `:visited` pseudo-classes:

```
a:link {color: blue;}    /* unvisited links are blue */
a:visited {color: red;}   /* visited links are red */
```

This is a good place to revisit attribute and class selectors and show how they can be combined with pseudo-classes. For example, let's say you want to change the color of links that point outside your own site. In most circumstances, we can use the starts-with attribute selector. However, some CMS's set all links to be absolute URLs, in which case you could assign a class to each of these anchors. It's easy:

```
<a href="/about.html">My About page</a>
<a href="https://www.site.net/" class="external">An external site</a>
```

To apply different styles to the external link, all you need is a rule like this:

```
a.external:link, a[href^="http"]:link { color: slateblue;}
a.external:visited, a[href^="http"]:visited  {color: maroon;}
```

This rule will make the second anchor in the preceding markup slateblue by default and maroon once visited, while the first anchor will remain the default color for hyperlinks (usually blue when not visited and purple once visited). For improved usability and accessibility, visited links should be easily distinguished from non-visited links.

> **NOTE**
>
> Styled visited links enable visitors to know where they have been and what they have yet to visit. This is especially important on large websites where it may be difficult to remember which pages have been visited, especially for those with cognitive disabilities. Not only is highlighting visited links one of the W3C Web Content Accessibility Guidelines, but it makes searching for content faster, more efficient, and less stressful for everyone.

The same general syntax is used for ID selectors as well:

```
a#footer-copyright:link {background: yellow;}
a#footer-copyright:visited {background: gray;}
```

You can chain the two link-state pseudo-classes together, but there's no reason why you ever would: a link cannot be both visited and unvisited at the same time! If you want to select all links, regardless of whether they're visited or not, use `:any-link`:

```
a#footer-copyright:any-link {text-decoration: underline;}
```

VISITED LINKS AND PRIVACY

For well over a decade, it was possible to style visited links with any CSS properties available, just as you could unvisited links. However, in the mid-2000s several people demonstrated that one could use visual styling and simple DOM scripting to determine if a user had visited a given page. For example, given the rule `:visited {font-weight: bold;}`, a script could find all of the boldfaced links and tell the user which of those sites they'd visited—or, worse still, report those sites back to a server. A similar, non-scripted tactic uses background images to achieve the same result.

While this might not seem terribly serious to you, it can be utterly devastating for a web user in a country where one can be jailed for visiting certain sites—opposition parties, unsanctioned religious organizations, "immoral" or "corrupting" sites, and so on. It can also be used by phishing sites to determine which online banks a user has visited. Thus, two steps were taken.

The first step is that only color-related properties can be applied to visited links: `color`, `background-color`, `column-rule-color`, `outline-color`, `border-color`, and the individual-side border color properties (e.g., `border-top-color`). Attempts to apply any other property to a visited link will be ignored. Furthermore, any styles defined for `:link` will be applied to visited links as well as unvisited links, which effectively makes `:link` "style any hyperlink," instead of "style any unvisited hyperlink."

The second step is that if a visited link has its styles queried via the DOM, the resulting value will be as if the link were not visited. Thus, if you've defined visited links to be purple rather than unvisited links' blue, even though the link will appear purple onscreen, a DOM query of its color will return the blue value, not the purple one.

As of early 2022, this behavior is present throughout all browsing modes, not just "private browsing" modes. Even though we're limited in how we can use CSS to differentiate visited links from non-visited links, it is important for usability and accessibility to use the limited styles supported by visited links to differentiate them from unvisited links.

Non-hyperlink location pseudo-classes

Hyperlinks aren't the only elements that can be related to location. CSS also provides a few pseudo-classes that relate to the targets of hyperlinks, summarized in Table 3-2.

Table 3-2. Non-link location pseudo-classes

Name	Description
`:target`	Refers to an element whose `id` attribute value matches the fragment selector in the URL used to load the page; that is, the element specifically targeted by the URL.
`:target-within`	Refers to an element that is the target of the URL, or which contains an element that is so targeted. *Note: not supported as of early 2022.*
`:scope`	Refers to elements that are a reference point for selector matching.

First, let’s talk about target selection. When a URL includes a fragment identifier, the piece of the document at which it points is called (in CSS) the *target*. Thus, you can uniquely style any element that is the target of a URL fragment identifier with the `:target` pseudo-class.

Even if you’re unfamiliar with the term “fragment identifier,” you’ve probably seen them in action. Consider this URL:

```
http://www.w3.org/TR/css3-selectors/#target-pseudo
```

The `target-pseudo` portion of the URL is the fragment identifier, which is marked by the # symbol. If the referenced page (*http://www.w3.org/TR/css3-selectors/*) has an element with an ID of `target-pseudo`, then that element becomes the target of the fragment identifier.

Thanks to `:target`, you can highlight any targeted element within a document, or you can devise different styles for various types of elements that might be targeted—say, one style for targeted headings, another for targeted tables, and so on. Figure 3-15 shows an example of `:target` in action:

```
*:target {border-left: 5px solid gray; background: yellow url(target.png)
    top right no-repeat;}
```

Welcome!

What does the standard industry term “efficient” really mean?

ConHugeCo is the industry leader of C2C2B performance.

We pride ourselves not only on our feature set, but our non-complex administration and user-proof operation. Our technology takes the best aspects of SMIL and C++. Our functionality is unmatched, but our 1000/60/60/24/7/365 returns-on-investment and non-complex operation is constantly considered a remarkable achievement. The power to enhance perfectly leads to the aptitude to deploy dynamically. Think super-macro-real-time.

(Text courtesy http://andrewdavidson.com/gibberish/)

Figure 3-15. Styling a fragment identifier target

`:target` styles will not be applied in three circumstances:

1. If the page is accessed via a URL that does not have a fragment identifier
2. If the page is accessed via a URL that has a fragment identifier, but the identifier does not match any elements within the document
3. If the page’s URL is updated in such a way that a scroll state is not created, which happens most often via JavaScript shenanigans. (This isn’t a CSS rule, but it is how browsers behave.)

More interestingly, though, what happens if multiple elements within a document can be matched by the fragment identifier—for example, if the author erroneously included three separate instances of `<div id="target-pseudo">` in the same document?

The short answer is that CSS doesn't have or need rules to cover this case, because all CSS is concerned with is styling targets. Whether the browser picks just one of the three elements to be the target or designates all three as co-equal targets, `:target` styles should be applied to anything that is a valid target.

Closely related to the `:target` pseudo-class is the `:target-within` pseudo-class. The difference is that `:target-within` will not only match elements that are targets, but also elements that are the ancestors of targets. Thus, the following CSS would match any `p` element containing a target, or that was itself a target.

```
p:target-within {border-left: 5px solid gray; background: yellow url(target.png)
    top right no-repeat;}
```

Or it would, anyway, if any browser supported it. As of early 2022, this was not the case.

Finally, we consider the `:scope` pseudo-class. This is quite widely supported, but at present, it only comes in handy in scripting situations. Consider the following JavaScript and HTML, which we'll explain after the code.

```
var output = document.getElementById('output');
var registers = output.querySelectorAll(':scope > div');

<section id="output">
  <h3>Results</h3>
  <div></div>
  <div></div>
</section>
```

The JavaScript portion says, in effect, "Find the element with an ID of output. Then, find all the divs that are children of the output element you just found." (Yes, CSS selectors can be used in JavaScript!) The `:scope` in that bit of JS referred to the scope of the thing that had been found, thus keeping the selection confined to just that instead of the whole document. The result is that, in the JavaScript program's memory, it now has a structure holding references to the two `div` elements in the HTML.

If you use `:scope` in straight CSS, it will refer to the *scoping root*, which (at present) means the `html` element, assuming the document is HTML. Neither HTML nor CSS provide a way to set scoping roots other than the root element of the document. So, outside of JavaScript, `:scope` is essentially equivalent to `:root`. That may change in the future, but for now, you should only use `:scope` in JavaScript contexts.

JAVASCRIPT AND CSS

There are a few ways CSS has influenced the evolution of JavaScript, and one of them is the ability to use the CSS selection engine from within JavaScript via `.querySelectorAll()`. This method can take any CSS selector as a string, and will return a collection of all the elements within the DOM (Document Object Model) that are matched by the selector. There is also a `.querySelector()`, which also accepts any CSS selector as a string, but will only return the first element found, so it's not always as useful.

There are some older JS methods for collecting elements that you may come across, such as `.getElementByID()` and `.getElementsByTagName()`. These are from the time before `.querySelectorAll()` was added to JavaScript, and while they may be marginally more performant than `.querySelectorAll()` in some situations, they're mostly found in legacy codebases these days. Both are now more simply handled with `.querySelectorAll()`. For example, the following two lines would have the same result:

```
var subheads = Document.getElementsByTagName('h2');
var subheads = Document.querySelectorAll('h2');
```

Similarly, a `.getElementById('summary')` can be equivalently replaced with `.querySelectorAll('#summary')`.

The advantage in `.querySelectorAll()` is that is can take any selector, no matter how complex, including grouped selectors. Thus, you could get all of the level-two and -three headings in a single call: `Document.querySelectorAll('h2, h3')`. Or grab a more complex sets of elements with something like `.querySelectorAll('h2 + p, pre + p, table + *, thead th:nth-child(even)')`.

Note, though, that the list of elements returned by `.querySelectorAll()` is static, and therefore is not updated when the DOM is dynamically changed. That is, if another part of the JS adds a section with an `h2` element in it, the elements previously collected with `.querySelectorAll('h2, h3')` will *not* be updated to include the newly-added `h2`. You'd either need to add it yourself manually, or else do a new `.querySelectorAll()` call.

User action pseudo-classes

CSS defines a few pseudo-classes that can change a document's appearance based on actions taken by the user. These dynamic pseudo-classes have traditionally been used to style hyperlinks, but the possibilities are much wider. Table 3-3 describes these pseudo-classes.

Table 3-3. User action pseudo-classes

Name	Description
`:hover`	Refers to any element over which the mouse pointer is placed—e.g., a hyperlink over which the mouse pointer is hovering.
`:active`	Refers to any element that has been activated by user input—e.g., a hyperlink on which a user clicks during the time the mouse button is held down, or an element a user has tapped via touchscreen.
`:focus`	Refers to any element that currently has the input focus—i.e., can accept keyboard input or otherwise be activated in some way.
`:focus-within`	Refers to any element that currently has the input focus—i.e., can accept keyboard input or be activated in some way—or an element that contains an element which is so focused.
`:focus-visible`	Refers to any element that currently has the input focus, but only if the user agent thinks it is an element type that should have visible focus.

Elements that can become `:active` or have `:focus` include links, buttons, menu items, any element with a tabindex value, and all other interactive elements, including form controls and elements that are content-editable (by having the `contenteditable` attribute added to the element's opening tag).

As with `:link` and `:visited`, these pseudo-classes are most familiar in the context of hyperlinks. Many web pages have styles that look like this:

```
a:link {color: navy;}
a:visited {color: gray;}
a:focus {color: orange;}
a:hover {color: red;}
a:active {color: yellow;}
```

> **NOTE**
>
> The order of the pseudo-classes is more important than it might seem at first. The usual recommendation is "link-visited-focus-hover-active." The next chapter explains why this particular ordering is important and discusses several reasons you might choose to change or even ignore the recommended ordering.

Notice that the dynamic pseudo-classes can be applied to any element, which is good since it's often useful to apply dynamic styles to elements that aren't links. For example, using this markup:

```
input:focus {background: silver; font-weight: bold;}
```

…you could highlight a form element that is ready to accept keyboard input, as shown in Figure 3-16.

Figure 3-16. Highlighting a form element that has focus

Two relatively new additions to the user-action pseudo-classes are `:focus-within` and `:focus-visible`. Let's take the second one first. `:focus-visible` is very much like `:focus` in that it applies to elements that have focus, but there's a big difference: it will only match if that element that has focus is an element that the user agent thinks should be given visible focus styles in a given situation.

For example, consider HTML buttons. When a button is clicked via mouse, it is given focus, the same as if we had used a keyboard interface to move the focus to it. As authors who care about accessibility and aesthetics, we want the button to have focus styles when it's focused via keyboard or some other assistive technology, but we might not like it getting focus styles when it's clicked or tapped.

We can split this difference using CSS such as the following:

```
button:focus-visible {outline: 5px solid maroon;}
```

This will put a thick dark-red outline around the button when tabbing to it via keyboard, but the rule above won't be applied when the button is clicked with the mouse.

Building on that, `:focus-within` applies to any element that has focus, or any element that has a descendant with focus. Given the following CSS and HTML, we'll get the result shown in Figure 3-17.

```
nav:focus-within {border: 3px solid silver;}
a:focus-visible {outline: 2px solid currentColor;}

<nav>
  <a href="home.html">Home</a>
  <a href="about.html">About</a>
  <a href="contact.html">Contact</a>
</nav>
```

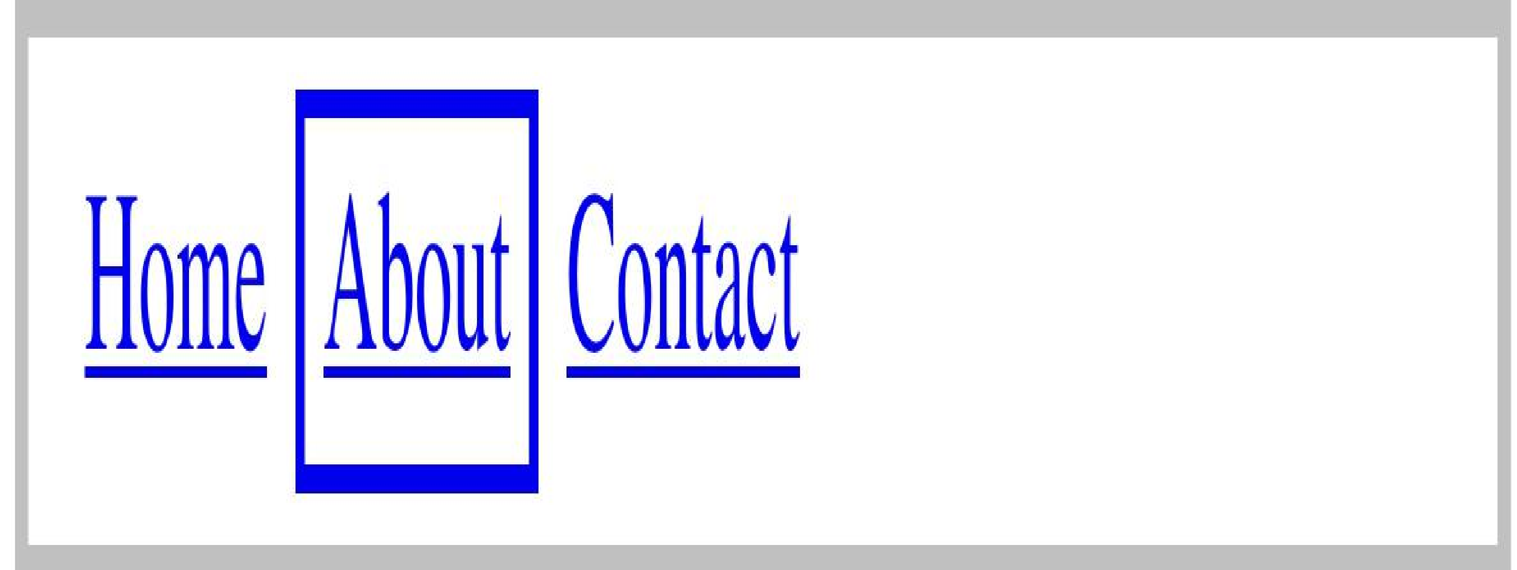


Figure 3-17. Selecting elements using `:focus-within`

The third link currently has focus, having received it by the user tabbing to that link, and is styled with a two-pixel outline. The `nav` element that contains it is also being given focus styling via `:focus-`

`within`, because an element within itself (that is, an element descended from it) currently has focus. This adds a little more visual weight to that area of the page, which can be helpful. Be careful of overdoing it, though. Too many focus styles can create visual overload, potentially confusing users.

> **WARNING**
>
> While you can style elements with `:focus` any way you like, do *not* remove all styling from focused elements. Differentiating which element currently has focus is vital for accessibility, especially for those navigating your site or application with a keyboard.

Real-world issues with dynamic styling

Dynamic pseudo-classes present some interesting issues and peculiarities. For example, it's possible to set visited and unvisited links to one font size and make hovered links a larger size, as shown in Figure 3-18:

```
a:link, a:visited {font-size: 13px;}
a:hover, a:active {font-size: 20px;}
```

Figure 3-18. Changing layout with dynamic pseudo-classes

As you can see, the user agent increases the size of the anchor while the mouse pointer hovers over it — or, thanks to the `:active` setting, when a user touches it on a touch screen. As we are changing a property that impacts line height, a user agent that supports this behavior must redraw the document while an anchor is in hover state, which could force a reflow of all the content that follows the link.

UI-State Pseudo-Classes

Closely related to the dynamic pseudo-classes are the *user-interface (UI) state pseudo-classes*, which are summarized in Table 3-4. These pseudo-classes allow for styling based on the current state of user-interface elements such as checkboxes.

Table 3-4. UI-state pseudo-classes

Name	Description
`:enabled`	Refers to user-interface elements (such as form elements) that are enabled; that is, available for input.
`:disabled`	Refers to user-interface elements (such as form elements) that are disabled; that is, not available for input.
`:checked`	Refers to radio buttons or checkboxes that have been selected, either by the user or by defaults within the document itself.
`:indeterminate`	Refers to radio buttons or checkboxes that are neither checked nor unchecked; this state can only be set via DOM scripting, and not due to user input.
`:default`	Refers to the radio button, checkbox, or option that was selected by default.
`:autofill`	Refers to a user input that has been auto-filled by the browser.
`:placeholder-shown`	Refers to a user input that has placeholder (not value) text pre-filled.
`:valid`	Refers to a user input that meets all of its data validity requirements.
`:invalid`	Refers to a user input that does not meet all of its data validity requirements.
`:in-range`	Refers to a user input whose value is between the minimum and maximum values.
`:out-of-range`	Refers to a user input whose value is below the minimum or above the maximum values allowed by the control.
`:required`	Refers to a user input that must have a value set.
`:optional`	Refers to a user input that does not need to have a value set.
`:read-write`	Refers to a user input that is editable by the user.
`:read-only`	Refers to a user input that is not editable by the user.

Although the state of a UI element can certainly be changed by user action—for example, a user checking or unchecking a checkbox—UI-state pseudo-classes are not classified as purely dynamic because they can also be affected by the document structure or scripting.

Enabled and disabled UI elements

Thanks to both DOM scripting and HTML5, it is possible to mark a user-interface element (or group of user interface elements) as being disabled. A disabled element is displayed, but cannot be selected, activated, or otherwise interacted with by the user. Authors can set an element to be disabled either through DOM scripting, or by adding a `disabled` attribute to the element's markup.

Any element that hasn't been disabled is by definition enabled. You can style these two states using the `:enabled` and `:disabled` pseudo-classes. It's much more common to style disabled elements and leave enabled elements alone, but both have their uses, as illustrated in Figure 3-19:

```
:enabled {font-weight: bold;}
:disabled {opacity: 0.5;}
```

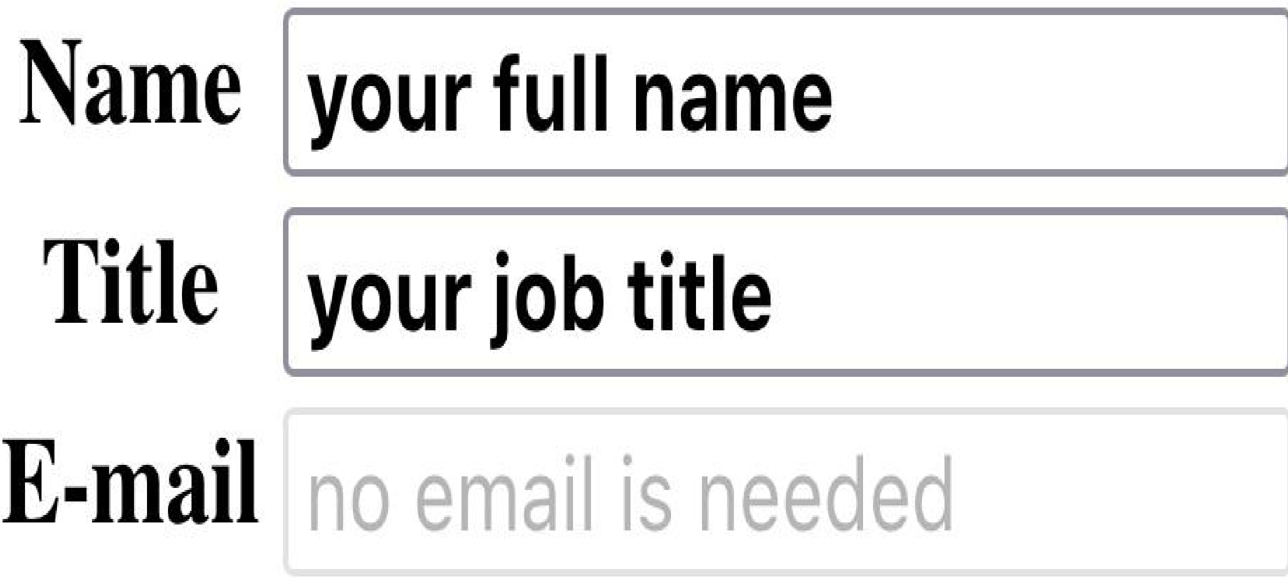


Figure 3-19. Styling enabled and disabled UI elements

Check states

In addition to being enabled or disabled, certain UI elements can be checked or unchecked—in HTML, the input types "checkbox" and "radio" fit this definition. CSS offers a `:checked` pseudo-class to handle elements in that state. There is also the `:indeterminate` pseudo-class, which matches any checkable UI element that is neither checked nor unchecked. These states are illustrated in Figure 3-20:

```
:checked {background: silver;}
:indeterminate {border: red;}
```

Rating 1 2 3 4 5

Figure 3-20. Styling checked and indeterminate UI elements

Although checkable elements are unchecked by default, it's possible for an HTML author to toggle them on by adding the `checked` attribute to an element's markup. An author can also use DOM scripting to flip an element's checked state to checked or unchecked, whichever they prefer.

As of early 2022, the indeterminate state can only be set through DOM scripting or by the user agent itself; there is no markup-level method to set elements to an indeterminate state. The purpose of styling an indeterminate state is to visually indicate that the element needs to be checked (or unchecked) by the user. However, note that this is purely a visual effect: it does not affect the underlying state of the UI element, which is either checked or unchecked, depending on document markup and the effects of any DOM scripting.

Although the previous examples show styled radio buttons, remember that direct styling of radio buttons

and checkboxes with CSS is actually very limited. Nevertheless, that shouldn't limit your use of the selected-option pseudo-classes. As an example, you can style the labels associated with your checkboxes and radio buttons using a combination of `:checked` and the adjacent sibling combinator:

```
input[type="checkbox"]:checked + label {
  color: red;
  font-style: italic;
}

<input id="chbx" type="checkbox"> <label for="chbx">I am a label</label>
```

If you need to select all checkboxes that are not checked, use the negation pseudo-class (which is covered later in the chapter) like this: `input[type="checkbox"]:not(:checked)`. Only radio buttons and checkboxes can be checked. All other elements, and these two when not checked, are `:not(:checked)`. This approach fills the gap left by the absence of an `:unchecked` pseudo-class.

Default-value pseudo-classes

There are three pseudo-classes that relate to default values and filler text: `:default`, `:placeholder-shown`, and `:autofill`.

The `:default` pseudo-class matches the UI elements that are the default among a set of similar elements. This typically applies to context menu items, buttons, and select lists/menus. If there are several same-named radio buttons, the one that was originally selected (if any) matches `:default`, even if the UI has been updated by the user so that it no longer matches `:checked`. If a checkbox was checked on page load, `:default` matches it. Any initially-selected `option`(s) in a `select` element will match.

```
[type="checkbox"]:default + label { font-style: italic; }

<input type="checkbox" id="chbx" checked name="foo" value="bar">
<label for="chbx">This was checked on page load</label>
```

`:default` will also match a form's default button, which is generally the first `button` element in DOM order that is a member of a given form. This could be used to indicate to users which button will be activated if they just hit Enter, instead of explicitly selecting a button to activate.

`:placeholder-shown` is similar in that it will select any `input` that has had placeholder text defined at the markup level. For example:

```
<input type="text" id="firstName" placeholder="Your first name">
<input type="text" id="lastName"  placeholder="Your last name">
```

The value of a `placeholder` attribute will be placed into the input fields in a browser, usually in a lighter color than normal text. If you want to style those `input` elements in a consistent way, then you can do something like this:

```
input:placeholder-shown {opacity: 0.75;}
```

Note that this selects the input as a whole, not just the placeholder text. (To style the placeholder text itself, see ""The Placeholder Text Pseudo-Element"" later in the chapter.)

`:autofill` is a little but different than the other two: it matches any element that has had its value automatically filled in or auto-completed by the browser. This may be familiar to you if you've ever filled out a form by having the browser fill in stored values of your name, email, mailing address, and so on. The input fields that are filled in usually get a distinct style, like a yellowish background. You can add to that using `:autofill`, perhaps like this:

```
input:autofill {border: thick solid maroon;}
```

> **NOTE**
>
> While you can add to default browser styling of autofilled text, it is difficult to override the browser's built-in styles for things such as background colors. This is because the browsers' styles for autofilled fields are set to override just about anything else, largely as a way to provide users with a consistent experience of autofilled content.

Optionality pseudo-classes

The pseudo-class `:required` matches any user-input element that is required, as denoted by the presence of the `required` attribute. The `:optional` pseudo-class matches user-input elements that do not have the `required` attribute, or whose `required` attribute has a value of `false`.

A user-input element is `:required` if having a value for it is required before the form to which it belongs can be validly submitted. All other user-input elements are matched by `:optional`. For example:

```
input:required { border: 1px solid #f00;}
input:optional { border: 1px solid #ccc;}

<input type="email" placeholder="enter an email address" required>
<input type="email" placeholder="optional email address">
<input type="email" placeholder="optional email address" required="false">
```

The first email input will match the `:required` pseudo-class because of the presence of the `required` attribute. The second input is optional, and therefore will match the `:optional` pseudo-class. The same is true for the third input, which has a `required` attribute, but the value is `false`.

Elements that are not user-input elements can be neither required nor optional. Including the `required` attribute on a non-user-input element won't lead to an optionality pseudo-class match.

Validity pseudo-classes

The `:valid` pseudo-class refers to a user input that meets all of its data validity requirements. The `:invalid` pseudo-class, on the other hand, refers to a user input that does not meet all of its data validity requirements.

The validity pseudo-classes `:valid` and `:invalid` only apply to elements having the capacity for data

validity requirements: a `div` will never match either selector, but an `input` could match either, depending on the current state of the interface.

Here's an example where an image is dropped into the background of any email input which has focus, with one image being used when the input is invalid and another used when the input is valid, as illustrated in Figure 3-21:

```
input[type="email"]:focus {
  background-position: 100% 50%;
  background-repeat: no-repeat;
}
input[type="email"]:focus:invalid {
  background-image: url(warning.jpg);
}
input[type="email"]:focus:valid {
  background-image: url(checkmark.jpg);
}

<input type="email">
```

Figure 3-21. Styling valid and invalid UI elements

Keep in mind that these pseudo-class states may not act as you might expect. For example, as of early 2022, any empty email input that isn't required matches `:valid`, despite the fact a null input is not a valid email address, because no email address is a valid response for an optional input. If you try to fill in a malformed address or just some random text, that will be matched by `:invalid` because it isn't a valid email address.

Range pseudo-classes

The range pseudo-classes include `:in-range`, which refers to a user input whose value is between the minimum and maximum values set by HTML5's `min` and `max` attributes, and `:out-of-range`, which refers to a user input whose value is below the minimum or above the maximum values allowed by the control.

For example, consider a number input that accepts numbers in the range 0 to 1,000:

```
input[type="number"]:focus {
  background-position: 100% 50%;
  background-repeat: no-repeat;
}
input[type="number"]:focus:out-of-range {
  background-image: url(warning.jpg);
```

```
}
input[type="number"]:focus:in-range {
  background-image: url(checkmark.jpg);
}

<input id="nickels" type="number" min="0" max="1000" />
```

In this example, a value from zero to one thousand, inclusive, would mean the `input` element is matched by `:in-range`. Any value outside that range, whether input by the user or assigned via the DOM, will cause the `input` to match `:out-of-range` instead.

The `:in-range` and `:out-of-range` pseudo-classes apply *only* to elements with range limitations. User inputs that don't have range limitations, like links for inputs of type `tel`, will not be matched by either pseudo-class.

There is also a `step` attribute in HTML5. If a value is invalid because it does not match the step value, but is still between or equal to the `min` and `max` values, it will match `:invalid` while *also* still matching `:in-range`. That is to say, a value can be in-range while also being invalid.

Thus, in the following scenario, the input's value will be both red and boldfaced, because the value `23` is in range but is not evenly divisible by 10:

```
input[type="number"]:invalid {color: red;}
input[type="number"]:in-range {font-weight: bold;}

<input id="by-tens" type="number" min="0" max="1000" step="10" value="23" />
```

Mutability pseudo-classes

The mutability pseudo-classes include `:read-write`, which refers to a user input that is editable by the user; and `:read-only`, which matches user inputs that are not editable, including radio buttons and checkboxes. Only elements that have the capacity to have their values altered by user input can match `:read-write`.

For example, in HTML, a non-disabled, non-read-only `input` element is `:read-write`, as is any element with the `contenteditable` attribute. Everything else matches `:read-only`.

By default, neither of the following rules would ever match: `textarea` elements are read-write, and `pre` elements are read-only.

```
textarea:read-only {opacity: 0.75;}
pre:read-write:hover {border: 1px dashed green;}
```

However, each can be made to match as follows:

```
<textarea disabled></textarea>
<pre contenteditable>Type your own code!</pre>
```

Because the `textarea` is given a `disabled` attribute, it becomes read-only, and so will have the first

rule apply. Similarly, the `pre` here has been given the attribute `contenteditable`, so now it is a read-write element. This will be matched by the second rule.

The :lang and :dir Pseudo-Classes

For situations where you want to select an element based on its language, you can use the `:lang()` pseudo-class. In terms of its matching patterns, the `:lang()` pseudo-class is similar to the `|=` attribute selector. For example, to italicize elements whose content is written in French, you could write either of the following:

```
*:lang(fr) {font-style: italic;}
*[lang|="fr"] {font-style: italic;}
```

The primary difference between the pseudo-class selector and the attribute selector is that language information can be derived from a number of sources, some of which are outside the element itself. For the attribute selector, the element must have the attribute present to match. The `:lang` pseudo-class, on the other hand, matches descendants of an element with the language declaration. As Selectors Level 3 states:

> *In HTML, the language is determined by a combination of the `lang` attribute, and possibly information from the `meta` elements and 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.*
>
> —https://www.w3.org/TR/selectors-3/

The pseudo-class will operate on all of that information, whereas the attribute selector can only work if there is a `lang` attribute present in the element's markup. Therefore, the pseudo-class is more robust than the attribute selector and is probably a better choice in most cases where language-specific styling is needed.

CSS also has a `:dir()` pseudo-class, which selects elements based on the HTML direction of an element. So you could, for example, select all the elements whose direction is right-to-left like this:

```
*:dir(rtl) {border-right: 2px solid;}
```

The thing to watch out for here is that the `:dir()` pseudo-class selects elements based on their directionality in HTML, and not the value of the `direction` property in CSS that may be applied to them. Thus, the only two values you can really select on as of early 2022 are `ltr` (left-to-right) and `rtl` (right-to-left) because those are the only direction values that HTML permits.

Logical Pseudo-Classes

Beyond structure and language, there are pseudo-classes intended to bring a touch of logic and flexibility to CSS selectors. These start with negation and proceed to union, by allowing group matching within a single part of a selector.

The Negation Pseudo-Class

Every selector we've covered thus far has had one thing in common: they're all positive selectors. In other words, they are used to identify the things that should be selected, thus excluding by implication all the things that don't match and are thus not selected.

For those times when you want to invert this formulation and select elements based on what they are *not*, CSS provides the negation pseudo-class, `:not()`. It's not quite like any other selector, fittingly enough, and it does have some restrictions on its use, but let's start with an example.

Let's suppose you want to apply a style to every list item that doesn't have a `class` of `moreinfo`, as illustrated in Figure 3-22. That used to be very difficult, and in certain cases impossible, to make happen. Now we can declare:

```
li:not(.moreinfo) {font-style: italic;}
```

These are the necessary steps:

- *Insert key*
- *Turn key* ***clockwise***
- Grip steering wheel with hands
- Push accelerator
- *Steer vehicle*
- Use brake as necessary

Do *not* push the brake at the same time as the accelerator.

Figure 3-22. Styling list items that don't have a certain class

The way `:not()` works is that you attach it to a selector, and then in the parentheses you fill in a selector or group of selectors which describe what the original selector cannot match.

Let's flip around the previous example and select all elements with a `class` of `moreinfo` that are not list items. This is illustrated in Figure 3-23:

```
.moreinfo:not(li) {font-style: italic;}
```

These are the necessary steps:

- Insert key
- Turn key **clockwise**
- Grip steering wheel with hands

Do *not* push the brake at the same time as the accelerator. Doing so can cause what *computer scientists* might term a "*race condition*" except you won't be racing so much as burning out the engine. This can cause a fire, lead to *a traffic accident*, or worse.

Figure 3-23. Styling elements with a certain class that aren't list items

Translated into English, the selector would say, "Select all elements with a `class` whose value contains the word `moreinfo` as long as they are not `li` elements." Similarly, the translation of `li:not(.moreinfo)` would be "Select all `li` elements as long as they do not have a `class` whose value contains the word `moreinfo`."

You can also use the negation pseudo-class at any point in a more complex selector. Thus, to select all tables that are not children of a `section` element, you would write `*:not(section) > table`. Similarly, to select table header cells that are not part of the table header, you'd write something like `table *:not(thead) > tr > th`, with a result like that shown in Figure 3-24.

State	Post	Capital	State Bird
Alabama	AL	Montgomery	Yellowhammer
Alaska	AK	Juneau	Willow Ptarmigan
Arizona	AZ	Phoenix	Cactus Wren
Arkansas	AR	Little Rock	Mockingbird
California	CA	Sacramento	California Quail
Colorado	CO	Denver	Lark Bunting
Connecticut	CT	Hartford	American Robin
Delaware	DE	Dover	Blue Hen Chicken
Florida	FL	Tallahassee	Northern Mockingbird
Georgia	GA	Atlanta	Brown Thrasher
State	**Post**	**Capital**	**State Bird**

Figure 3-24. Styling header cells outside the table's head area

What you cannot do is nest negation pseudo-classes; thus, `p:not(:not(p))` is invalid and will be ignored. It's also, logically, the equivalent of just writing `p`, so there's no point anyway. Furthermore, you cannot reference pseudo-elements (which we'll cover shortly) inside the parentheses, since they are not simple selectors. You can include attribute selectors and pseudoclasses; they may seem complicated, but they are simple selectors.

Technically, you can put a universal selector into the parentheses, but there's very little point. After all, `p:not(*)` would mean "Select any `p` element as long as it isn't any element," and there's no such thing as an element that is not an element. Very similar to that would be `p:not(p)`, which would also select nothing. It's also possible to write things like `p:not(div)`, which will select any `p` element that is not a `div` element—in other words, all of them. Again, there is very little reason to do this.

On the other hand, it's possible to chain negations together to create a sort of "and also not this" effect. For example, you might want to select all elements with a `class` of `link` that are neither list items nor paragraphs:

```
*.link:not(li):not(p) {font-style: italic;}
```

That translates to "Select all elements with a `class` whose value contains the word `link` as long as they are neither `li` nor `p` elements." This used to be the only way to exclude a group of elements, but CSS (and browsers) support selector lists in negations. That allows us to rewrite the previous example like so:

```
*.link:not(li, p) {font-style: italic;}
```

Along with this came the ability to use more complex selectors, such as those using descendant combinators. If you need to select all elements that are descended from a `form` element, but do not immediately follow a `p` element, you could write it as:

```
form *:not(p + *)
```

Translated, that's "select any element that is not the adjacent sibling a `p` element, and is also the descendant of a `form` element". And you can put these into groups, so if you also want to exclude list items and table-header cells, it would go something like this:

```
form *:not(p + *, li, thead > tr > th)
```

> **NOTE**
>
> The ability to use complex selectors in `:not()` only came to browsers in early 2021, so exercise caution when using it, especially in legacy settings.

One thing to watch out for with `:not()` is that you can have situations where rules combine in unexpected ways, mostly because we're not used to thinking of selection in the negative. Consider this test case:

```
div:not(.one) p {font-weight: normal;}
div.one p {font-weight: bold;}

<div class="one">
   <div class="two">
      <p>I'm a paragraph!</p>
   </div>
</div>
```

The paragraph will be boldfaced, not normal-weight. This is because both rules match: the `p` element is descended from a `div` whose `class` does not contain the word `one` (`<div class="two">`), but it is *also* descended from a `div` whose `class` contains the word `one`. Both rules match, and so both apply. Since there is a conflict, the cascade (which is explained in the next chapter) is used to resolve the conflict, and the second rule wins. The structural arrangement of the markup, with the `div.two` being "closer" to the paragraph than `div.one`, is irrelevant.

The matches-any pseudo-classes

CSS has two pseudo-classes that allow for group matching within a complex selector, `is()` and `:where()`. These are almost identical to each other, with just a minor difference that we'll cover once we understand how they work. Let's start with `is()`.

Suppose you want to select all list items, whether or not they are part of an ordered or an unordered list. The traditional way to do that is:

```
ol li, ul li {font-style: italic;}
```

With `:is()`, we can rewrite that like so:

```
:is(ol, ul) li {font-style: italic;}
```

The matched elements will be exactly the same: all list items that are part of either ordered or unordered lists.

This might seem slightly pointless: not only is the syntax slightly less clear, it's also one character longer. And it's true that in simple situations like that, `:is()` isn't terribly compelling. The more complex the situation, though, the more likely `:is()` will really shine.

> **NOTE**
>
> `:is()` used to be called `:matches()` before a 2021 rename, and was also present in vendor-prefixed form as `:-wekbit-any()` and `:-moz-any()` in still older browsers.

For example, what if we want to style all list items that are at least two levels deep in nested lists, no matter what combination of ordered and unordered lists are above them? Compare the following rules, both of which will have the effect shown in Figure 3-25, except one uses the traditional approach and the other uses `:is()`.

```
ol ol li, ol ul li, ul ol li, ul ul li {font-style: italic;}

:is(ol, ul) :is(ol, ul) li {font-style: italic;}
```

- It's a list
- A right smart list
    1. *Within, another list*
        - *This is deep*
        - *So very deep*
    2. *A list of lists to see*
- And all the lists for me!

Figure 3-25. Using matches-any to select elements

Now consider what the traditional approach would look like for three, four, or even more levels deep of nested lists!

This can be used in all sort of situations: selecting all links inside lists that are themselves inside headers, footers, and `nav` elements could look like this:

```
:is(header,footer,nav,#header,#footer) :is(ol,ul) a[href] {font-style: italic;}
```

Even better: the list of selectors inside `:is()` is what's called a "forgiving selector list." By default, if any one thing in a selector is invalid, the whole rule is marked invalid. Forgiving selector lists, on the other hand, will throw any part that's invalid and honor the rest.

So, given all that, what's the difference between `:is()` and `:where()`? The sole difference between them is that `is()` takes the specificity of the most-specific selector in its selector list, whereas `:where()` has zero specificity. If that last sentence didn't make sense to you, don't worry! We haven't discussed specificity yet, but will in the next chapter.

WARNING

`:is()` and `:where()` only came to browsers in early 2021, so exercise caution when using them, especially in legacy settings.

Selecting defined elements

As the web has advanced, it's added more and more capabilities. One of the more recent is the ability to add custom elements to your markup in a standardized way. This happens a lot with pattern libraries, which often define Web Components based on elements that are specific to the library.

One thing such libraries do to be more efficient is hold off on defining an element until it's needed, or it's ready to be populated with whatever content is supposed to go into it. Such a custom element might look like this in markup:

```
<mylib-combobox>options go here</mylib-combobox>
```

The actual goal is to fill that combobox (a dropdown list that also allows users to enter arbitrary values) with whatever options the backend CMS (Content Management System) provides for it, downloaded via a script that requests the latest data in order to build the list locally, and removing the placeholder text in the process. But what happens if the server fails to respond, leaving the custom element undefined and stuck with its placeholder text? Without taking steps, the text "options go here" will get inserted into the page, probably with minimal styling.

That's where `:defined` comes in. You can use it to select any defined element, and combine it with `:not()` to select elements that aren't yet defined. Here's a simple way to hide undefined comboboxes, and also to apply styles to defined comboboxes.

```
mylib-combobox:not(:defined) {display: none;}
mylib-combobox:defined {display: inline-block;}
mylib-combobox {font-size: inherit; border: 1px solid gray; outline: 1px solid silver;}
```

The :has() pseudo-class

This one is a little bit tricky, because it doesn't quite follow all the rules we've been working under until now—but as a result, it's also *insanely* powerful.

Imagine you want to apply special styles to any `div` element that contains an image. Another way of saying that is that if a `div` element *has* an `img` element inside it, you want to apply certain styles to the `div`. And that's exactly what `:has()` makes possible.

The previous example would be written something like this, with the result illustrated in Figure 3-26:

```
div:has(img) {
        border: 3px double gray;
}

<div>
   <img src="masthead.jpg" alt="">
```

```
</div>
<div>
   <p>No image here!</p>
</div>
<div>
   <p>This has text and <img src="redbox.gif" alt="an image">.
</div>
```

No image here!

Figure 3-26. Using :has() to select elements

The second `div`, which does not have an `img` element as a descendant, doesn’t get the border. If you only wanted the first `div` to get the border, because you actually wanted to only style div`s that have images as direct children, just modify the selector to use the child combinator, like this: `div:has(> img). That would prevent the third `div` from getting the border.

`:has()` is, in one very real sense, the mythical “parent selector” CSS authors have wished for since the beginning of CSS itself. Except it isn’t just for parent selection, because you can select based on siblings, or make the selection happen as far up the ancestry chain as you like. And if all that didn’t quite make sense to you, hang on: we’ll explain further.

There are two things to note right off the bat:

1. Inside the parentheses of `:has()`, you can provide a comma-separated list of selectors, and each of those selectors can be simple or complex.

2. Those selectors are considered relative to the element to which the `:has()` is attached.

Let’s take those in order. All of the following are valid `:has()` uses:

```
table:has(tbody th) {…}
/* tables whose body contains table headers */

a:any-link:has(img:only-child) {…}
/* links containing only an image */

header:has(nav, form.search) {…}
/* headers containing either nav or a form classed search */

section:has(+ h2 em, table + table, ol ul ol ol) {…}
/* sections immediately followed by an 'h2' that contains an 'em'
   OR that contain a table immediately followed by another table
   OR that contain an 'ol' inside an 'ol' inside a 'ul' inside an 'ol' */
```

That last example might be a bit overwhelming, so let’s break it down a bit further. We could restate in a longer way, like this:

```
section:has(+ h2 em),
section:has(table + table),
section:has(ol ul ol ol) {…}
```

And here are two examples of the markup patterns that would be selected:

```
<section>(…section content…)</section>
<h2>I’m an h2 with an <em>emphasis element</em> inside, which means
    the section right before me gets selected!</h2>

<section>
<h2>This h2 doesn’t get the section selected, because it’s a child of
    the section, not its immediately-following sibling</h2>
<p>This paragraph is just here.</p>
<aside>
<h3>Q1 Results</h3>
```

```
<table>(…table contents…)</table>
<table>(…table contents…)</table>
</aside>
<p>Those adjacent-sibling tables mean this paragraph’s parent section element
   DOES get selected!</p>
</section>
```

In the first example, the selection isn’t based on parentage or any other ancestry: instead, the `section` is selected because its immediate sibling (the `h2`) has an `em` element as one of its descendants. In the second, the *section* is selected because it has a descendant `table` that’s immedaitely followed by another `table`, both of which happen in this case to be inside an `aside` element. That makes this specific example one of grandparent selection, not parent selection, because the `section` is grandparent to the tables.

Right, so that’s the first point that was raised earlier. The second was that the selectors inside the parentheses are relative to the element bearing the `:has()`. What that means is that, for example, the following selector is never going to match anything:

```
div:has(html body h1)
```

That’s because while an `h1` can certainly be a descendant of a `div`, the `html` and `body` elements cannot. What that selector means, translated into English, is: “select any `div` that has a descendant `html` which itself has a descendant `body` which has a descendant `h1`”. `html` will never be a descendant of `div`, so this selector can’t match.

To pick something a little more realisitc, here’s a bit of markup showing lists nested inside each other, which has the document structure shown in Figure 3-27.

```
<ol>
<li>List item</li>
<li>List item
        <ul>
        <li>List item</li>
        <li>List item</li>
        <li>List item</li>
        </ul>
</li>
<li>List item</li>
<li>List item</li>
<li>List item
        <ul>
        <li>List item</li>
        <li>List item
                <ol>
                <li>List item</li>
                <li>List item</li>
                <li>List item</li>
                </ol>
        </li>
        <li>List item</li>
        </ul>
</li>
</ol>
```

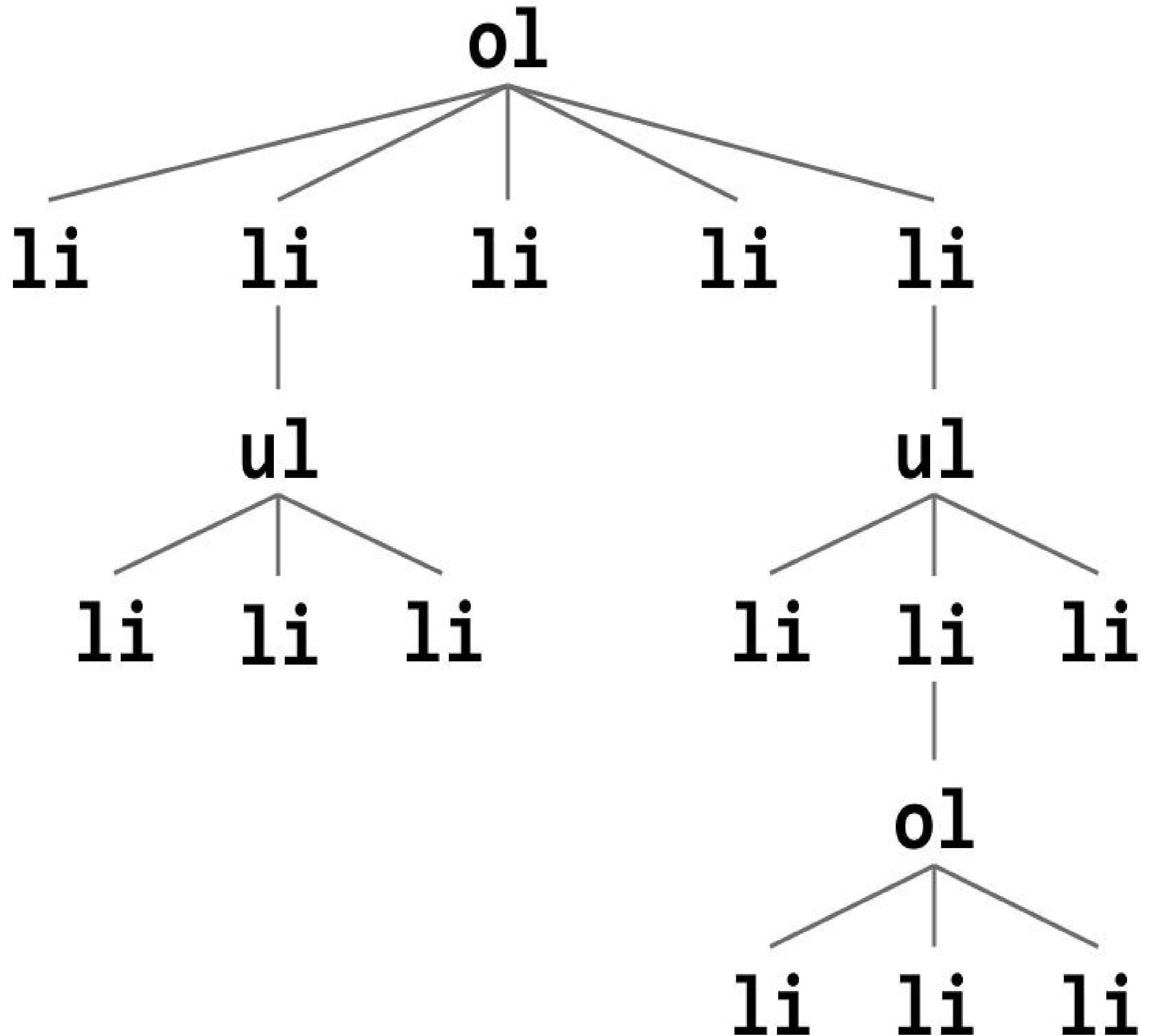


Figure 3-27. A fragment of a document's structure

To that structure, we'll apply the following rules. Spoiler alert: one of them will match an element, and the other will not.

```
ul:has(li ol) {border: 1px solid red;}
ul:has(ol ul ol) {font-style: italic;}
```

The first causes the browser to look at all the `ul` elements. For any `ul` it finds, it looks at the structure of the elements that descend from that `ul`. If it finds an `li ol` relationship in the elements that descend from the `ul`, then the `ul` is matched, and in this case will be given a red border.

If we study the markup structure, either in the code or in Figure 3-27, we can see there are two `ul` elements. The first has `li` descendants, but not any `ol` descendants, so it won't be matched. The second `ul` also has `li` descendants, and one of them as an `ol` descendant. It's a match! The `ul` will be given a red border.

The second rule also causes the browser to look at all the `ul` elements. In this case, for any `ul` it finds,

the browser looks to see if there is an `ol ul ol` relationship within the descendants of the `ul`. Elements outside the `ul` don't count: only those within it are considered. Of the two `ul` elements in the document, neither has an `ol` inside a `ul` that's inside another `ol` that is itself descended from the `ul` being considered. There's no match, so neither of the `ul` elements will be italicized.

Even more powerfully, you're free to mix `:has()` with other pseudo-classes. You might, for example, want to select any heading level if it has an image inside. There are two ways to do this: the long clumsy way, and the compact way. Both are shown here:

```
h1:has(img), h2:has(img), h3:has(img), h4:has(img), h5:has(img), h6:has(img)

:is(h1, h2, h3, h4, h5, h6):has(img)
```

The two selectors have the same outcome, which is to say, if an element *is* one of the listed heading elements, and that element *has* among its descendant elements an `img` element, then the heading will be selected.

For that matter, you could select any headings that *don't* have images inside:

```
:is(h1, h2, h3, h4, h5, h6):not(:has(img))
```

That is to say: if an element *is* one of the listed heading levels, but an `img` element is *not* one of the descendants it *has*, then the heading will be selected. If we bring them together and apply them to a number of headings, we get the results shown in Figure 3-28.

Text

Text and

Text

Text and

Text

Text and

Text

Text and

Text

Text and

Text

Text and

Figure 3-28. To has and has not

As you can already see, there is a lot of power in this selector. There are also dangers: it is entirely possible to write selectors that cause major performance hits to the browser, especially in settings where scripting may be used to modify the document structure. Consider the following:

```
div:has(*.popup) {…}
```

This is saying, “apply these styles to any `div` that has an element with a `class` of `popup` as a descendant.” When the page is loaded into the browser, it has to check all the `div`s to see if they match this selector. That could mean a few trips up and down the document’s structural tree, but ideally it would resolve in less than a second, and the page can then be displayed.

But suppose then we have a script that can add `.popup` to an element, or even several elements, on the page. As soon as the class values change, the browser not only has to check to see if there are styles that apply to `.popup` elements and their descendants, it also has to check to see if there are any ancestor or sibling elements that are affected by this change. Instead of only looking down the document tree, the browser now has to look up as well. And any change triggered by this could mean changes all throughout the page’s layout, both when an element is marked as `.popup` and whenever a `.popup` element loses that class value, potentially affecting elements in entirely different parts of the document.

This sort of performance hit is why there hasn’t been a “parent selector” or anything like it before. Computers are getting fast enough, and browser engines smart enough, that this is much less of a worry than it was in the past — but it’s still something to keep in mind, and test out thoroughly.

NOTE

It is not possible to nest pesudo-elements like `::first-line` or `::selection` in `has()`. (We’ll discuss pseudo-elements shortly.)

WARNING

As of mid-2022, the Firefox family of browser did not support `:has()`, though there were plans to add it. That said, be careful not to use it in ways that make its support necessary, particularly in the years following 2022.

Other pseudo-classes

There are even more pseudo-classes defined in in the CSS Selectors specification, but they are barely supported in browsers, or in some cases not supported at all as of early 2022, or else are things we’ll cover elswhere in the book. We’re listing them here for the sake of completeness, and to point you toward pseudo-classes that might be supported between this edition of the book and the next one. (Or could be replaced with an equivalent pseudo-class with a different name; that happens sometimes.)

Table 3-5. Other pseudo-classes

Name	Description
`:nth-col()`	Refers to table cells or grid items that are in an *nth* column, which is found using the *An + b* pattern. Essentially the same as `:nth-child()`, but refers specifically to table or grid columns.
`:nth-last-col()`	Refers to table cells or grid items that are in an *nth*-last column, which is found using the *An + b* pattern. Essentially the same as `:nth-last-child()`, but refers specifically to table or grid columns.
`:left`	Refers to any left-hand page in a printed document. See XREF HERE for more.
`:right`	Refers to any right-hand page in a printed document. See XREF HERE for more.
`:fullscreen`	Refers to an element that is being displayed fullscreen; e.g., a video that's in fullscreen mode.
`:past`	Refers to an element that appeared before (in time) an element being matched by `:current`.
`:current`	Refers to an element, or the ancestor of an element, that is currently being displayed in a time-based format like a video; e.g., an element containing closed-caption text.
`:future`	Refers to an element that will appear after (in time) an element being matched by `:current`.
`:paused`	Refers to any element that can have the states "playing" or "paused" (e.g., audio, video, etc.) when it is in the "paused" state.
`:playing`	Refers to any element that can have the states "playing" or "paused" (e.g., audio, video, etc.) when it is in the "playing" state.
`:picture-in-picture`	Refers to an element that is used as a picture-in-picture display.

Pseudo-Element Selectors

Much as pseudo-classes assign phantom classes to anchors, pseudo-elements insert fictional elements into a document in order to achieve certain effects.

Unlike the single colon of pseudo-classes, pseudo-elements employ a double-colon syntax, like `::first-line`. This is meant to distinguish pseudo-elements from pseudo-classes. This was not always the case—in CSS2, both selector types used a single colon—so for backward compatibility, browsers will accept single-colon pseudo-type selectors. Don't take this as an excuse to be sloppy, though! Use the proper number of colons at all times in order to future-proof your CSS; after all, there is no way to predict when browsers will stop accepting single-colon pseudo-type selectors.

Styling the First Letter

The `::first-letter` pseudo-element styles the first letter, or a leading punctuation character and the first letter (if the text starts with punctuation), of any non-inline element. This rule causes the first letter of every paragraph to be colored red:

```
p::first-letter {color: red;}
```

The ::first-letter pseudo-element is most commonly used to create an “initial cap” or “drop cap” typographic effect. You could make the first letter of each p twice as big as the rest of the heading, though you may want to only apply this styling to the first letter of the first paragraph:

```
p:first-of-type::first-letter {font-size: 200%;}
```

The result of this rule is illustrated in Figure 3-29.

This is an h2 element

Figure 3-29. The ::first-letter pseudo-element in action

This rule effectively causes the user agent to style a fictional, or “faux”, element that encloses the first letter of each p. It would look something like this:

```
<p><p-first-letter>T</p-first-letter>his is a p element, with a styled first
    letter</h2>
```

The ::first-letter styles are applied only to the contents of the fictional element shown in the example. This <p-first-letter> element does *not* appear in the document source, nor even in the DOM tree. Instead, its existence is constructed on the fly by the user agent and is used to apply the ::first-letter style(s) to the appropriate bit of text. In other words, <p-first-letter> is a pseudo-element. Remember, you don’t have to add any new tags. The user agent styles the first letter for you as if you had encased it in a styled element.

The first letter is defined as the first typographic letter unit of the originating element, if it is not preceded by other content, like an image. The specifications use the term “letter unit” because some languages have letters made up of more than character, like “œ” in Old West Norse. Punctuation that precedes or follows the first letter unit, even if there are several such symbols, should be included in the ::first-letter pseudo-element. The browser does this for you.

Styling the First Line

Similarly, ::first-line can be used to affect the first line of text in an element. For example, you could make the first line of each paragraph in a document large and purple:

```
p::first-line {
  font-size: 150%;
  color: purple;
}
```

In Figure 3-30, the style is applied to the first displayed line of text in each paragraph. This is true no matter how wide or narrow the display region is. If the first line contains only the first five words of the paragraph, then only those five words will be big and purple. If the first line contains the first 30 words of the element, then all 30 will be big and purple.

This is a paragraph of text that has only one stylesheet applied to it. That style causes the first line to be big and purple. No other line will have those styles applied.

Figure 3-30. The ::first-line pseudo-element in action

Because the text from “This” to “only” should be big and purple, the user agent employs a fictional markup that looks something like this:

```
<p>
<p-first-line>This is a paragraph of text that has only</p-first-line>
one stylesheet applied to it. That style causes the first line to
be big and purple. No other line will have those styles applied.
</p>
```

If the first line of text were edited to include only the first seven words of the paragraph, then the fictional `</p-first-line>` would move back and occur just after the word “that.” If the user were to increase or decrease the font-size rendering, or expand or contract the browser window causing the width of the text to change, thereby causing the number of words on the first line to increase or decrease, the browser automatically sets only the words in the currently displayed first line to be both big and purple.

The length of the first line depends on a number of factors, including the font-size, letter spacing, width of the parent container, etc. Depending on the markup and the length of that first line, it is possible that the end of the first line comes in the middle of a nested element. If the `::first-line` breaks up a nested element, such as an `em` or a hyperlink, the properties attached to the `::first-line` will only apply to the portion of that nested element that is displayed on the first line.

Restrictions on ::first-letter and ::first-line

The `::first-letter` and `::first-line` pseudo-elements currently can be applied only to block-display elements such as headings or paragraphs, and not to inline-display elements such as hyperlinks. There are also limits on the CSS properties that may be applied to `::first-line` and `::first-letter`. The following table gives an idea of these limitations.

++ <table id="properties_permitted_on_pseudo-elements"> <caption>Properties permitted on pseudo-elements</caption> <thead> <tr> <th>::first-letter</th> <th>::first-line</th> </tr> </thead> <tbody> <tr> <td> All font properties All background properties All text decoration properties All inline typesetting properties All inline layout properties All border properties <code>box-shadow</code> <code>color</code> <code>opacity</code> </td>

<td> All font properties All background properties All margin properties All padding properties All border properties All text decoration

properties All inline typesetting properties <code>color</code>
<code>opacity</code> </td> </tr>

</tbody> </table> ++

The Placeholder Text Pseudo-Element

As it happens, the restrictions on what styles can be applied via `::first-line` are exactly the same as the restrictions on styles applied via `::placeholder`. This pseudo-element matches any placeholder text placed into text inputs and textareas. You could, for example, italicize text input placeholder text and turn textarea placeholder text a dusky blue like this:

```
input::placeholder {font-style: italic;}
textarea::placeholder {color: cornflowerblue;}
```

For both `input` and `textarea` elements, this text is defined by the `placeholder` attribute in HTML. The markup will look something very much like this:

```
<input type="text" placeholder="(XXX) XXX-XXXX" id="phoneno">
<textarea placeholder="Tell us what you think!"></textarea>
```

If text is pre-filled using the `value` attribute on `input` elements, or by placing content inside the `textarea` element, that will override the value of any `placeholder` attribute, and the resulting text won't be selected with the `::placeholder` pseudo-element.

The Form Button Pseudo-ELement

Speaking of forms elements, it's also possible to directly select the file-selector button — and *only* the file-selector button — in an `input` element that has a `type` of `file`. This gives you a way to call attention to the button a user needs to click to open the file-selection dialog, even if no other part of the input can be directly styled.

If you've never seen a file-selection input, it usually looks like this:

```
<label for="uploadField">Select file from computer</label>
<input id="uploadField" type="file">
```

That second line gets replaced with a control whose appearance is dependent on the combination of operating system and browser, so it tends to look at least a little different (sometimes a lot different) from one user to the next. Figure 3-31 shows on possible rendering of the input, with the button styled by the following CSS.

```
input::file-selector-button {
   border: thick solid gray;
   border-radius: 2em;
}
```

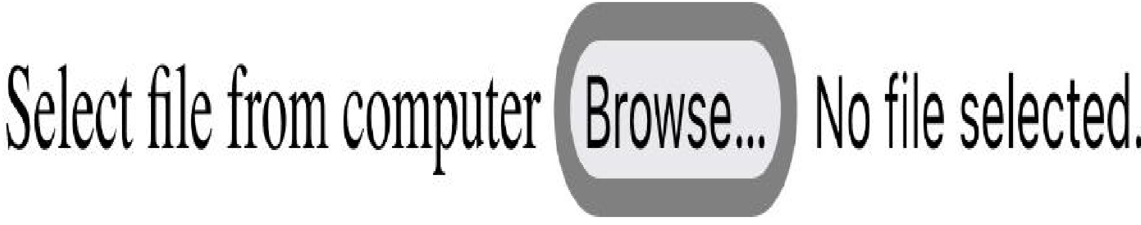


Figure 3-31. Styling the button in a file submission input

Styling (or Creating) Content Before and After Elements

Let’s say you want to preface every `h2` element with a pair of silver square brackets as a typographical effect:

```
h2::before {content: "]]"; color: silver;}
```

CSS lets you insert *generated content*, and then style it directly using the pseudo-elements `::before` and `::after`. Figure 3-32 illustrates an example.

]]This is an h2 element

Figure 3-32. Inserting content before an element

The pseudo-element is used to insert the generated content and to style it. To place content at the end of an element, right before the closing tag, use the pseudo-element `::after`. You could end your documents with an appropriate finish:

```
body::after {content: "The End.";}
```

Conversely, if you want to insert some content at the beginning of an element, right after the opening tag, use `::before`. Just remember that in either case, you have to use the `content` property in order to insert something to style.

Generated content is its own subject, and the entire topic (including more detail on `::before` and `::after`) is covered more thoroughly in XREF HERE.

Highlight pseudo-elements

A relatively new concept in CSS is the ability to style pieces of content that have been highlighted, either by user selection or by the user agent itself. These are summarized in Table 3-6.

Table 3-6. Highlight pseudo-elements

Name	Description
`::selection`	Refers to any part of a document that has been highlighted for user operation; e.g., text which has been drag-selected with a mouse.
`::target-text`	Refers to the text of a document which has been targeted. This is distinct from the `:target` pseudo-class, which refers to a targeted element as a whole, not a fragment of text.
`::spelling-error`	Refers to the part of a document that has been marked by the user agent as a misspelling.
`::grammar-error`	Refers to the part of a document that has been marked by the user agent as a grammar error.

Of the four pseudo-elements in Table 3-6, only one, `::selection`, has any appreciable support as of early 2022. So we'll explore it, and leave the others for a future edition.

When a user uses a mouse pointer to click-hold-and-drag in order to highlight some text, that's a selection. Most browsers have default styles set for text selection. Authors can apply a limited set of CSS properties to such selections, overriding the browser's default styles, by styling the `::selection` pseudo-element. Let's say you want selected text to be white on a navy-blue background. The CSS for that would look like this:

```
::selection {color: white; background-color: navy;}
```

The primary use cases for `::selection` are when you want to specify a color scheme for selected text that doesn't clash with the rest of the design, or when you want to define different selection styles for different parts of a document. For example:

```
::selection {color: white; background-color: navy;}
form::selection {color: silver; background-color: maroon;}
```

Be careful in styling selection highlights: users generally expect text they select to look a certain way, usually defined by settings in their operating system. Thus, if you get too clever with selection styling, you could confuse users. That said, if you know that selected text can be difficult to see because your design's colors tend to obscure it, defining more obvious highlight styles is probably a good idea.

Note that selected text can cross element boundaries, and that there can be multiple selections within a given document. Imagine a situation where a user selects text starting from the middle of one paragraph to the middle of the next. In effect, each paragraph will get its own selection pseudo-element nested inside, and selection styling will be handled as appropriate for the context. This means that, given the following CSS and HTML, you'll get a result like that shown in Figure 3-33.

```
.p1::selection {color: silver; background-color: black;}
```

```
.p2::selection {color: black; background-color: silver;}

<p class="p1">This is a paragraph with some text that can be selected, one of two.</p>
<p class="p2">This is a paragraph with some text that can be selected, two of two.</p>
```

This is a paragraph with some text that can be selected, one of two.

This is a paragraph with some text that can be selected, two of two.

Figure 3-33. Selection styling

This underscores a point made earlier: **be careful** with your selection styling. It is all too easy to make text unreadable for some users, particularly if your selection styles interact badly with the user's default selection styles.

Furthermore, you can only apply a limited number of CSS properties to selections: `color`, `background-color`, `text-decoration` and related properties, `text-shadow`, and the `stroke` properties (in SVG).

> **NOTE**
>
> As of early 2022, selections did not have their styles inherited: selecting text containing some inline elements would apply the selection styling to the text outside the inline elements, but not within the inline elements. It is not clear if this behavior is intended, but it was consistent across major browsers.

Beyond `::selection`, there will likely be increasing support for `::target-text`. As of early 2022, this was only supported in Chromium browsers, which introduced a feature that needs it. With this feature, text can be added to the end of a URL as part of the fragment identifier for highlighting, in order to draw attention to one or more parts of the page.

For example, a URL might look something like:
`https://example.org/#:~:text=for%20use%20in%20illustrative%20examples`.
The part at the end says to the browser, “once you’ve loaded the page, highlight any examples of this text.” The text is encoded for use in URLs, which is why it’s filled with `%20` strings — they represent spaces. The result will look something like Figure 3-34.

Figure 3-34. Targeted text styling

If you wanted to suppress this content highlighting on your own pages, you might do something like this:

```
::target-text {color: inherit; background-color: inherit;}
```

As for `::spelling-error` and `::grammar-error`, these are meant to apply highlighting of some sort to any spelling or grammar errors within a document. You can see the utility for something like Google Docs or the editing fields of content management systems like WordPress or Craft. For most other things, though, they seem unlikely to be very popular. Regardless, as of this writing, there was no browser support for either, and the Working Group was still hashing out the details of how they should work.

The backdrop pseudo-element

Suppose you have an element that's being presented full-screen, like a video. Furthermore, suppose that element doesn't neatly fill the full screen all the way to the edges, perhaps because the aspect ratio of the element doesn't match the aspect ratio of the screen. What should be filled in for the parts of the screen where the element doesn't reach? And how would you do select that non-element region with CSS?

Enter the `::backdrop` pseudo-element. This represents a box that's the exact size of the full-screen viewport, and it is always drawn beneath a fullscreen element. So you might put a dark-gray backdrop behind any fullscreen video like this:

```
video::backdrop {background: #111;}
```

There aren't any restrictions on what styles can be applied to backdrops, but since they're essentially empty boxes placed behind a fullscreen element, most of the time, you'll probably be setting background colors or images.

An important thing to remember is that backdrops do *not* participate in inheritance. That means they can't inherit styles from ancestor elements, nor do they pass any of their styles on to any children. Whatever styles you apply to the backdrop will exist in their own little pocket universe.

The video-cue pseudo-element

On the subject of videos, some videos can have WebVTT (Web Video Text Tracks) data containing the text captions. These captions are known as *cues*, and can be styled with the `::cue` pseudo-element.

Let's say you have a video that's mostly dark, with a few light segments. You might then style the cues to be a light-ish gray text on a translucent dark background, as follows:

```
::cue {
  color: silver;
  background: rgba(0,0,0,0.5);
}
```

This will always apply to the currently-visible cue.

You can also select parts of individual cues using a selector pattern inside parentheses. This can be used to style specific elements defined in the WebVTT data, drawn from a small list allowed by the WebVTT specification. For example, any italicized cue text could be selected as follows:

```
::cue(i) {…}
```

It is possible to use stuctural pseudo-classes like `:nth-child`, but these will only apply within a given cue, not across cues. That is, you can't select every other cue for styling, but you can select every other element within a given cue. Assume the following WebVTT data:

```
00:00:01.500 --> 00:00:02.999
<v Hildy>Tell me, is the lord of the universe in?</v>

00:00:03.000 --> 00:00:04.299
- Yes, he's in.
- In a bad humor.
```

In the second cue, there are two lines of text. These are treated as separate elements, in effect, even though no elements are specified. Thus, we could make "Hildy"'s text in the first cue boldface, and give alternate colors to the two lines of dialogue in the second cue, like so:

```
::cue(v[voice="Hildy"]) {font-weight: bold;}
::cue(:nth-child(odd)) {color: yellow;}
::cue(:nth-child(even)) {color: white;}
```

As of early 2022, there is a limited range of properties that can be applied to cues. They are:

- `color`
- `background` and its associated longhand properties (e.g., `background-color`)
- `text-decoration` and its associated longhand properties (e.g., `text-decoration-thickness`)
- `text-shadow`
- `text-combine-upright`
- `font` and its associated longhand proeprties (e.g., `font-weight`)
- `ruby-position`
- `opacity`
- `visibility`
- `white-space`
- `outline` and its associated longhand properties (e.g., `outline-width`)

Shadow Pseudo-classes and -Elements

Another recent innovation in HTML has been the introduction of the Shadow DOM, which is a very deep and complex subject we don't have the space to explore here. At a very basic level, the Shadow DOM allows developers to create encapsulated markup, style, and scripting within the regular (or "light") DOM. This keeps the styles and scripts of one shadow DOM from affecting any other part of the

document, whether those parts are in the light or shadow DOM.

We're bringing this up here because CSS does provide ways to hook into Shadow DOMs, as well as to reach up from within a shadow DOM to select the piece of the light DOM that hosts the shadow. (This all sounds very panel-van-artistic, doesn't it?)

Shadow pseudo-classes

To see what this means, let's bring back the combobox example from earlier in the chapter. It looked like this:

```
<mylib-combobox>options go here</mylib-combobox>
```

Now, within this custom element, a whole set of scripting and CSS could be attached. These scripts and styles would apply **only** within the `mylib-combobox` element. Even if the CSS says something like `li {color: red;}`, that will only apply to `li` elements constructed within the `mylib-combobox`. It can't leak out to turn list items elsewhere on the page red.

That's all good, but what if you want to style the host element in a certain way? The host element, more generally called the *shadow host*, is in this case `mylib-combobox`. From within the shadow host, CSS can select the host using the `:host` pseudo-class. For example:

```
:host {border: 2px solid red;}
```

That will reach up, so to speak, "pierce through the shadow boundary" (to use an evocative phrase from the specification), and select the `mylib-combobox` element.

Now, suppose there can be different kinds of combo boxes, each with its own class. Something like this:

```
<mylib-combobox class="countries">options go here</mylib-combobox>
<mylib-combobox class="regions">options go here</mylib-combobox>
<mylib-combobox class="cities">options go here</mylib-combobox>
```

You might want to style each class of combocbox differently. For that, the `:host()` pseudo-class exists.

```
:host(.countries) {border: 2px solid red;}
:host(.regions) {border: 1px solid silver;}
:host(.cities) {border: none; background: gray;}
```

These rules could then be included in a single stylesheet that's loaded by all comboboxes, using the presence of classes on the shadow hosts to style as appropriate.

But wait! What if, instead of latching on to classes, we want to style our shadow hosts based on where they appear in the light DOM? In that case, `:host-context()` has you covered. Thus, we can style our comboxes one way if they're part of a form, and a different way if they're part of a header navigation element.

```
:host-context(form) {border: 2px solid red;}
```

```
:host-context(header nav) {border: 1px solid silver;}
```

The first of these means, “if the shadow host is the descendant of a `form` element, apply these styles.” The second means, “if the shadow host is the descendant of a `nav` element that is itself descended from a *header* element, apply these styles.” To be clear, `form` and `nav` are **not** the shadow hosts in these situations! The selector in `:host-context()` is only described the context in which the host needs to be placed in order to be selected.

> **NOTE**
>
> As of early 2022, `:host-context()` wasn’t supported by the Firefox family.

Shadow pseudo-elements

In addition to having hosts, Shadow DOMs can also define *slots*. These are elements that are meant to have other things slotted into them, much as you would place an expansion card into an expansion slot. Let’s expand the markup of the combobox by a little bit.

```
<mylib-combobox>
     <span slot="label">Country</span>
     ["shadow-tree"]
          <slot name="label"></slot>
     [/"shadow tree"]
</mylib-combobox>
```

Now, to be clear, the `"shadow tree"` thing there isn’t actual markup. It’s just there to represent the shadow DOM that gets constructed by whatever script builds it. So please don’t go writing square-bracketed quoted element names into your documents: they will fail.

That said, given a setup like the above, the `span` would be slotted into the `slot` element, because the names match. You could try applying styles to the slot, but what if you’d rather style the thing that got plugged into the slot? That’s represented by the `::slotted()` pseudo-element, which accepts a selector as needed.

Thus, if you want to style all slotted elements one way and then add some extra style if the slotted element is a `span`, you would write something like:

```
::slotted(*) {outline: 2px solid red;}
::slotted(span) {font-style-italic;}
```

More practically, you could style all slots red, and then remove that red from any slot that’s been slotted with content, thus making the slots that failed to get any content stand out. Something like this:

```
slot {color: red;}
::slotted(*) {color: black;}
```

WARNING

The Shadow DOM and its use is a complex topic, and one which we have not even begun to scratch the surface of in this section. Our only goal was to introduce the pseudo-classes and -elements that pertain to the Shadow DOM, not explain the Shadow DOM or illustrate best practices.

Summary

As we saw in this chapter, pseudo-classes and pseudo-elements bring a whole lot of power and flexibility to the table. Whether selecting hyperlinks based on their visited state, matching elements based on their placement in the document structure, or styling pieces of the Shadow DOM, there’s a pseudo selector for nearly every taste.

In this chapter and the last one, we’ve mentioned the concepts of “specificity” and “the cascade” a few times, and promised to talk about them soon. Well, “soon” is now: That’s exactly what we’ll do in the next chapter.

Chapter 4. Specificity, Inheritance, and the Cascade

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at rfernando@oreilly.com.

Chapter 2 showed how document structure and CSS selectors allow you to apply a wide variety of styles to elements. Knowing that every valid document generates a structural tree, you can create selectors that target elements based on their ancestors, attributes, sibling elements, and more. The structural tree is what allows selectors to function and is also central to a similarly crucial aspect of CSS: inheritance.

Inheritance is the mechanism by which some property values are passed on from an element to a descendant element. When determining which values should apply to an element, a user agent must consider not only inheritance but also the *specificity* of the declarations, as well as the origin of the declarations themselves. This process of consideration is what's known as the *cascade*.

We will explore the interrelation between these three mechanisms—specificity, inheritance, and the cascade—in this chapter, but the difference between the latter two can be summed up this way: looking at `h1 {color: red; color: blue;}` and making the `h1` blue happens because of the cascade; and a `span` inside the `h1` also being blue happens because of inheritance.

Above all, regardless of how abstract things may seem, keep going! Your perseverance will be rewarded.

Specificity

You know from Chapter 2 that you can select elements using a wide variety of means. In fact, it often happens that the same element is selected by two or more rules, each with its own selector. Let's consider the following three pairs of rules. Assume that each pair will match the same element:

```
h1 {color: red;}
body h1 {color: green;}

h2.grape {color: purple;}
h2 {color: silver;}
```

```
html > body table tr[id="totals"] td ul > li {color: maroon;}
li#answer {color: navy;}
```

Only one of the two rules in each pair can be applied, or “win,” since the matched elements can be only one color or the other. How do we know which one will win?

The answer is found in the *specificity* of each selector. For every rule, the user agent (i.e., a web browser) evaluates the specificity of the selector and attaches the specificity to each declaration in the rule within the cascade layer that has precedence. When an element has two or more conflicting property declarations, the one with the highest specificity will win out.

> **NOTE**
>
> This isn’t the whole story in terms of conflict resolution, which is a bit more complicated than a single paragraph can cover. For now, just keep in mind that selector specificity is only compared to other selectors that share the same origin and cascade layer. We’ll cover those terms, and more, a bit later in this chapter (in “The Cascade”).

A selector’s specificity is determined by the components of the selector itself. A specificity value can be expressed in three parts, like this: `0,0,0`. The actual specificity of a selector is determined as follows:

- For every ID attribute value given in the selector, add `1,0,0`.
- For every class attribute value, attribute selection, or pseudo-class given in the selector, add `0,1,0`.
- For every element and pseudo-element given in the selector, add `0,0,1`.
- Combinators do not contribute anything to the specificity.
- Anything listed inside a `:where()` pseudo-class, and the universal selector, adds `0,0,0`. (While they do not contribute anything to the specificity weight, they do match elements, unlike combinators.)
- The specificity of an `:is()`, `:not()`, or `:has()` pseudo-class is equal to the specificity of the most specific selector in its selector list argument.

For example, the following rules’ selectors result in the indicated specificities:

```
h1 {color: red;}                          /* specificity = 0,0,1 */
p em {color: purple;}                     /* specificity = 0,0,2 */
.grape {color: purple;}                   /* specificity = 0,1,0 */
*.bright {color: yellow;}                 /* specificity = 0,1,0 */
p.bright em.dark {color: maroon;}         /* specificity = 0,2,2 */
#id216 {color: blue;}                     /* specificity = 1,0,0 */
*:is(aside#warn, code) {color: red;}      /* specificity = 1,0,1 */
div#sidebar *[href] {color: silver;}      /* specificity = 1,1,1 */
```

Given a case where an `em` element is matched by both the second and fifth rules in this example, that element will be maroon because the sixth rule’s specificity outweighs the second’s.

Take special note of the next-last selector, `*:is(aside#warn, code)`. The `:is()` pseudo-class is one of a small group of pseudo-classes where the specificity is equal to the most specific selector in the selector list. Here, the selector list was `aside#warn, code`. The `aside#warn` compound selector has a specificity of `1,0,1` and the `code` selector has a specificity of `0,0,1`. Thus, the whole `:is()` portion of the selector is set to the specificity of the `aside#warn` selector.

Now, let's return to the pairs of rules from earlier in the section and fill in the specificities:

```
h1 {color: red;}          /* 0,0,1 */
body h1 {color: green;}   /* 0,0,2 (winner)*/

h2.grape {color: purple;}  /* 0,1,1 (winner) */
h2 {color: silver;}        /* 0,0,1 */

html > body table tr[id="totals"] td ul > li {color: maroon;}  /* 0,1,7 */
li#answer {color: navy;}                                       /* 1,0,1
   (winner) */
```

We've indicated the winning rule in each pair; in each case, it's because the specificity is higher. Notice how they're listed, and that the order the rules are in doesn't actually matter here.

In the second pair, the selector `h2.grape` wins because it has an extra class: `0,1,1` beats out `0,0,1`. In the third pair, the second rule wins because `1,0,1` wins out over `0,1,7`. In fact, the specificity value `0,1,0` would win out over the value `0,0,13`.

This happens because the values are compared from left to right. A specificity of `1,0,0` will win out over any specificity that begins with a `0`, no matter what the rest of the numbers might be. So `1,0,1` wins over `0,1,7` because the `1` in the first value's first position beats the `0` in the second value's first position.

Declarations and Specificity

Once the specificity of a selector has been determined, the specificity value will be conferred on all of its associated declarations. Consider this rule:

```
h1 {color: silver; background: black;}
```

For specificity purposes, the user agent must treat the rule as if it were "ungrouped" into separate rules. Thus, the previous example would become:

```
h1 {color: silver;}
h1 {background: black;}
```

Both have a specificity of `0,0,1`, and that's the value conferred on each declaration. The same splitting-up process happens with a grouped selector as well. Given the rule:

```
h1, h2.section {color: silver; background: black;}
```

the user agent treats it if it were the following:

```
h1 {color: silver;}             /* 0,0,1 */
h1 {background: black;}         /* 0,0,1 */
h2.section {color: silver;}     /* 0,1,1 */
h2.section {background: black;} /* 0,1,1 */
```

This becomes important in situations where multiple rules match the same element and some of the declarations clash. For example, consider these rules:

```
h1 + p {color: black; font-style: italic;}                /* 0,0,2 */
p {color: gray; background: white; font-style: normal;} /* 0,0,1 */
*.callout {color: black; background: silver;}           /* 0,1,0 */
```

When applied to the following markup, the content will be rendered as shown in Figure 4-1:

```
<h1>Greetings!</h1>
<p class="callout">
It's a fine way to start a day, don't you think?
</p>
<p>
There are many ways to greet a person, but the words are not as important
as the act of greeting itself.
</p>
<h1>Salutations!</h1>
<p>
There is nothing finer than a hearty welcome from one's neighbor.
</p>
<p class="callout">
Although a steaming pot of fresh-made jambalaya runs a close second.
</p>
```

Greetings!

It's a fine way to start a day, don't you think?

There are many ways to greet a person, but the words are not as important as the act of greeting itself.

Salutations!

There is nothing finer than a hearty welcome from one's neighbor.

Although a steaming pot of fresh-made jambalaya runs a close second.

Figure 4-1. How different rules affect a document

In every case, the user agent determines which rules match a given element, calculates all of the associated declarations and their specificities, determines which rules win out, and then applies the

winners to the element to get the styled result. These machinations must be performed on every element, selector, and declaration. Fortunately, the user agent does it all automatically, and nearly instantly. This behavior is an important component of the cascade, which we'll discuss later in this chapter.

Resolving multiple matches

When an element is matched by more than one selector in a grouped selector, the most specific selector is used. Consider the following CSS:

```
li,              /* 0,0,1 */
.quirky,         /* 0,1,0 */
#friendly,       /* 1,0,0 */
li.happy.happy.happy#friendly { /* 1,3,1 */
   color: blue;
}
```

Here we have one rule with a grouped selector, and each of the individual selectors has a very different specificity. Now suppose we find this in our HTML:

```
<li class="happy quirky" id="friendly">This will be blue.</li>
```

Every one of the selectors in the grouped selector applies to the list item! Which one is used for specificity purposes? The most specific. Thus, in this example, the blue is applied with a specificity of `1,3,1`.

You might have noticed that we repeated the `happy` class name three times in one of the selectors. This is a bit of hack that can be used with classes, attributes, pseudo-classes and even ID selectors to increase specificity. Do be careful with it, since artifically inflating specificity can create problems in the future: you might want to override that rule with another, and that rule will need even more classes chained together.

Zeroed Selector Specificity

The universal selector does not contribute to specificity. In other words, it has a specificity of `0,0,0`, which is different than having no specificity (as we'll discuss in "Inheritance"). Therefore, given the following two rules, a paragraph descended from a `div` will be black, but all other elements will be gray:

```
div p {color: black;} /* 0,0,2 */
* {color: gray;}      /* 0,0,0 */
```

This means the specificity of a selector that contains a universal selector along with other selectors is not changed by the presence of the universal selector. The following two selectors have exactly the same specificity:

```
div p          /* 0,0,2 */
body * strong /* 0,0,2 */
```

The same is true for the `:where()` pseudo-class, regardless of whatever selectors might be in its selector list. Thus, `:where(aside#warn, code)` has a specificity of `0,0,0`.

Combinators, including `~`, `>`, `+`, and the space character, have no specificity at all—not even zero specificity. Thus, they have no impact on a selector's overall specificity.

ID and Attribute Selector Specificity

It's important to note the difference in specificity between an ID selector and an attribute selector that targets an `id` attribute. Returning to the third pair of rules in the example code, we find:

```
html > body table tr[id="totals"] td ul > li {color: maroon;} /* 0,1,7 */
li#answer {color: navy;}                                      /* 1,0,1 (wins) */
```

The ID selector (`#answer`) in the second rule contributes `1,0,0` to the overall specificity of the selector. In the first rule, however, the attribute selector (`[id="totals"]`) contributes `0,1,0` to the overall specificity. Thus, given the following rules, the element with an `id` of `meadow` will be green:

```
#meadow {color: green;}      /* 1,0,0 */
*[id="meadow"] {color: red;} /* 0,1,0 */
```

Importance

Sometimes, a declaration is so important that it outweighs all other considerations. CSS calls these *important declarations* (for hopefully obvious reasons) and lets you mark them by inserting the flag `!important` just before the terminating semicolon in a declaration:

```
p.dark {color: #333 !important; background: white;}
```

Here, the color value of `#333` is marked with the `!important` flag, whereas the background value of `white` is not. If you wish to mark both declarations as important, each declaration needs its own `!important` flag:

```
p.dark {color: #333 !important; background: white !important;}
```

You must place the `!important` flag correctly, or the declaration may be invalidated. `!important` *always* goes at the end of a declaration, just before the semicolon. This placement is especially critical when it comes to properties that allow values containing multiple keywords, such as `font`:

```
p.light {color: yellow; font: smaller Times, serif !important;}
```

If `!important` were placed anywhere else in the `font` declaration, the entire declaration would likely be invalidated and none of its styles applied.

> **NOTE**
>
> We realize that to those of you who come from a programming background, the syntax of this token instinctively translates to "not important." For whatever reason, the bang (`!`) was chosen as the delimiter for important flags, and it does *not* mean "not" in CSS, no matter how many other languages give it that very meaning. This association is unfortunate, but we're stuck with it.

Declarations that are marked `!important` do not have a special specificity value, but are instead considered separately from non-important declarations. In effect, all `!important` declarations are grouped together, and specificity conflicts are resolved relatively within that group. Similarly, all non-important declarations are considered as a group, with any conflicts within the non-important group resolved via the cascade, of which specificity is a part. Thus, in any case where an important and a non-important declaration conflict, an important declaration will always win (unless the user agent or user have declared the same property as important, which we'll see later in the chapter.)

Figure 4-2 illustrates the result of the following rules and markup fragment:

```
h1 {font-style: italic; color: gray !important;}
.title {color: black; background: silver;}
* {background: black !important;}

<h1 class="title">NightWing</h1>
```

Figure 4-2. Important rules always win

> **WARNING**
>
> It's generally bad practice to use `!important` in your CSS, and it is rarely needed. If you find yourself reaching for `!important`, stop and look for other ways to get the same result without using `!important`. Cascade Layers are one such possibility; see "Sorting by Cascade Layer" for more details.

Inheritance

Another key concept in understanding how styles are applied to elements is *inheritance*. Inheritance is the mechanism by which some styles are applied not only to a specified element, but also to its descendants. If a color is applied to an `h1` element, for example, then that color is applied to all text inside the `h1`, even the text enclosed within child elements of that `h1`:

```
h1 {color: gray;}

<h1>Meerkat <em>Central</em></h1>
```

Both the ordinary `h1` text and the `em` text are colored gray because the `em` element inherits the value of

`color` from the `h1`. If property values could not be inherited by descendant elements, the `em` text would be black, not gray, and we'd have to color the elements separately.

Consider an unordered list. Let's say we apply a style of `color: gray;` for `ul` elements:

```
ul {color: gray;}
```

We expect that style applied to a `ul` will also be applied to its list items, and also to any content of those list items, including the marker (i.e., the "bullet" next to each list item). Thanks to inheritance, that's exactly what happens, as Figure 4-3 demonstrates.

- Oh, don't you wish
- That you could be a fish
- And swim along with me
- Underneath the sea

1. Strap on some fins
2. Adjust your mask
3. Dive in!

Figure 4-3. Inheritance of styles

It's easier to see how inheritance works by turning to a tree diagram of a document. Figure 4-4 shows the tree diagram for a document much like the very simple document shown in Figure 4-3.

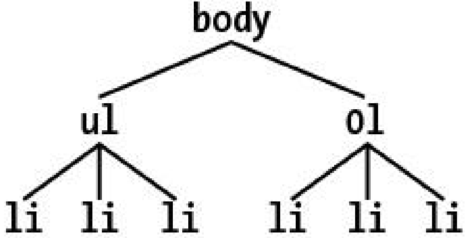


Figure 4-4. A simple tree diagram

When the declaration `color: gray;` is applied to the `ul` element, that element takes on that declaration. The value is then propagated down the tree to the descendant elements and continues on until there are no more descendants to inherit the value. Values are *never* propagated upward; that is, an element never passes values up to its ancestors.

> **NOTE**
>
> There is a notable exception to the upward propagation rule in HTML: background styles applied to the `body` element can be passed to the `html` element, which is the document's root element and therefore defines its canvas. This only happens if the `body` element has a defined background and the `html` element does not. There are a few other properties that share this body-to-root behavior, such as `overflow`, but it only happens with the `body` element. There are no other elements that risk inheriting properties from a descendant.

Inheritance is one of those things about CSS that is so basic that you almost never think about it unless you have to. However, you should still keep a couple of things in mind.

First, note that many properties are not inherited—generally in order to avoid undesirable outcomes. For example, the property `border` (which is used to set borders on elements) does not inherit. A quick

glance at Figure 4-5 reveals why this is the case. If borders were inherited, documents would become much more cluttered—unless the author took the extra effort to turn off the inherited borders.

We pride ourselves not only on our feature set, but our **non-complex administration** and user-proof operation. Our technology takes the best aspects of SMIL and C++. Our functionality is unmatched, but our 1000/60/60/24/7/365 returns-on-investment and non-complex operation is constantly considered a remarkable achievement. The power to enhance perfectly leads to **the aptitude to deploy dynamically**. Think super-macro-real-time. (Text courtesy http://andrewdavidson.com/gibberish/)

Figure 4-5. Why borders aren't inherited

As it happens, most of the box-model properties—including margins, padding, backgrounds, and borders—are not inherited for the same reason. After all, you likely wouldn't want all of the links in a paragraph to inherit a 30-pixel left margin from their parent element!

Second, inherited values have no specificity at all, not even zero specificity. This seems like an academic distinction until you work through the consequences of the lack of inherited specificity. Consider the following rules and markup fragment and compare them to the result shown in Figure 4-6:

```
* {color: gray;}
h1#page-title {color: black;}

<h1 id="page-title">Meerkat <em>Central</em></h1>
<p>
Welcome to the best place on the web for meerkat information!
</p>
```

Meerkat ***Central***

Welcome to the best place on the Web for meerkat information!

Figure 4-6. Zero specificity defeats no specificity

Since the universal selector applies to all elements and has zero specificity, its color declaration's value of `gray` wins out over the inherited value of `black`, which has no specificity at all. (And now you may understand why we listed `:where()` and the universal selector as having `0,0,0` specificity: they add no weight, but do match elements.) Therefore, the `em` element is rendered gray instead of black.

This example vividly illustrates one of the potential problems of using the universal selector indiscriminately. Because it can match *any* element or pseudo-element, the universal selector often has the effect of short-circuiting inheritance. This can be worked around, but it's usually more sensible to avoid the problem in the first place by not using the universal selector by itself indiscriminately.

The complete lack of specificity for inherited values is not a trivial point. For example, assume that a style sheet has been written such that all text in a "toolbar" is to be white on black:

```
#toolbar {color: white; background: black;}
```

This will work so long as the element with an `id` of `toolbar` contains nothing but plain text. If,

however, the text within this element is all hyperlinks (`a` elements), then the user agent's styles for hyperlinks will take over. In a web browser, this means they'll likely be colored blue, since the browser's internal style sheet probably contains an entry like this:

```
a:link {color: blue;}
```

To overcome this problem, you must declare something like this:

```
#toolbar {color: white; background: black;}
#toolbar a:any-link {color: white;}
```

By targeting a rule directly at the `a` elements within the toolbar, you'll get the result shown in Figure 4-7.

Home | Products | Services | Contact | About

Figure 4-7. Directly assigning styles to the relevant elements

Another way to get the same result is to use the value `inherit`, covered in the next chapter. We can alter the previous example like so:

```
#toolbar {color: white; background: black;}
#toolbar a:link {color: inherit;}
```

This also leads to the result shown in Figure 4-7, because the value of `color` is explicitly inherited thanks to an assigned rule whose selector has specificity.

The Cascade

Throughout this chapter, we've skirted one rather important issue: what happens when two rules of equal specificity apply to the same element? How does the browser resolve the conflict? For example, consider the following rules:

```
h1 {color: red;}
h1 {color: blue;}
```

Which one wins? Both have a specificity of `0,0,1`, so they have equal weight and should both apply. That can't be the case because the element can't be both red and blue. So which will it be?

At last, the name "Cascading Style Sheets" comes into focus: CSS is based on a method of causing styles to *cascade* together, which is made possible by combining inheritance and specificity with a few rules. The cascade rules for CSS are:

1. Find all rules containing a selector that matches a given element.

2. Sort all declarations applying to the given element by *explicit weight*.

3. Sort all declarations applying to the given element by *origin*. There are three basic origins: author, reader, and user agent. Under normal circumstances, the author's styles win out over the reader's

styles, and both author and reader styles override the user agent's default styles. This is reversed for rules marked `!important`, where user agent styles override author styles, and both ovedrride reader styles.

4. Sort all declarations applying to the given element by *encapsulation context*. If a style is assigned via a shadow DOM (Document Object Model), for example, it has an encapsulation context for all elements within that same shadow DOM, and does not apply to elements outside that shadow DOM. This allows encapsulated styles to override styles that are inherited from outside the shadow DOM.

5. Sort all declarations by whether or not they are *element-attached*. Styles assigned via a `style` attribute are element-attached. Styles assigned from a stylesheet, whether external or embedded, are not.

6. Sort all declarations by *cascade layer*. For normal-weight styles, the later a cascade layer first appears in the CSS, the greater the precedence. Styles without a layer are considered to be part of a "default" pseudo-layer, one which has higher precedence than styles in explicitly-created layers. For important-weight styles, the *earlier* a cascade layer appears in the CSS, the higher the weight, and all important-weight styles in explicitly-created layers win out over styles in the default layer, important or otherwise. Cascade layers can appear in any origin.

7. Sort all declarations applying to the given element by *specificity*. Those elements with a higher specificity have more weight than those with lower specificity.

8. Sort all declarations applying to the given element by *order of appearance*. The later a declaration appears in the style sheet or document, the more weight it is given. Declarations that appear in an imported style sheet are considered to come before all declarations within the style sheet that imports them.

To be clear about how this all works, let's consider some examples that illustrate the some of the cascade rules.

Sorting by Importance and Origin

If two rules apply to an element, and one is marked `!important`, the important rule wins out:

```
p {color: gray !important;}

<p style="color: black;">Well, <em>hello</em> there!</p>
```

Despite the fact that there is a color assigned in the `style` attribute of the paragraph, the `!important` rule wins out, and the paragraph is gray. This occurs because sorting by `!important` has higher precedence that sorting by element-attached styles (`style=""`). The gray is inherited by the `em` element as well.

Note that if an `!important` is added to the inline style in this situation, then *it* will be the winner. Thus, given the following, the paragraph (and its descendant element) will be black:

```
p {color: gray !important;}

<p style="color: black !important;">Well, <em>hello</em> there!</p>
```

In situations where the importance is the same, the origin of a rule is considered. If an element is matched by normal styles in both the author's style sheet and the reader's style sheet, then the author's styles are used. For example, assume that the following styles come from the indicated origins:

```
p em {color: black;}    /* author's style sheet */

p em {color: yellow;}   /* reader's style sheet */
```

In this case, emphasized text within paragraphs is colored black, not yellow, because the author styles win out over the reader styles. However, if both rules are marked `!important`, the situation changes:

```
p em {color: black !important;}    /* author's style sheet */

p em {color: yellow !important;}   /* reader's style sheet */
```

Now the emphasized text in paragraphs will be yellow, not black.

As it happens, the user agent's default styles—which are often influenced by the user preferences—are figured into this step. The default style declarations are the least influential of all. Therefore, if an author-defined rule applies to anchors (e.g., declaring them to be `white`), then this rule overrides the user agent's defaults.

To sum up, there are eight basic levels to consider in terms of declaration precedence. In order of most to least precedence, these are:

1. Transition declarations (see XREF HERE)
2. User agent important declarations
3. Reader important declarations
4. Author important declarations
5. Animation declarations (see XREF HERE)
6. Author normal declarations
7. Reader normal declarations
8. User agent declarations

Thus, a transition style will override all other rules, regardless of whether those other rules are marked `!important` or from what origin the rules come.

Sorting by Element Attachment

Styles can be attached to an element using a markup attribute such as `style`. These are called *element-*

attached styles, and they are only outweighed by considerations of origin and weight.

To understand this, consider the following rule and markup fragment:

```
h1 {color: red;}

<h1 style="color: green;">The Meadow Party</h1>
```

Given that the rule is applied to the `h1` element, you would still probably expect the text of the `h1` to be green. This happens because every inline declaration is element-attached, and so has a higher weight than styles that aren't element attached, like the `color: red` rule.

This means that even elements with `id` attributes that match a rule will obey the inline style declaration. Let's modify the previous example to include an `id`:

```
h1#meadow {color: red;}

<h1 id="meadow" style="color: green;">The Meadow Party</h1>
```

Thanks to the inline declaration's weight, the text of the `h1` element will still be green.

Just remember that inline styles are generally a bad practice, so try not to use them if at all possible.

Sorting by Cascade Layer

Cascade layers allow authors to group styles together so that they share a precedence level within the cascade. This might sound like `!important`, and in some ways they are similar — but in others, very different. This is easier to demonstrate than it is to describe. The ability to create cascade layers means authors can balance various needs, such as the needs of a component library, against the needs of a specific page or part of a web app.

> **NOTE**
>
> Cascade layers were introduced to CSS at the end of 2021, so browser support for them will only exist in browsers released from that point forward.

If conflicting declarations apply to an element and they all have the same explicit weight and origin, and none are element-attached, they are next sorted by cascade layer. The order of precedence for layers is set by the order in which the layers are first declared or used, with later declared layers taking precedence over earlier declared layers for normal styles. Thus, given the following:

```
@layer site {
     h1 {color: red;}
}
@layer page {
     h1 {color: blue;}
}
```

…then `h1` elements will be colored blue. This is because the `page` layer comes later in the CSS than the `site` layer, and so has higher precedence.)

Any style not part of a named cascade layer is assigned to an implicit "default" layer, one which has higher precedence than any named layer for non-important rules. Suppose we alter the previous example as follows:

```
h1 {color: maroon;}
@layer site {
     h1 {color: red;}
}
@layer page {
     h1 {color: blue;}
}
```

`h1` elements will now be maroon-colored, because the implicit "default" layer to which the `h1 {color: maroon;}` belongs has higher precedence than any named layer.

It is also possible to define a specific precedence order for named cascade layers. Consider the following CSS:

```
@layer site, page;

@layer page {
   h1 {color: blue;}
}

@layer site {
   h1 {color: red;}
}
```

Here, the first line defines an order of precedence for the layers: the `page` layer will be given higher precedence than `site` layer for normal-weight rules like those shown in the example. Thus, in this case, `h1` elements will be blue, because when the layers were sorted, `page` was given more precedence than `site`. For important-flagged rules, the order of precedence is reversed. Thus, if both rules were marked `!important`, the precedence would flip: in that case, `h1` elements would be red.

Let's talk a little bit more about how cascade layers specifically work, especially since they're so new to CSS. Let's say you want to define three layers: one for the basic site styles, one for individual page styles, one for a component library whose styles are imported from an external stylesheet. The CSS might look like this:

```
@layer site, page;
@import url(/assets/css/components.css) layer(components);
```

This ordering will have normal-weight `components` styles override `page` and `site` normal-weight styles, whereas normal-weight `page` styles will only override `site` normal-weight styles. Conversely, important `site` styles will override all `page` and `components` styles, whether they're important or normal-weight, and `page` important styles will override all `components` styles.

Here’s a small example of how layers might be managed.

```
@layer site, component, page;
@import url(/c/lib/core.css) layer(component);
@import url(/c/lib/widgets.css) layer(component);
@import url(/c/site.css) layer(site);

@layer page {
   h1 {color: maroon;}
   p {margin-top: 0;}
}

@layer site {
   body {font-size: 1.1rem;}
   h1 {color: orange;}
   p {margin-top: 0.5em;}
}

p {margin-top: 1em;}
```

In this example, there are three imported stylesheets, one of which is assigned to the `site` layer and two of which are in the `component` layer. Then there are some rules assigned to the `page` layer, and a couple of rules placed in the `site` layer. The rules in the `@layer site {}` block will be combined with the rules from `/c/site.css` into a single `site` layer.

After that, there’s a rule outside the explicit cascade layers, which means it’s part of the implicit “default” layer. Rules in this default layer will override the styles of any of the other layers. So, given the code shown, paragraphs will have top margins of `1em`.

But before all of that, there’s a directive that sets the precedence order of the named layers: `page` overrules `component` and `site`, and `component` overrules `site`. Here’s how those various rules are grouped as far as the cascade is concerned, with comments to describe their placement in the sorting:

```
/* 'site' layer is the lowest weighted */
@import url(/c/site.css) layer(site);
@layer site {
   body {font-size: 1.1rem;}
   h1 {color: orange;}
   p {margin-top: 0.5em;}
}

/* 'component' layer is the next-lowest weighted */
@import url(/c/lib/core.css) layer(component);
@import url(/c/lib/widgets.css) layer(component);

/* 'page' layer is the next-highest weighted */
@layer page {
   h1 {color: maroon;}
   p {margin-top: 0;}
}

/* the implicit layer is the highest-weighted */
p {margin-top: 1em;}
```

As you can see, the later a layer comes in the ordering of the layers, the more weight it's given by the cascade's sorting algorithm.

Cascade layers don't have to be named, to be clear. It just keeps things a lot more clear in terms of setting an order for them. Here are some examples of using un-named cascade layers:

```
@import url(base.css) layer;

p {margin-top: 1em;}

@layer {
   h1 {color: maroon;}
   body p {margin-top: 0;}
}
```

In this case, the rules imported from `base.css` are assigned to an un-named layer. Even though it doesn't actually have a name, let's think of it as "CL1". Then there's a rule outside the layers, setting paragraph top margins to be `1em`. Finally, there's an un-named layer block with a couple of rules; let's think of it as "CL2".

So now we have rules in three layers: "CL1", "CL2", and the implicit layer. And that's the order they're considered in, so in the case of any conflicting normal rules, the rules in the implicit default layer (which comes last in the ordering) will win over conflicting rules in the other two layers, and rules in "CL2" will win over conflicting rules in "CL1".

At least, that's the case for normal-weight rules. For `!important` rules, the order of precedence is flipped, so those in "CL1" will win over conflicting important rules in the other two layers, and important rules in "CL2" win over conflicting important rules in the implicit layer. Strange but true!

This sorting-by-order will come up again in just a little bit, but first, let's bring specificity into the cascade.

Sorting by Specificity

If conflicting declarations apply to an element and they all have the same explicit weight, origin, element attachment (or lack thereof), and cascade layer, they are then sorted by specificity, with the most specific declaration winning out, like this:

```
@layer page {
  p#bright#bright#bright {color: grey;}
}
p#bright {color: silver;}
p {color: black;}

<p id="bright">Well, hello there!</p>
```

Given the rules shown, the text of the paragraph will be silver, as illustrated in Figure 4-8. Why? Because the specificity of `p#bright (`1,0,1`) overrode the specificity of `p` (`0,0,1`), even though the latter rule comes later in the style sheet. The styles from the `page` layer, even though they have the strongest

selector `(3,0,1)` aren't even compared. Only the declarations from the layer with precedence are in contention.

Well, hello there!

Figure 4-8. Higher specificity wins out over lower specificity

Remember that this rule only applies if the rules are part of the same cascade layer. If not, specificity doesn't matter: a `0,0,1` selector in the implicit layer will win over any non-important rule in an explicitly-created cascade layer, no matter how high the latter's specificity gets.

Sorting by Order

Finally, if two rules have exactly the same explicit weight, origin, element attachment, cascade layer, and specificity, then the one that appears later in the style sheet wins out, similar to how cascade layers are sorted in order so that later layers win over earlier layers.

Let's return to an earlier example, where we find the following two rules in the document's style sheet:

```
body h1 {color: red;}
html h1 {color: blue;}
```

In this case, the value of `color` for all `h1` elements in the document will be `blue`, not `red`. This is because the two rules are tied with each other in terms of explicit weight and origin, are in the same cascade layer, and the selectors have equal specificity, so the last one declared is the winner. Note that it doesn't matter how close together the elements are in the document tree; even though `body` and `h1` are closer together than `html` and `h1`, the later one wins. The only thing that matters (when the origin, cascade layer, layer, and specificity are the same) is the order in which the rules appear in the CSS.

So what happens if rules from completely separate style sheets conflict? For example, suppose the following:

```
@import url(basic.css);
h1 {color: blue;}
```

What if `h1 {color: red;}` appears in `basic.css`? In this case, since there are no cascade layers in play, the entire contents of `basic.css` are treated as if they were pasted into the style sheet at the point where the `@import` occurs. Thus, any rule contained in the document's style sheet occurs later than those from the `@import`. If they tie in terms of explicit weight and specificity, the document's style sheet contains the winner. Consider the following:

```
p em {color: purple;}  /* from imported style sheet */

p em {color: gray;}    /* rule contained within the document */
```

In this case, the second rule shown wins out over the imported rule because it was the last one specified, and both are in the implicit cascade layer.

Order sorting is the reason behind the often-recommended ordering of link styles. The recommendation is that you write your link styles in the order link-visited-focus-hover-active, or LVFHA, like this:

```
a:link {color: blue;}
a:visited {color: purple;}
a:focus {color: green;}
a:hover {color: red;}
a:active {color: orange;}
```

Thanks to the information in this chapter, you now know that the specificity of all of these selectors is the same: `0,1,1`. Because they all have the same explicit weight, origin, and specificity, the last one that matches an element will win out. An unvisited link that is being "clicked" or otherwise activated, such as via the keyboard, is matched by four of the rules—`:link`, `:focus`, `:hover`, and `:active`—so the last one of those four will win out. Given the LVFHA ordering, `:active` will win, which is likely what the author intended.

Assume for a moment that you decide to ignore the common ordering and alphabetize your link styles instead. This would yield:

```
a:active {color: orange;}
a:focus {color: green;}
a:hover {color: red;}
a:link {color: blue;}
a:visited {color: purple;}
```

Given this ordering, no link would ever show `:hover`, `:focus`, or `:active` styles because the `:link` and `:visited` rules come after the other three. Every link must be either visited or unvisited, so those styles will always override the others.

Let's consider a variation on the LVFHA order that an author might want to use. In this ordering, only unvisited links will get a hover style; visited links do not. Both visited and unvisited links will get an active style:

```
a:link {color: blue;}
a:hover {color: red;}
a:visited {color: purple;}
a:focus {color: green;}
a:active {color: orange;}
```

Such conflicts arise only when all the states attempt to set the same property. If each state's styles address a different property, then the order does not matter. In the following case, the link styles could be given in any order and would still function as intended:

```
a:link {font-weight: bold;}
a:visited {font-style: italic;}
a:focus {color: green;}
a:hover {color: red;}
a:active {background: yellow;}
```

You may also have realized that the order of the `:link` and `:visited` styles doesn't matter. You could

order the styles LVFHA or VLFHA with no ill effect.

The ability to chain pseudo-classes together eliminates all these worries. The following could be listed in any order without any overrides, as the specificity of the latter two is greater than that of the first two:

```
a:link {color: blue;}
a:visited {color: purple;}
a:link:hover {color: red;}
a:visited:hover {color: gray;}
```

Because each rule applies to a unique set of link states, they do not conflict. Therefore, changing their order will not change the styling of the document. The last two rules do have the same specificity, but that doesn't matter. A hovered unvisited link will not be matched by the rule regarding hovered visited links, and vice versa. If we were to add active-state styles, then order would start to matter again. Consider:

```
a:link {color: blue;}
a:visited {color: purple;}
a:link:hover {color: red;}
a:visited:hover {color: gray;}
a:link:active {color: orange;}
a:visited:active {color: silver;}
```

If the active styles were moved before the hover styles, they would be ignored. Again, this would happen due to specificity conflicts. The conflicts could be avoided by adding more pseudo-classes to the chains, like this:

```
a:link:hover:active {color: orange;}
a:visited:hover:active {color: silver;}
```

This does have the effect of raising the specificity of the selectors—both have a specificity value of `0,3,1`—but they don't conflict because the actual selection states are mutually exclusive. A link can't be both a visited hovered active link *and* an unvisited hovered active link: only one of the two rules will match.

Non-CSS Presentational Hints

It is possible that a document will contain presentational hints that are not CSS—for example, the deprecated `font` element, or the still-very-much-used `height`, `width`, and `hidden` attributes. Such presentational hints will be overridden by any author or reader styles, but not by the user agent's styles. In modern browsers, presentational hints from outside CSS are treated as if they belong to the user agent's stylesheet.

Summary

Perhaps the most fundamental aspect of Cascading Style Sheets is the cascade itself—the process by which conflicting declarations are sorted out and from which the final document presentation is determined. Integral to this process is the specificity of selectors and their associated declarations, and

the mechanism of inheritance.

Chapter 5. Values and Units

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at rfernando@oreilly.com.

In this chapter, we'll tackle features that are the basis for almost everything you can do with CSS: the units that affect the colors, distances, and sizes of a whole host of properties, as well as the units that help to define those values. Without units, you couldn't declare that an image should have 10 pixels of blank space around it, or that a heading's text should be a certain size. By understanding the concepts put forth here, you'll be able to learn and use the rest of CSS much more quickly.

Keywords, Strings, and Other Text Values

Everything in a stylesheet is text, but there are certain value types that directly represent strings of text as opposed to, say, numbers or colors. Included in this category are URLs and, interestingly enough, images.

Keywords

For those times when a value needs to be described with a word of some kind, there are *keywords*. A very common example is the keyword `none`, which is distinct from `0` (zero). Thus, to remove the underline from links in an HTML document, you would write:

```
a[href] {text-decoration: none;}
```

Similarly, if you want to force underlines on the links, then you would use the keyword `underline` instead of `none`.

If a property accepts keywords, then its keywords will be defined only for the scope of that property. If two properties use the same word as a keyword, the behavior of the keyword for one property will not necessarily be shared with the other. As an example, `normal`, as defined for `letter-spacing`, means something very different than the `normal` defined for `font-style`.

Global keywords

CSS3 defines four “global” keywords that are accepted by every property in the specification: `inherit`, `initial`, `unset`, and `revert`.

inherit

The keyword `inherit` makes the value of a property on an element the same as the value of that property on its parent element. In other words, it forces inheritance to occur even in situations where it would not normally operate. In many cases, you don’t need to specify inheritance, since many properties inherit naturally. Nevertheless, `inherit` can still be very useful.

For example, consider the following styles and markup:

```
#toolbar {background: blue; color: white;}

<div id="toolbar">
<a href="one.html">One</a> | <a href="two.html">Two</a> |
<a href="three.html">Three</a>
</div>
```

The `div` itself will have a blue background and a white foreground, but the links will be styled according to the browser’s preference settings. They’ll most likely end up as blue text on a blue background, with white vertical bars between them.

You could write a rule that explicitly sets the links in the “toolbar” to be white, but you can make things a little more robust by using `inherit`. You just add the following rule to the stylesheet:

```
#toolbar a {color: inherit;}
```

This will cause the links to use the inherited value of `color` in place of the user agent’s default styles.

Ordinarily, directly assigned styles override inherited styles, but `inherit` can undo that behavior. It might not always be a good idea—for example, here links might blend into surrounding text too much, and become a usability and accessibility concern—but it can be done.

Similarly, you can pull a property value down from a parent even if it wouldn’t happen normally. Take `border`, for example, which is (rightfully) not inherited. If you want a `span` to inherit the border of its parent, all you need is `span {border: inherit;}`. More likely, though, you just want the border on a `span` to use the same border color as its parent. In that case `span {border-color: inherit;}` will do the trick.

initial

The keyword `initial` sets the value of a property to the defined initial value, which in a way means it “resets” the value. For example, the default value of `font-weight` is `normal`. Thus, declaring `font-weight: initial` is the same as declaring `font-weight: normal`.

This might seem a little bit silly until you consider that not all values have explicitly defined initial values. For example, the initial value for `color` is “depends on user agent.” That’s not a funky keyword you should type! What it means is that the default value of `color` depends on things like the preferences

settings in a browser. While almost nobody changes the default text color setting from black, someone might set it to a dark gray or even a bright red. By declaring `color: initial;`, you're telling the browser to set the color of the element to whatever the user's default color is set to be.

Another benefit of `initial` is that you can set a property back to its initial value without having to know what that initial value actually was. This can be especially useful when resetting a lot of properties all at once, either via JavaScript or CSS.

unset

The keyword `unset` acts as a universal stand-in for both `inherit` and `initial`. If the property is inherited, then `unset` has the same effect as if `inherit` was used. If the property is *not* inherited, then `unset` has the same effect as if `initial` was used. This makes it very useful for resetting a property by canceling out any other styles that might be applied to it.

revert

The keyword `revert` sets the value of a property to the value the property would have had if no changes had been made by the current style origin. In effect, `revert` lets you say, "All property values for this element should be as if the author styles don't exist, but user agent and user styles do exist."

Thus, given the following basic example, `p` elements will be rendered as gray text with a transparent background.

```
p {background: lime; color: gray;}
p {background: revert;}
```

This does mean that any property whose value is inherited will be given the same value as that of their parent. `revert` is useful for cases where you have a bunch of site-wide styles applying to an element, and you want to strip them all away so as to apply a set of one-off styles to just that element. Rather than have to override all those properties, you can revert them to defaults — and you can do it with a single property, which is the topic of the next section.

revert-layer

If you're using Cascade Layers (see "Sorting by Cascade Layer") and want to "undo" whatever styles might be applied by the current layer, the `revert-layer` value is here to help. The difference here is that `revert-layer` effectivelty means, "All property values for this element should be as if the author styles *in the current Cascade Layer* don't exist, but other author Cascade Layers (inluding the default), user agent, and user styles do exist."

Thus, given the following, paragraphs with a `class` containing the word `example` will be rendered as red text on a yellow background:

```
@layer site, system;

p {color: red;}
@layer system {
        p {background: yellow; color: fuchsia;}
}
```

```
@layer site {
        p {background: lime; color: gray;}
        p.example {background: revert; color: revert;}
}
```

For the background, the browser looks at the assigned values in previous Cascade Layers and picks the one with the highest weight. Only one layer (`system`) set a background color, so that's what's used instead of `lime`. The same is done for the foreground color, and since there is a color assigned in the default layer, and the default layer overrides all explicitly-created layers, `red` is used instead of `gray`.

> **NOTE**
>
> As of late 2022, only Firefox supported `revert-layer`, but we anticipate it being widely supported in the near future.

The `all` property

These global values are usable on all properties, but there is a special property that *only* accepts the global keywords: `all`.

ALL

Values	`inherit` | `initial` | `unset` | `revert`
Initial value	See individual properties

`all` is a stand-in for all properties *except* `direction`, `unicode-bidi`, and any custom properties (see "Custom Properties"). Thus, if you declare `all: inherit` on an element, you're saying that you want all properties except `direction`, `unicode-bidi`, and custom properties to inherit their values from the element's parent. Consider the following:

```
section {color: white; background: black; font-weight: bold;}
#example {all: inherit;}

<section>
    <div id="example">This is a div.</div>
</section>
```

You might think this causes the `div` element to inherit the values of `color`, `background`, and `font-weight` from the `section` element. And it does do that, yes—but it will *also* force inheritance of the values of *every single other property in CSS* (minus the two exceptions) from the `section` element.

Maybe that's what you want, in which case, great. But if you just want to inherit the property values you wrote out for the `section` element, then the CSS would need to look more like this:

```
section {color: white; background: black; font-weight: bold;}
#example {color: inherit; background: inherit; font-weight: inherit;}
```

Odds are what you really want in these situations is `all: unset`, but your stylesheet may vary.

Strings

A *string value* is an arbitrary sequence of characters wrapped in either single or double quotes, and is represented in value definitions with <*string*>. Two simple examples:

```
"I like to play with strings."
'Strings are fun to play with.'
```

Note that the quotes balance, which is to say that you always start and end with the same kind of quotes. Getting this wrong can lead to all kinds of parsing problems, since starting with one kind of quote and trying to end with the other means the string won't actually be terminated. You could accidentally incorporate subsequent rules into the string that way!

If you want to put quote marks inside strings, that's OK, as long as they're either not the kind you used to enclose the string or are escaped using a backslash:

```
"I've always liked to play with strings."
'He said to me, "I like to play with strings."'
"It's been said that \"haste makes waste.\""
'There\'s never been a "string theory" that I\'ve liked.'
```

Note that the only acceptable string delimiters are `'` and `"`, sometimes called "straight quotes." That means you can't use "curly" or "smart" quotes to begin or end a string value. You can use them inside a string value, as in this code example, though, and they don't have to be escaped:

```
"It’s been said that “haste makes waste.”"
'There’s never been a “string theory” that I’ve liked.'
```

This requires that you use Unicode encoding for your documents, but you should be doing that regardless. (You can find the Unicode standard at *http://www.unicode.org/standard/standard.html*.)

If you have some reason to include a newline in your string value, you can do that by escaping the newline itself. CSS will then remove it, making things as if it had never been there. Thus, the following two string values are identical from a CSS point of view:

```
"This is the right place \
for a newline."
"This is the right place for a newline."
```

If, on the other hand, you actually want a string value that includes a newline character, then use the Unicode reference `\A` where you want the newline to occur:

```
"This is a better place \Afor a newline."
```

Identifiers

One word, case-sensitive strings that should not be quoted are known as *identifiers*, represented in the CSS syntax as *<ident>* or *<custom-ident>*, depending on the specification and context. Identifiers are used as animation names, gridline names, and counter names, among others. There is also *<dashed-ident>*, which is used for custom properties.

Rules for creating a custom identifier include not starting the word with a number, a double hyphen, or a single hyphen followed by a number. Other than that, really any character is valid, including emojis, but if you use certain characters, including a space or a backslash, you need to escape them with a backslash.

Identifiers themselves are words, and are case-sensitive; thus, `myID` and `MyID` are, as far as CSS is concerned, completely distinct and unrelated to each other. In cases where a property accepts both an identifier and one or more keywords, the author should take care to never define an identifier identical to a valid keyword, including the global keywords `initial`, `inherit`, `unset`, and `revert`. `none` is also a really bad idea, as is `running` in cases where you're naming an animation.

URLs

If you've written web pages, you're almost certainly familiar with URLs (Uniform Resource Locators). Whenever you need to refer to one—as in the `@import` statement, which is used when importing an external stylesheet—the general format is:

```
url(protocol://server/pathname/filename)
url("<string>")   /* can use single or double quotes. */
```

This example defines what is known as an *absolute URL*. By absolute, we mean a URL that will work no matter where (or rather, in what page) it's found, because it defines an absolute location in web space. Let's say that you have a server called *web.waffles.org*. On that server, there is a directory called *pix*, and in this directory is an image *waffle22.gif*. In this case, the absolute URL of that image would be:

```
https://web.waffles.org/pix/waffle22.gif
```

This URL is valid no matter where it is written, whether the page containing it is located on the server *web.waffles.org* or *web.pancakes.com*.

The other type of URL is a *relative URL*, so named because it specifies a location that is relative to the document that uses it. If you're referring to a relative location, such as a file in the same directory as your web page, then the general format is:

```
url(pathname)
url("<string>")   /* can use single or double quotes. */
```

This works only if the image is on the same server as the page that contains the URL. For argument's sake, assume that you have a web page located at *http://web.waffles.org/syrup.html* and that you want the image *waffle22.gif* to appear on this page. In that case, the URL would be:

```
pix/waffle22.gif
```

This path works because the web browser knows it should start with the place it found the web document and then add the relative URL. In this case, the pathname *pix/waffle22.gif* added to the server name *http://web.waffles.org* equals *http://web.waffles.org/pix/waffle22.gif*. You can almost always use an absolute URL in place of a relative URL; it doesn't matter which you use, as long as it defines a valid location.

In CSS, relative URLs are relative to the stylesheet itself, not to the HTML document that uses the stylesheet. For example, you may have an external stylesheet that imports another stylesheet. If you use a relative URL to import the second stylesheet, it must be relative to the first stylesheet. In fact, if you have a URL in any imported stylesheet, it needs to be relative to the imported stylesheet.

As an example, consider an HTML document at *http://web.waffles.org/toppings/tips.html*, which has a `link` to the stylesheet *http://web.waffles.org/styles/basic.css*:

```
<link rel="stylesheet" type="text/css"
    href="http://web.waffles.org/styles/basic.css">
```

Inside the file *basic.css* is an `@import` statement referring to another stylesheet:

```
@import url(special/toppings.css);
```

This `@import` will cause the browser to look for the stylesheet at *http://web.waffles.org/styles/special/toppings.css*, not at *http://web.waffles.org/toppings/special/toppings.css*. If you have a stylesheet at the latter location, then the `@import` in *basic.css* should read one of the two following ways:

```
@import url(http://web.waffles.org/toppings/special/toppings.css);

@import url("../special/toppings.css");
```

Note that there cannot be a space between the `url` and the opening parenthesis:

```
body {background: url(http://www.pix.web/picture1.jpg);}   /* correct */
body {background: url  (images/picture2.jpg);}          /* INCORRECT */
```

If the space is present, the entire declaration will be invalidated and thus ignored.

> **NOTE**
>
> As of this writing in late 2022, the CSS Working Group is planning to introduce a new function called `src()`, which will only accept strings and not unquoted URLs. This is meant to allow custom properties to be used inside `src()`, which will let authors define which file should be loaded based on the value of a custom property.

Images

An *image value* is a reference to an image, as you might have guessed. Its syntax representation is *<image>*.

At the most basic level of support, which is to say the one every CSS engine on the planet would understand, an *<image>* value is a *<url>* value. In more modern user agents, *<image>* stands for one of the following:

<url>

A URL identifier of an external resource; in this case, the URL of an image.

<gradient>

Refers to either a linear, radial, or conic gradient image, either singly or in a repeating pattern. Gradients are fairly complex, and are covered in detail in Chapter 8.

<image-set>

A set of images, chosen based on a set of conditions embedded into the value, which is defined as `image-set()` but is more widely recognized with the `-webkit-` prefix. For example, `-webkit-image-set()` could specify that a larger image be used for desktop layouts, whereas a smaller image (both in pixel size and file size) be used for a mobile design. It is intended to at least approximate the behavior of the `srcset` attribute for `picture` elements. As of late 2022, `-webkit-image-set` was basically universally supported, with most browsers other than Safari also accepting `image-set()` (without the prefix).

<cross-fade>

Used to blend two (or more) images together, with a specific transparency given to each image. Use cases include blending two images together, blending an image with a gradient, and so on. As of early 2022, this was supported as `-webkit-cross-fade()` in Blink- and WebKit-based browsers, and not supported at all in the Firefox family, with or without the prefix.

There are also the `image()` and `element()` functions, but as of late 2022, neither is supported by any browser, except for a vendor-prefixed version of `element()` supported by Firefox 57 and later. Finally, there is `paint()` which refers to an image painted by CSS Houdini’s PaintWorklet. As of late 2022, this is only supported in a basic form by Blink-based browsers like Chrome.

Numbers and Percentages

These value types serve as the foundation for many other values types. For example, font sizes can be defined using the `em` unit (covered later in this chapter) preceded by a number. But what kind of number? Understanding the types of numbers here lets us be clear what we mean when defining other value types later on.

Integers

An *integer value* is about as simple as it gets: one or more numbers, optionally prefixed by a `+` or `−` (plus or minus) sign to indicate a positive or negative value. That's it. Integer values are represented in value syntax as <*integer*>. Examples include `13`, `−42`, `712`, and `1066`.

Some properties define a range of acceptable integer values. Integer values that fall outside a defined range are, by default, considered invalid and cause the entire declaration to be ignored. However, some properties define behavior that causes values outside the accepted range to be set to the accepted value closest to the declared value, known as *clamping*.

In cases (such as the property `z-index`) where there is no restricted range, user agents must support values up to ±1,073,741,824 ($\pm 2^{30}$).

Numbers

A *number value* is either an <*integer*> or a real number, which is to say an integer followed by a dot and then some number of following integers. Additionally, it can be prefixed by either `+` or `−` to indicate positive or negative values. Number values are represented in value syntax as <*number*>. Examples include 5, 2.7183, −3.1416, 6.2832, and 1.0218e29 (scientific notation).

The reason a <*number*> can be an <*integer*> and yet there are separate value types is that some properties will only accept integers (e.g., `z-index`), whereas others will accept any real number (e.g., `flex-grow`).

As with integer values, number values may have limits imposed on them by a property definition; for example, `opacity` restricts its value to be any valid <*number*> in the range `0` to `1`, inclusive. Some properties define behavior that causes values outside the accepted range to be clamped to an acceptable value closest to the declared value; e.g., `opacity: 1.7` would be clamped to `opacity: 1`. For those that do not, number values that fall outside a defined range are considered invalid and cause the entire declaration to be ignored.

Percentages

A *percentage value* is a <*number*> followed by a percentage sign (`%`), and is represented in value syntax as <*percentage*>. Examples would include `50%` and `33.333%`. Percentage values are always relative to another value, which can be anything—the value of another property of the same element, a value inherited from the parent element, or a value of an ancestor element. Properties that accept percentage values will define any restrictions on the range of allowed percentage values, as well as the way in which the percentage is relatively calculated.

Fractions

A *fraction value* (or *flexible ratio*) is a <*number*> followed by the `fr` unit label. Thus, one fractional unit is `1fr`. The `fr` unit represents a fraction of the leftover space, if any, in a grid container.

As with all CSS dimensions, there is no space between the unit and the number. Fraction values are not lengths (nor are they compatible with <length>s, unlike some <percentage> values), so they cannot be

used with other unit types in `calc()` functions.

> **NOTE**
>
> Fraction values are mostly used in Grid layout (see XREF HERE), but there are plans to use it in more contexts, such as the planned (as of late 2022) `stripes()` function.

Distances

Many CSS properties, such as margins, depend on length measurements to properly display various page elements. It's likely no surprise, then, that there are a number of ways to measure length in CSS.

All length units can be expressed as either positive or negative numbers followed by a label, although note that some properties will accept only positive numbers. You can also use real numbers—that is, numbers with decimal fractions, such as 10.5 or 4.561.

All length units are followed by short abbreviation that represents the actual unit of length being specified, such as `in` (inches) or `pt` (points). The only exception to this rule is a length of `0` (zero), which need not be followed by a unit when describing lengths.

These length units are divided into two types: *absolute length units* and *relative length units*.

Absolute Length Units

We'll start with absolute units because they're easiest to understand. The six types of absolute units are as follows:

Inches (`in`)

: As you might expect, this notation refers to the inches you'd find on a ruler in the United States. (The fact that this unit is in the specification, even though almost the entire world uses the metric system, is an interesting insight into the pervasiveness of US interests on the internet—but let's not get into virtual sociopolitical theory right now.)

Centimeters (`cm`)

: Refers to the centimeters that you'd find on rulers the world over. There are 2.54 centimeters to an inch, and one centimeter equals 0.394 inches.

Millimeters (`mm`)

: For those Americans who are metric-challenged, there are 10 millimeters to a centimeter, so an inch equals 25.4 millimeters, and a millimeter equals 0.0394 inches.

Quarter-millimeters (`Q`)

: There are 40 Q units in a centimeter; thus, setting an element to be 1/10 of a centimeter wide—which is also to say, a millimeter wide—would mean a value of `4Q`.

Points (`pt`)

Points are standard typographical measurements that have been used by printers and typesetters for decades and by word processing programs for many years. Traditionally, there are 72 points to an inch. Therefore the capital letters of text set to 12 points should be one-sixth of an inch tall. For example, `p {font-size: 18pt;}` is equivalent to `p {font-size: 0.25in;}`.

Picas (`pc`)

Pica is another typographical term. A pica is equivalent to 12 points, which means there are 6 picas to an inch. As just shown, the capital letters of text set to 1 pica should be one-sixth of an inch tall. For example, `p {font-size: 1.5pc;}` would set text to the same size as the example declarations found in the definition of points.

Pixels (`px`)

A pixel is a small box on screen, but CSS defines pixels more abstractly. In CSS terms, a pixel is defined to be the size required to yield 96 pixels per inch. Many user agents ignore this definition in favor of simply addressing the pixels on the screen. Scaling factors are brought into play when page zooming or printing, where an element `100px` wide can be rendered more than 100 device dots wide.

These units are only really useful if the browser knows all the details of the screen on which your page is displayed, the printer you're using, or whatever other user agent might apply. On a web browser, display is affected by the size of the screen and the resolution to which the screen is set; there isn't much that you, as the author, can do about these factors. If nothing else, it should be the case that measurements will be consistent in relation to each other—that is, that a setting of `1.0in` will be twice as large as `0.5in`, as shown in Figure 5-1.

[one] This paragraph has a one-"inch" left margin.

[two] This paragraph has a half-"inch" left margin.

Figure 5-1. Setting absolute-length left margins

Let's make the (fairly suspect) assumption that your computer knows enough about its display system to accurately reproduce real-world measurements. In that case, you could make sure every paragraph has a top margin of half an inch by declaring `p {margin-top: 0.5in;}`.

Absolute units are much more useful in defining stylesheets for printed documents, where measuring things in terms of inches, points, and picas is much more common.

Pixel lengths

On the face of things, pixels are straightforward. If you look at a screen closely enough, you can see that it's broken up into a grid of tiny little boxes. Each box is a pixel. If you define an element to be a certain number of pixels tall and wide, as in the following markup:

```
<p>
The following image is 20 pixels tall and wide: <img src="test.gif"
```

```
    style="width: 20px; height: 20px;" alt="" />
</p>
```

then it follows that the element will be that many screen elements tall and wide, as shown in Figure 5-2.

The following image is 20 pixels tall and wide:

Figure 5-2. Using pixel lengths

The problem is, thanks to high-density displays like those found on mobile devices and modern laptops, the individual screen elements aren't treated as pixels any more. Instead, the pixels used in your CSS are translated into something that aligns with human expectations, which is covered in the next section.

Pixel theory

In its discussion of pixels, the CSS specification recommends that, in cases where a display's resolution density is significantly different than 96 pixels per inch (ppi), user agents should scale pixel measurements to a *reference pixel*.

A reference pixel is defined as:

> *…the visual angle of one pixel on a device with a device pixel density of 96dpi 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 0.0213 degrees. For reading at arm's length, 1px thus corresponds to about 0.26 mm (1/96 inch). (https://www.w3.org/TR/css-values-4/#reference-pixel)*

On most modern displays, the actual number of pixels per inch (ppi) is higher than 96—sometimes much higher. The Retina display on an iPhone 13, for example, is physically 326 ppi, and the display on the iPad Pro is physically 264 ppi. As long as a browser on one of those devices sets the reference pixel such that an element set to be `10px` tall appears to be 2.6 millimeters tall on the screen, then the physical display density isn't something you have to worry about, any more than having to worry about the number of dots per inch on a printout.

Resolution Units

There are unit types based on display resolution:

Dots per inch (`dpi`)

The number of display dots per linear inch. This can refer to the dots in a paper printer's output, the physical pixels in an LED screen or other device, or the elements in an e-ink display such as that used by a Kindle.

Dots per centimeter (`dpcm`)

Same as `dpi`, except the linear measure is one centimeter instead of one inch.

Dots per pixel unit (`dppx`)

The number of display dots per CSS `px` unit. As of CSS3, `1dppx` is equivalent to `96dpi` because CSS defines pixel units at that ratio. Just bear in mind that ratio could change in future versions of

CSS.

These units are most often used in the context of media queries. As an example, an author can create a media block to be used only on displays that have higher than 500 dpi:

```
@media (min-resolution: 500dpi) {
    /* rules go here */
}
```

Again, it’s important to remember that CSS pixels are *not* device resolution pixels. Text with `font-size: 16px` will be a relatively consistent size whether the device has 96 dpi or 470 dpi. While a reference pixel is defined to appear to be 1/96 of an inch in size, when a device has has more than 96 dpi, the content will not look smaller. Zooming is created by expanding CSS pixels as much as is needed; an image will appear larger, but the image size doesn’t actually change: rather, the width of the screen, in terms of reference pixels, gets smaller.

Relative Length Units

Relative units are so called because they are measured in relation to other things. The actual (or absolute) distance they measure can change due to factors beyond their control, such as screen resolution, the width of the viewing area, the user’s preference settings, and a whole host of other things. In addition, for some relative units, their size is almost always relative to the element that uses them and will thus change from element to element.

em and ex units

First, let’s consider the character based length units, including `em`, `ex`, and `ch`, which are closely related. There are two other font-relative units, , `cap`, and `ic`, which are not well supported as of early 2022.

The em unit

In CSS, one “em” is defined to be the value of `font-size` for a given font. If the `font-size` of an element is 14 pixels, then for that element, `1em` is equal to 14 pixels.

As you may suspect, this value can change from element to element. For example, let’s say you have an `h1` with a font size of 24 pixels, an `h2` element with a font size of 18 pixels, and a paragraph with a font size of 12 pixels. If you set the left margin of all three at `1em`, they will have left margins of 24 pixels, 18 pixels, and 12 pixels, respectively:

```
h1 {font-size: 24px;}
h2 {font-size: 18px;}
p {font-size: 12px;}
h1, h2, p {margin-left: 1em;}
small {font-size: 0.8em;}

<h1>Left margin = <small>24 pixels</small></h1>
<h2>Left margin = <small>18 pixels</small></h2>
<p>Left margin = <small>12 pixels</small></p>
```

When setting the size of the font, on the other hand, the value of `em` is relative to the font size of the parent element, as illustrated by Figure 5-3.

Figure 5-3. Using em for margins and font sizing

In theory, one em is equal to the width of a lowercase *m* in the font used—that's where the name comes from, in fact. It's an old typographer's term. However, this is not assured in CSS.

The ex unit

`ex` refers to the height of a lowercase *x* in the font being used. Therefore, if you have two paragraphs in which the text is 24 points in size, but each paragraph uses a different font, then the value of `ex` could be different for each paragraph. This is because different fonts have different heights for *x*, as you can see in Figure 5-4. Even though the examples use 24-point text—and therefore each example's `em` value is 24 points—the x-height for each is different.

Figure 5-4. Varying x heights

The ch unit

An interesting addition to CSS3 is the `ch` unit, which is broadly meant to represent “one character.” CSS Values and Units Level 4 defines `ch` as:

> *Equal to the advance measure of the “0” (ZERO, U+0030) glyph found in the font used to render it.*
>
> —https://www.w3.org/TR/css-values-4/#ch

The term *advance measure* is a CSS-ism that corresponds to the term “advance width” in Western typography. CSS uses the term “measure” because some scripts are not right to left or left to right, but instead top to bottom or bottom to top, and so may have an advance height rather than an advance width.

Without getting into too many details, a character glyph’s advance width is the distance from the start of a character glyph to the start of the next. This generally corresponds to the width of the glyph itself plus any built-in spacing to the sides. (Although that built-in spacing can be either positive or negative.)

The easiest way to demonstrate this unit is to run a bunch of zeroes together and then set an image to have a width with the same number of `ch` units as the number of zeroes, as shown in Figure 5-5:

```
img {height: 1em; width: 25ch;}
```

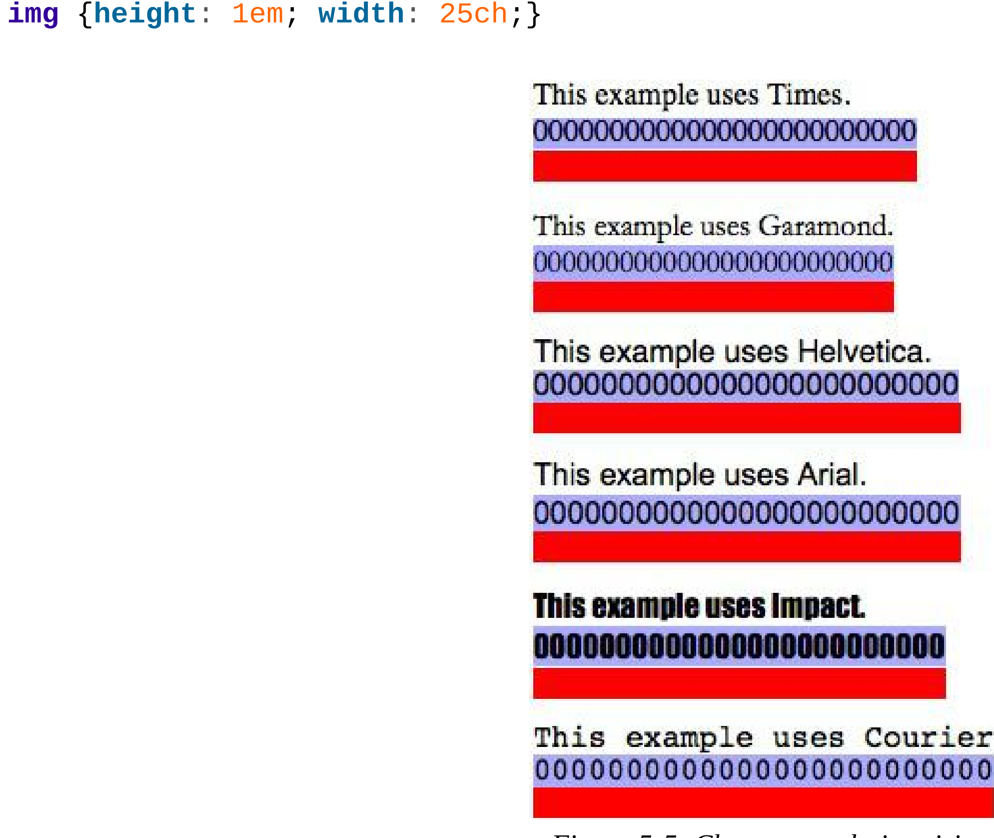


Figure 5-5. Character-relative sizing

Given a monospace font, all characters are by definition `1ch` wide. In any proportional face type, which is what the vast majority of Western typefaces are, characters may be wider or narrower than the “0” and so cannot be assumed to be exactly `1ch` wide.

Other relative length units

There are a few other relative length units to be mentioned:

1. `ic` refers to the advance measure of the “” (CJK water ideograph, U+6C34) glyph found in the first font that can render it. This is like `ch` in that it uses an advance measure, but defines a measure

more useful for ideographic languages than the “0” character. If it can’t be calculated for a given situation, then it’s assumed to be equal to `1em`.

2. `cap` refers to the cap-height which is approximately equal to the height of a capital Latin letter, even in fonts that do not contain Latin letters. If it can’t be calculated for a given situation, then it’s assumed to be equal to the font’s ascent height.

3. `lh` is equal to the computed value of the `line-height` property of the element on which it is used.

As of late 2022, only developer preview builds of Firefox supported `cap`, and preview builds of Chrome supported `lh`.

Root-relative length units

Most of the character-based length units discussed in the previous section have a corresponding *root-relative* value. A root-relative value is one that is calculated with respect to the root element of the document, and thus provide a uniform value no matter what context they’re used in. We’ll discuss the most widely-supported such unit, and then summarize the rest.

The rem unit

The `rem` unit is calculated using the font size of the document’s root element. In HTML, that’s the `html` element. Thus, declaring any element to have `font-size: 1rem;` is setting it to have the same font-size value as the root element of the document.

As an example, consider the following markup fragment. It will have the result shown in Figure 5-6.

```
<p> This paragraph has the same font size as the root element thanks to
    inheritance.</p>
<div style="font-size: 30px; background: silver;">
  <p style="font-size: 1em;">This paragraph has the same font size as its parent
    element.</p>
  <p style="font-size: 1rem;">This paragraph has the same font size as the root
    element.</p>
</div>
```

This paragraph has the same font size as the root element thanks to inheritance.

This paragraph has the same font size as its parent element.

This paragraph has the same font size as the root element.

Figure 5-6. ems versus rems

In effect, `rem` acts as a reset for font size: no matter what relative font sizing has happened to the ancestors of an element, giving it `font-size: 1rem;` will put it right back where the root element is set. This will usually be the user’s default font size, unless you (or the user) have set the root element to a specific font size.

For example, given this declaration, `1rem` will always be equivalent to `13px`:

```
html {font-size: 13px;}
```

However, given *this* declaration, `1rem` will always be equivalent to three-quarters the user's default font size:

```
html {font-size: 75%;}
```

In this case, if the user's default is 16 pixels, then `1rem` will equal `12px`. If the user has actually set their default to 12 pixels—and yes, some people do this—then `1rem` will equal `9px`; if the default setting is 20 pixels, then `1rem` equals `15px`. And so on.

You are not restricted to the value `1rem`. Any real number can be used, just as with the `em` unit, so you can do fun things like set all of your headings to be multiples of the root element's font size:

```
h1 {font-size: 2rem;}
h2 {font-size: 1.75rem;}
h3 {font-size: 1.4rem;}
h4 {font-size: 1.1rem;}
h5 {font-size: 1rem;}
h6 {font-size: 0.8rem;}
```

> **NOTE**
>
> `font-size: 1rem` is equivalent to `font-size: initial` as long as no font size is set for the root element.

Other root-relative units

As mentioned previously, `rem` is not the only root-relative unit defined by CSS. These are summarized in Table 5-1.

Table 5-1. Link pseudo-classes

Length	Root-relative unit	Relative to
`em`	`rem`	Computed `font-size`
`ex`	`rex`	Computed x-height
`ch`	`rch`	Advance measure of the `0` character
`cap`	`rcap`	Height of a Roman capital letter
`ic`	`ric`	Advance measure of the ideograph
`lh`	`rlh`	Computed `line-height`

Of all the root-relative units, only `rem` was supported as of late 2022, but it was supported by essentially all browsers.

Viewport-relative units

Another new addition are the viewport-relative size units. These are calculated with respect to the size of the viewport—browser window, printable area, mobile device display, etc. The six introduced in the late 2010s were:

Viewport width unit (`vw`)

> Equal to the viewport's width divided by 100. Therefore, if the viewport is 937 pixels wide, `1vw` is equal to `9.37px`. If the viewport's width changes, say by dragging the browser window wider or more narrow, the value of `vw` changes along with it.

Viewport height unit (`vh`)

> Equal to the viewport's height divided by 100. Therefore, if the viewport is 650 pixels tall, `1vh` is equal to `6.5px`. If the viewport's height changes, say by dragging the browser window taller or shorter, the value of `vh` changes along with it.

Viewport block unit (`vb`)

> Equal to the size of the viewport along the block axis, divided by 100. The block axis is explained in Chapter 6. In top-to-bottom languages like English or Arabic, `vb` will be equal to `vh` by default.

Viewport inline unit (`vi`)

> Equal to the size of the viewport along the inline axis, divided by 100. The inline axis is explained in Chapter 6. In horizontally written languages like English or Arabic, `vi` will be equal to `vw` by default.

Viewport minimum unit (`vmin`)

> Equal to 1/100 of the viewport's width or height, whichever is *lesser*. Thus, given a viewport that is 937 pixels wide by 650 pixels tall, `1vmin` is equal to `6.5px`.

Viewport maximum unit (`vmax`)

> Equal to 1/100 of the viewport's width or height, whichever is *greater*. Thus, given a viewport that is 937 pixels wide by 650 pixels tall, `1vmax` is equal to `9.37px`.

WARNING

As of late 2022, `vb` and `vi` were not supported in browsers other than Firefox.

Note that these are length units like any other, and so can be used anywhere a length unit is permitted. You can scale the font size of a heading in terms of the viewport, height, for example, with something like `h1 {font-size: 10vh;}`. This sets the font size to be 1/10 the height of the viewport—a technique potentially useful for article titles and the like.

These units can be particularly handy for creating full-viewport interfaces, such as those one would

expect to find on a mobile device, because it can allow elements to be sized compared to the viewport and not any of the elements within the document tree. It's thus very simple to fill up the entire viewport, or at least major portions of it, and not have to worry about the precise dimensions of the actual viewport in any particular case.

Here's a very basic example of viewport-relative sizing, which is illustrated in Figure 5-7:

```
div {width: 50vh; height: 33vw; background: gray;}
```

An interesting (though perhaps not useful) fact about these units is that they aren't bound to their own primary axis. Thus, for example, you can declare `width: 25vh;` to make an element as wide as one-quarter the height of the viewport.

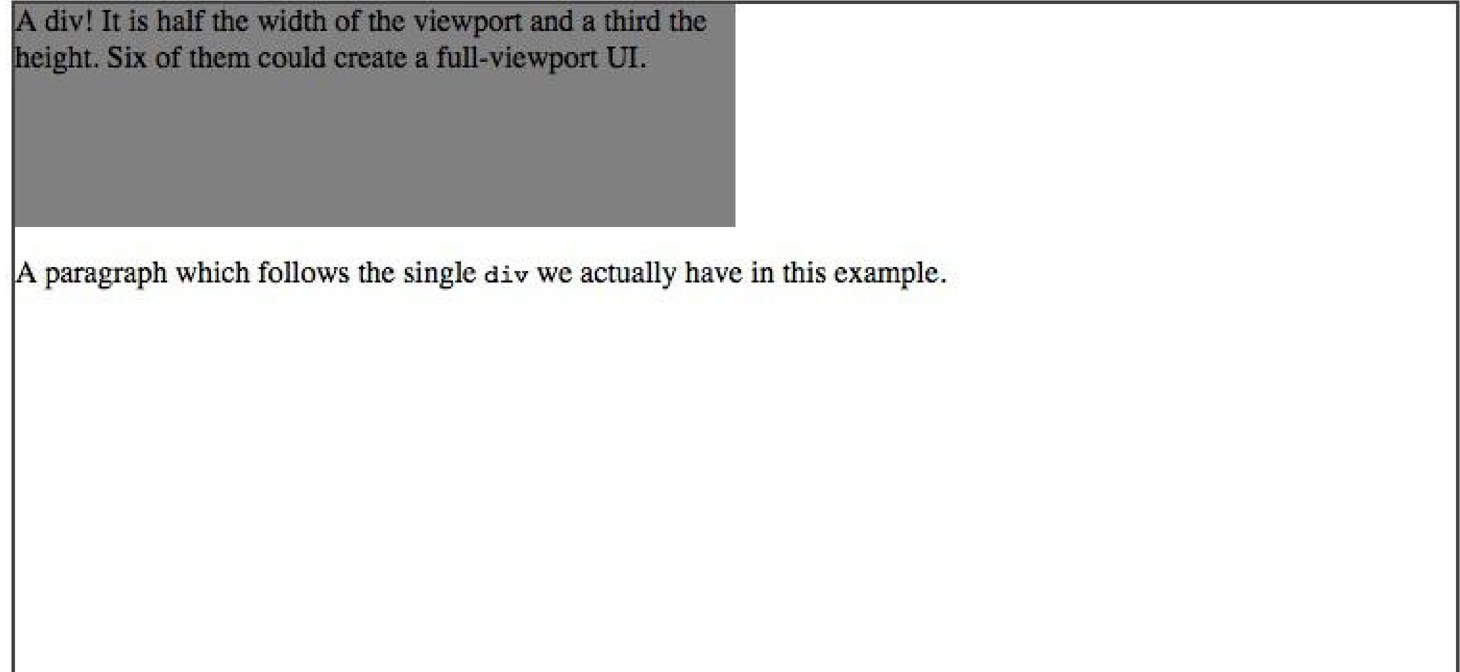


Figure 5-7. Viewport-relative sizing

In 2022, new variants of these units were introduced to accommodate the vagaries of viewports and how they can be sized, particularly on devices where the user interface may expand and contract based on user input. These variants are based on four viewport types:

Default

The default viewport size, as defined by the user agent (browser). This viewport type is expected to correspond to the units `vw`, `vh`, `vb`, `vi`, `vmin`, and `vmax`. The default viewport may correspond to one of the other viewport types; e.g., the default viewport could be the same as the large viewport, but that's up to each browser to decide.

Large

The largest possible viewport after any user-agent interfaces are contracted to their fullest extent. For example, on a mobile device, the browser chrome may be minimized or hidden most of the time so that the maximum screen area can be used to show page content. This is the state described by the large viewport. If you want an element's size to be determined by the full viewport area, even if that

will lead to it being overlapped by user interface, the large-viewport units are the way to go. The units corresponding this viewport type are `lvw`, `lvh`, `lvb`, `lvi`, `lvmin`, and `lvmax`.

Small

The smallest possible viewport after any user-agent interfaces are expanded to their fullest extent. This is the state where the browser's chrome take up as much screen space as it possibly can, leaving a minimum space for the page content. If you want to be sure an element's sizing will take into account any possible interface actions, use these units. The units corresponding this viewport type are `svw`, `svh`, `svb`, `svi`, `svmin`, and `svmax`.

Dynamic

The dynamic viewport is the area in which content is visible, and can change as the user interface expands or contracts. As an example, consider how the browser interface can appear or disappear on mobile devices, depending on how the content is scrolled or where on the screen the user taps. If you want to set lengths based on the size of the viewport at every moment, regardless of how it changes, these are the units for you. The units corresponding this viewport type are `dvw`, `dvh`, `dvb`, `dvi`, `dvmin`, and `dvmax`.

As of late 2022, scrollbars (if any) are ignored for the purposes of calculating all of the previous units. Thus, the calculated size of `svw` or `dvw` will *not* change if scrollbars appear or disappear, or at least shouldn't.

Function values

One of the more recent developments in CSS is an increase in the number of values that are effectively functions. This can range from doing math calculations to clamping value ranges to pulling values out of HTML attributes. There are, in fact, a *lot* of these, listed here:

- `abs()`
- `acos()`
- `annotation()`
- `asin()`
- `atan()`
- `atan2()`
- `attr()`
- `blur()`
- `brightness()`

- calc()
- character-variant()
- circle()
- clamp()
- color-contrast()
- color-mix()
- color()
- conic-gradient()
- contrast()
- cos()
- counter()
- counters()
- cross-fade()
- device-cmyk()
- drop-shadow()
- element()
- ellipse()
- env()
- exp()
- fit-content()
- grayscale()
- hsl()
- hsla()
- hue-rotate()
- hwb()
- hypot()
- image-set()
- image()

- inset()
- invert()
- lab()
- lch()
- linear-gradient()
- log()
- matrix()
- matrix3d()
- max()
- min()
- minmax()
- mod()
- oklab()
- oklch()
- opacity()
- ornaments()
- paint()
- path()
- perspective()
- polygon()
- pow()
- radial-gradient()
- rem()
- repeat()
- repeat-conic-gradiant()
- repeating-linear-gradiant()
- repeating-radial-gradient()
- rgb()

- rgba()
- rotate()
- rotate3d()
- rotateX()
- rotateY()
- rotateZ()
- round()
- saturate()
- scale()
- scale3d()
- scaleX()
- scaleY()
- scaleZ()
- sepia()
- sign()
- sin()
- skew()
- skewX()
- skewY()
- sqrt()
- styleset()
- stylistic()
- swash()
- symbols()
- tan()
- translate()
- translate3d()
- translateX()

- `translateY()`
- `translateZ()`
- `url()`
- `var()`

That's ninety-five different function values. We'll cover some of them in the rest of this chapter. The rest will be covered in other chapters, as appropriate for their topics (e.g., the filter functions are described in XREF HERE).

Calculation values

In situations where you need to do a little math, CSS provides a `calc()` value. Inside the parentheses, you can construct simple mathematical expressions. The permitted operators are `+` (addition), `-` (subtraction), `*` (multiplication), and `/` (division), as well as parentheses. These follow the traditional PEMDAS (parentheses, exponents, multiplication, division, addition, subtraction) precedence order, although in this case it's really just PMDAS since exponents are not permitted in `calc()`.

As an example, suppose you want your paragraphs to have a width that's 2 em less than 90% the width of their parent element. Here's how you express that with `calc()`:

```
p {width: calc(90% - 2em);}
```

`calc()` can be used in any property value where one of the following value types is permitted: *<length>*, *<frequency>*, *<angle>*, *<time>*, *<percentage>*, *<number>*, and *<integer>*. You can also use all these unit types within a `calc()` value, though there are some limitations to keep in mind.

The basic limitation is that `calc()` does basic type checking to make sure that units are, in effect, compatible. The checking works like this:

1. To either side of a `+` or `-` sign, both values must have the same unit type, or be a *<number>* and *<integer>* (in which case, the result is a *<number>*). Thus, `5 + 2.7` is valid, and results in `7.7`. On the other hand, `5em + 2.7` is invalid, because one side has a length unit and the other does not. Note that `5em + 20px` *is* valid, because `em` and `px` are both length units.

2. Given a `*`, one of the values involved must be a *<number>* (which, remember, includes integer values). So `2.5rem * 2` and `2 * 2.5rem` are both valid, and each result in `5rem`. On the flip side, `2.5rem * 2rem` is *not* valid, because the result would be $5rem^2$, and length units cannot be area units.

3. Given a `/`, the value on the *right* side must be a *<number>*. If the left side is an *<integer>*, the result is a *<number>*. Otherwise, the result is of the unit type used on the left side. This means that `30em / 2.75` is valid, but `30 / 2.75em` is not valid.

4. Furthermore, any circumstance that yields division by zero is invalid. This is easiest to see in a case like `30px/0`, but there are other ways to get there.

There’s one more notable limitation, which is that whitespace is *required* on both sides of the `+` and `-` operators, while it is not for `*` and `/`. This was done to allow future development of `calc()` values to support keywords that contain dashes (e.g., `max-content`).

Furthermore, it’s valid (and supported) to nest `calc()` functions inside each other. Thus you can say something like:

```
p {width: calc(90% - calc(1em + 0.1vh));}
```

Beyond that, the CSS specification requires that user agents support a *minimum* of 20 terms inside any single `calc()` function, where a term is a number, percentage, or dimension (e.g., a length). In situations where the number of terms somehow exceeds the user agent’s term limits, the entire function is treated as invalid.

Maximum Values

Calculation is nice, but sometimes you just want to make sure a property is set to one of a number of values, whichever is smallest. In those cases, the `min()` function value comes in very handy. Yes, this is confusing at first, but give us a minute and hopefully it will make sense.

Suppose you have an element you want to make sure is never wider than a certain amount; say, an image that should be one-quarter the width of the viewport or 200 pixels wide, whichever is *smaller*. This allows it to be constrained to 200 pixels of width on wide viewports, but take up to a quarter the width of smaller viewports. For that, you’d say:

```
.figure {width: min(25vw, 200px);}
```

The browser will compute the width of both `25vw` and compare that to `200px`, and use whichever is smaller. If `200px` is smaller than 25% the width of the viewport, then it will be used. Otherwise, the element will be 25% as wide as the viewport, which could easily be smaller than `1em`. Note that “smaller” in this case means “closest to negative infinity,” not “closest to zero.” Thus, if you compare two terms that compute to (say) `-1500px` and `-2px`, `min()` will pick `-1500px`.

You can nest `min()` inside `min()`, or throw a mathematical expression in there for one of the values, without having to wrap it in `calc()`. For that matter, you can put in `max()` and `clamp()`, which we haven’t even discussed yet. You can supply as many terms as you like: if you want to compare four different ways of measuring something, picking the minimum, then just separate them with commas. A slightly contrived example:

```
.figure {width: min(25vw, 200px, 33%, 50rem - 30px);}
```

Whichever of those values is computed to be the minimum (closest to negative infinity) will be used, thus defining a maximum for the `width` value. The order you list them in doesn’t actually matter, since the minimum value will always be picked regardless of where it appears in the function.

In general, `min()` can be used in any property value that permits *<length>*, *<frequency>*, *<angle>*,

<time>, *<percentage>*, *<number>*, or *<integer>*.

> **WARNING**
>
> Remember that setting a maximum value on font sizes is an accessibility concern. You should **never** set a maximum font size using pixels, because that would likely prevent text zooming by users. You probably shouldn't use `min()` for font sizing in any case, but if you do, keep `px` lengths out of the values!

Minimum Values

The mirror image of `min()` is `max()`, which can be used to set a minimum value for a property. It can appear the same places `min()` can, can be nested in the same ways `min()` can, and is generally just the same except that it picks the largest (closest to positive infinity) value from among the alternatives given.

As an example, perhaps the top of a page's design should be a minimum of 100 pixels tall, but it can be taller if conditions permit. In that case, you could use something like:

```
header {height: max(100px, 15vh, 5rem);}
```

Whichever of the values is largest will be used. For a desktop browser window, that would probably be `15vh`, unless the base size text is really enormous. For a handheld display, it's more likely that `5rem` or `100px` will be the largest value. In effect, this sets a minimum size of 100 pixels tall, since getting either `15vh` or `5rem` below that value is easily possible.

Remember that setting even a minimum value on font sizes can create an accessibility problem, since a too-small minimum is still too small. A good way to handle this is to always include `1rem` in your `max()` expressions for font sizes. Something like this:

```
.sosumi {font-size: max(1vh, 0.75em, 1rem);}
```

Alternatively, you could not use `max()` for font sizing at all. It's probably best left to box sizing and other such uses.

Clamping Values

If you've already been thinking about ways to nest `min()` and `max()` to set upper and lower bounds on a value, here's a way to not only do that, but set an "ideal" value as well: `clamp()`. This function value takes three parameters representing, in order, the minimum allowed value, preferred value, and maximum allowed value.

For example, consider some text you want to be about 5% the height of the viewport, while keeping its minimum the base font size and its maximum three times the text around it. That would be expressed like so:

Example 5-1.

---

footer {font-size: clamp(1rem, 2vh, 3em);}

Given those styles and assuming the base font size is 16 pixels, as it is by default in most browsers, then the footer text will be equal to the base font size up to a viewport height of 800 pixels (16 divided by .02). If the viewport gets taller, the text will start to get bigger, unless doing so would make it bigger than `3em`. If it ever gets to the same size as `3em`, then it will stop growing. (This is fairly unlikely, but one never knows.)

In any case where the maximum value of a `clamp()` is computed to be smaller than the minimum value, then the maximum is ignored and the minimum value is used instead.

You can use `clamp()` anywhere you can use `min()` and `max()`, including nesting them inside each other. For example:

Example 5-2.

---

footer {font-size: clamp(1rem, max(2vh, 1.5em), 3em);}

This is basically the same as the previous example, except in this case the preferred value is either 2% the height of the viewport or 1.5 times the size of the parent element's text, whichever is larger.

Attribute Values

In a few CSS properties, it's possible to pull in the value of an HTML attribute defined for the element being styled. This is done with the `attr()` function.

For example, with generated content, you can insert the value of any attribute. It looks something like this (don't worry about understanding the exact syntax, which we'll explore in XREF HERE):

```
p::before {content: "[" attr(id) "]";}
```

That expression would prefix any paragraph that has an `id` attribute with the value of that `id`, enclosed in square brackets. Therefore applying the previous style to the following paragraphs would have the result shown in Figure 5-8:

```
<p id="leadoff">This is the first paragraph.</p>
<p>This is the second paragraph.</p>
<p id="conclusion">This is the third paragraph.</p>
```

[leadoff]This is the first paragraph.

[]This is the second paragraph.

[conclusion]This is the third paragraph.

Figure 5-8. Inserting attribute values

While `attr()` is supported in the `content` property value, it isn't parsed. In other words, if the `attr()` returns an image URL from an attribute value, the generated content will be the URL written out as text, and not the image that lives at that URL. As of late 2022, anyway; there are plans to change things such that `attr()` can be parsed (and also be used for all properties, not just `content`).

Color

One of the first questions every starting web author asks is, “How do I set colors on my page?” Under HTML, you have two choices: you could use one of a large but limited number of colors with names, such as `red` or `purple`, or employ a vaguely cryptic method using hexadecimal codes. Both of these methods for describing colors remain in CSS, along with several—and, we think, more intuitive—methods.

Named Colors

Over the years, CSS has added a set of 148 colors that are identified by a human-readable names like `red` or `firebrickred`. CSS calls these, logically enough, *named colors*. In the early days of CSS, there were only the 16 basic color keywords defined in HTML 4.01. These are shown in Table 5-2.

Table 5-2. The basic 16 color keywords

aqua	gray	navy	silver	black	green	olive
blue	lime	purple	white	fuchsia	maroon	red

So, let’s say you want all first-level headings to be maroon. The best declaration would be:

```
h1 {color: maroon;}
```

Simple enough, isn’t it? Figure 5-9 shows a few more examples:

```
h1 {color: silver;}
h2 {color: gray;}
h3 {color: black;}
```

Greetings!

Salutations!

Howdy-do!

Figure 5-9. Named colors

You’ve probably seen (and maybe even used) color names other than the ones listed earlier. For example, you could say:

```
h1 {color: lightgreen;}
```

…and get a light green (but not exactly lime) color applied to `h1` elements.

The CSS color specification includes those original 16 named colors in a longer list of 148 color keywords. This extended list is based on the standard X11 RGB values that have been in use for decades, and have been recognized by browsers for many years, with the addition of some color names from SVG (mostly involving variants of “gray” and “grey”) and a memorial color. A table of color equivalents for all 148 keywords defined in the CSS Color Module Level 4 is given in XREF HERE.

Color Keywords

There are two special keywords that can be used anywhere a color value is permitted. These are `transparent` and `currentColor`.

As its name suggests, `transparent` defines a completely transparent color. The CSS Color Module defines it to be equivalent to `rgba(0,0,0,0)`, and that's its computed value. This keyword is not often used to set text color, for example, but it is the default value for element background colors. It can also be used to define element borders that take up space, but are not visible, and is often used when defining gradients—all topics we'll cover in later chapters.

By contrast, `currentColor` means "whatever the computed value of `color` is for this element." Consider the following:

```
main {color: gray; border-color: currentColor;}
```

The first declaration causes any `main` elements to have a foreground color of `gray`. The second declaration uses `currentColor` to copy the computed value of `color`—in this case `gray`—and apply it to any borders the `main` elements might have. Incidentally, `currentColor` is actually the default value for `border-color`, which we'll cover in Chapter 7.

As with all the named colors, these color names are case-insensitive. currentColor was shown here with mixed capitalization for legibility, and is generally written that way, again, for legibility.

Fortunately, there are more detailed and precise ways to specify colors in CSS. The advantage is that, with these methods, you can specify any color in the color spectrum, not just a limited list of named colors.

Colors by RGB and RGBa

Computers create colors by combining different levels of the primary colors red, green, and blue, a combination that is often referred to as *RGB color*. So, it makes sense that you be able to specify your own combinations of these primary colors in CSS. That solution is a bit complex, but possible, and the payoffs are worth it because there are very few limits on which colors you can produce. There are four ways to affect color in this manner.

Functional RGB colors

There are two color value types that use *functional RGB notation* as opposed to hexadecimal notation. The generic syntax for this type of color value is `rgb(color)`, where `color` is expressed using a triplet of either percentages or numbers. The percentage values can be in the range `0%`–`100%`, and the integers can be in the range `0`–`255`.

Thus, to specify white and black, respectively, using percentage notation, the values would be:

```
rgb(100%,100%,100%)
rgb(0%,0%,0%)
```

Using the integer-triplet notation, the same colors would be represented as:

```
rgb(255,255,255)
rgb(0,0,0)
```

An important thing to remember is that you can’t mix integers and percentages in the same color value. Thus, `rgb(255,66.67%,50%)` would be invalid and thus ignored.

> **NOTE**
>
> In more recent browsers, the separating commas in RGB values can be replaced with simple whitespace. Thus, black can be represented `rgb(0 0 0)` or `rgb(0% 0% 0%)`. This is true of all the color values we’ll see throughout the chapter that allow commas, but we’ll mostly stick to the comma notation for backwards compatibility and clarity’s sake. Also bear in mind that some of the newer color functions do not allow commas.

Assume you want your `h1` elements to be a shade of red that lies between the values for red and maroon. `red` is equivalent to `rgb(100%,0%,0%)`, whereas `maroon` is equal to `(50%,0%,0%)`. To get a color between those two, you might try this:

```
h1 {color: rgb(75%,0%,0%);}
```

This makes the red component of the color lighter than `maroon`, but darker than `red`. If, on the other hand, you want to create a pale red color, you would raise the green and blue levels:

```
h1 {color: rgb(75%,50%,50%);}
```

The closest equivalent color using integer-triplet notation is:

```
h1 {color: rgb(191,127,127);}
```

The easiest way to visualize how these values correspond to color is to create a table of gray values. The result is shown in Figure 5-10:

```
p.one {color: rgb(0%,0%,0%);}
p.two {color: rgb(20%,20%,20%);}
p.three {color: rgb(40%,40%,40%);}
p.four {color: rgb(60%,60%,60%);}
p.five {color: rgb(80%,80%,80%);}
p.six {color: rgb(0,0,0);}
p.seven {color: rgb(51,51,51);}
p.eight {color: rgb(102,102,102);}
p.nine {color: rgb(153,153,153);}
p.ten {color: rgb(204,204,204);}
```

[one] This is a paragraph.
[two] This is a paragraph.
[three] This is a paragraph.
[four] This is a paragraph.
[five] This is a paragraph.
[six] This is a paragraph.
[seven] This is a paragraph.
[eight] This is a paragraph.
[nine] This is a paragraph.
[ten] This is a paragraph.

Figure 5-10. Text set in shades of gray

Since we're dealing in shades of gray, all three RGB values are the same in each statement. If any one of them were different from the others, then a color hue would start to emerge. If, for example, `rgb(50%,50%,50%)` were modified to be `rgb(50%,50%,60%)`, the result would be a medium-dark color with just a hint of blue.

It is possible to use fractional numbers in percentage notation. You might, for some reason, want to specify that a color be exactly 25.5 percent red, 40 percent green, and 98.6 percent blue:

```
h2 {color: rgb(25.5%,40%,98.6%);}
```

Values that fall outside the allowed range for each notation are *clipped* to the nearest range edge, meaning that a value that is greater than `100%` or less than `0%` will default to those allowed extremes. Thus, the following declarations would be treated as if they were the values indicated in the comments:

```
P.one {color: rgb(300%,4200%,110%);}   /*  100%,100%,100%  */
P.two {color: rgb(0%,-40%,-5000%);}   /*  0%,0%,0%  */
p.three {color: rgb(42,444,-13);}    /* 42,255,0  */
```

Conversion between percentages and integers may seem arbitrary, but there's no need to guess at the integer you want—there's a simple formula for calculating them. If you know the percentages for each of the RGB levels you want, then you need only apply them to the number 255 to get the resulting values. Let's say you have a color of 25 percent red, 37.5 percent green, and 60 percent blue. Multiply each of these percentages by 255, and you get 63.75, 95.625, and 153. Round these values to the nearest integers, and *voilà*: `rgb(64,96,153)`.

If you already know the percentage values, there isn't much point in converting them into integers. Integer notation is more useful for people who use programs such as Photoshop, which can display integer values in the Info dialog, or for those who are so familiar with the technical details of color generation that they normally think in values of 0–255.

RGBa colors

Many years ago, the two functional RGB notations were extended into a functional RGBa notation. This notation adds an alpha value to the end of the RGB triplets; thus "red-green-blue-alpha" becomes RGBa. The alpha stands for *alpha channel*, which is a measure of opacity.

For example, suppose you wanted an element's text to be half-opaque white. That way, any background color behind the text would "shine through," mixing with the half-transparent white. You would write one of the following two values:

```
rgba(255 255 255 / 0.5)
rgba(100% 100% 100% / 0.5)  /* commas would also be allowed */
```

To make a color completely transparent, you set the alpha value to `0`; to be completely opaque, the correct value is `1`. Thus `rgb(0,0,0)` and `rgba(0,0,0,1)` will yield precisely the same result (black). Figure 5-11 shows a series of paragraphs set in increasingly transparent black, which is the result of the following rules.

```
p.one {color: rgba(0,0,0,1);}
p.two {color: rgba(0%,0%,0%,0.8);}
p.three {color: rgba(0,0,0,0.6);}
p.four {color: rgba(0%,0%,0%,0.4);}
p.five {color: rgba(0,0,0,0.2);}
```

[one] This is a paragraph.
[two] This is a paragraph.
[three] This is a paragraph.
[four] This is a paragraph.
[five] This is a paragraph.

Figure 5-11. Text set in progressive translucency

Alpha values are always real numbers in the range `0` to `1`, or percentages in the range `0%` to `100%`. Any value outside that range will either be ignored or reset to the nearest valid alpha value. You cannot use <*percentage*> to represent alpha values, despite the mathematical equivalence.

Hexadecimal RGB colors

CSS allows you to define a color using the same *hexadecimal color notation* so familiar to old-school HTML web authors:

```
h1 {color: #FF0000;}   /* set H1s to red */
h2 {color: #903BC0;}   /* set H2s to a dusky purple */
h3 {color: #000000;}   /* set H3s to black */
h4 {color: #808080;}   /* set H4s to medium gray */
```

Computers have been using hex notation for quite some time now, and programmers are typically either trained in its use or pick it up through experience. Their familiarity with hexadecimal notation likely led to its use in setting colors in HTML. That practice was carried over to CSS.

Here's how it works: by stringing together three hexadecimal numbers in the range `00` through `FF`, you can set a color. The generic syntax for this notation is `#RRGGBB`. Note that there are no spaces, commas, or other separators between the three numbers.

Hexadecimal notation is mathematically equivalent to integer-pair notation. For example, `rgb(255,255,255)` is precisely equivalent to `#FFFFFF`, and `rgb(51,102,128)` is the same as

#336680. Feel free to use whichever notation you prefer—it will be rendered identically by most user agents. If you have a calculator that converts between decimal and hexadecimal, making the jump from one to the other should be pretty simple.

For hexadecimal numbers that are composed of three matched pairs of digits, CSS permits a shortened notation. The generic syntax of this notation is #RGB:

```
h1 {color: #000;}   /* set H1s to black */
h2 {color: #666;}   /* set H2s to dark gray */
h3 {color: #FFF;}   /* set H3s to white */
```

As you can see from the markup, there are only three digits in each color value. However, since hexadecimal numbers between 00 and FF need two digits each, and you have only three total digits, how does this method work?

The answer is that the browser takes each digit and replicates it. Therefore, #F00 is equivalent to #FF0000, #6FA would be the same as #66FFAA, and #FFF would come out #FFFFFF, which is the same as white. Not every color can be represented in this manner. Medium gray, for example, would be written in standard hexadecimal notation as #808080. This cannot be expressed in shorthand; the closest equivalent would be #888, which is the same as #888888.

Hexadecimal RGBa colors

Hexadecimal notation can have a fourth hex value to represent the alpha channel value. Figure 5-11 shows a series of paragraphs set in increasingly transparent black, just as we saw in the previous section, which is the result of the following rules:

```
p.one {color: #000000FF;}
p.two {color: #000000CC;}
p.three {color: #00000099;}
p.four {color: #00000066;}
p.five {color: #00000033;}
```

[one] This is a paragraph.
[two] This is a paragraph.
[three] This is a paragraph.
[four] This is a paragraph.
[five] This is a paragraph.

Figure 5-12. Text set in progressive translucency, redux

As with non-alpha hexadecimal values, it's possible to shorten a value composed of matched pairs to a four-digit value. Thus, a value of #663399AA can be written as #639A. If the value has any pairs that are not repetitive, then the entire eight-digit value must be written out: #663399CA cannot be shortened to #639CA.

HSL and HSLa colors

HSL (hue-saturation-lightness) color notation is similar to HSB (hue-saturation-brightness), the color

system in image editing software like Photoshop, and just as intuitive. The hue is expressed as an angle value, saturation is a percentage value from 0 (no saturation) to 100 (full saturation), and lightness is a percentage value from 0 (completely dark) to 100 (completely light). If you're intimately familiar with RGB, then HSL may be confusing at first. (But then, RGB is confusing for people familiar with HSL.)

The hue angle is expressed in terms of a circle around which the full spectrum of colors progresses. It starts with red at 0 degrees and then proceeds through the rainbow until it comes to red again at 360 degrees.

As for the other two values, saturation measures the intensity of a color. A saturation of `0%` always yields a shade of gray, no matter what hue angle you have set, and a saturation of `100%` creates the most vivid possible shade of that hue (in the HSL color space) for a given lightness.

Similarly, lightness defines how dark or light the color appears. A lightness of `0%` is always black, regardless of the other hue and saturation values, just as a lightness of `100%` always yields white. Consider the results of the following styles, illustrated on the left side of Figure 5-13.

```
p.one {color: hsl(0,0%,0%);}
p.two{color: hsl(60,0%,25%);}
p.three {color: hsl(120,0%,50%);}
p.four {color: hsl(180,0%,75%);}
p.five {color: hsl(240,0%,0%);}
p.six {color: hsl(300,0%,25%);}
p.seven {color: hsl(360,0%,50%);}
```

NOTE

Remember that in more recent browsers, the commas in `hsl()` values can be replaced with whitespace.

[one] This paragraph's color has 0% saturation.	[one] This paragraph's color has 50% saturation.
[two] This paragraph's color has 0% saturation.	[two] This paragraph's color has 50% saturation.
[three] This paragraph's color has 0% saturation.	[three] This paragraph's color has 50% saturation.
[four] This paragraph's color has 0% saturation.	[four] This paragraph's color has 50% saturation.
[five] This paragraph's color has 0% saturation.	[five] This paragraph's color has 50% saturation.
[six] This paragraph's color has 0% saturation.	[six] This paragraph's color has 50% saturation.
[seven] This paragraph's color has 0% saturation.	[seven] This paragraph's color has 50% saturation.

Figure 5-13. Varying lightness and hues

The gray you see on the left side isn't just a function of the limitations of print: every one of those paragraphs is a shade of gray, because every color value has `0%` in the saturation (middle) position. The degree of lightness or darkness is set by the lightness (third) position. In all seven examples, the hue angle changes, and in none of them does it matter. But that's only so long as the saturation remains at `0%`. If that value is raised to, say, `50%`, then the hue angle will become very important, because it will control what sort of color you see. Consider the same set of values that we saw before, but all set to `50%` saturation, as

illustrated on the right side of Figure 5-13.

Just as RGB has its RGBa counterpart, HSL has an HSLa counterpart. This is an HSL triplet followed by an alpha value in the range 0–1. The following HSLa values are all black with varying shades of transparency, just as in "Hexadecimal RGBa colors" (and illustrated in Figure 5-11):

```
p.one {color: hsla(0,0%,0%,1);}
p.two {color: hsla(0,0%,0%,0.8);}
p.three {color: hsla(0,0%,0%,0.6);}
p.four {color: hsla(0,0%,0%,0.4);}
p.five {color: hsla(0,0%,0%,0.2);}
```

Colors with HWB

Colors can also be represented in terms of their Hue, White level, and Black level by using the `hwb()` functional value. This function value accepts hue values expressed as an angle value. After the hue angle, instead of lightness and saturation, whiteness and blackness values are specified as percentages.

Unlike HSL, however, there is no `hwba()` function. Instead, the value syntax for `hwb()` allows an opacity to be defined after the HWB values, separated from them by a solidus (`/`). The Opacity can be expressed either as a percentage or as a real value from 0 to 1, inclusive. Also unlike HSL, commas are not supported: the HWB values must be separated by whitespace.

Here are some examples of using HWB notation:

```
/* Varying shades of red */
hwb(0 40% 20%)
hwb(360 50% 10%)
hwb(0deg 10% 10%)
hwb(0rad 60% 0%)
hwb(0turn 0% 40%)

/* Partially translucent red */
hwb(0 10% 10% / 0.4)
hwb(0 10% 10% / 40%)
```

Lab colors

Historically, all CSS colors were defined in the sRGB color space, which was more than older display monitors could represent. Modern displays, on the other hand, can handle about 150% of the the sRGB color space, which still isn't the full range of color humans can perceive, but it's a lot closer.

In 1931, the *Commission Internationale de l'Éclairage* (International Commission on Illumination), or CIE, defined a scientific system for defining colors created via light, as opposed to those created with paint or dyes. Now, almost a century later, CSS has brought the work of the CIE into its repertoire.

It does this using the `lab()` function value to express color using the CIE L*a*b* (hereafter shortened as "Lab") color space. Lab is designed to represent the entire range of color that humans can see. The `lab()` function accepts 3 to 4 parameters: `lab(L a b / A)`. Similar to HWB, the parameters are space separated (no commas allowed) and a solidus (`/`) precedes alpha value, if provided.

The L (Lightness) component specifies the CIE Lightness, and is a <*percentage*> between `0%` representing black and `100%` representing white, or else a <*number*> from `0` to `1`. The second component, `a`, is the distance along the a axis in the Lab colorspace. This axis runs from a purplish red in the positive direction to a shade of green in the negative direction. The third component, `b`, is the distance along the b axis in the Lab colorspace. This axis runs from a yellow in the positive direction to a blue-violet in the negative direction.

The fourth, optional parameter is the opacity, with a value between 0 and 1 inclusive, or 0 to 100% inclusive. If omitted, the opacity defaults to 1 (100%), or full opacity.

Here are some examples of Lab color expressed in CSS.

```
lab(29.2345% 39.3825 -20.0664);
lab(52.2345% 40.1645 59.9971);
lab(52.2345% 40.1645 59.9971 / .5);
```

The main reason to bring Lab (and LCH, which we'll discuss in a moment) colors into CSS is that they are systematically designed to be *perceptually uniform*. What the means is, color values that share a given coordinate will seem consistent in terms of that coordinate. Two colors with different hues but the same lightness will appear to have similar lightnesses. Two colors with the same hue but different lightnesses will appear to be shades of a single hue. This is often not the case with RGB and HSL values, so Lab and LCH represent a big improvement.

They're also defined to be device-independent, so you should be able to specify colors in these color spaces and get a visually consistent result from one device to another.

> **WARNING**
>
> As of late 2022, only WebKit supported `lab()`.

LCH colors

LCH, for "Lightness Chroma Hue", is a version of Lab designed to represent the entire spectrum of human vision. It does this using a different notation: `lch(L C H / A)`. The main difference is that `C` and `H` are polar coordinates, rather than linear values along color axes.

The L (Lightness) component is the same as the CIE Lightness, and is a <*percentage*> between `0%` representing black and `100%` representing white.

The C (Chroma amount) component roughly represents the "amount of color". Its minimum value is `0`, and there is no defined maximum. Negative `C` values are clamped to zero.

The H (Hue angle) component is essentially a combination of the `a` and `b` values in `lab()`. The value `0` points along the positive "a" axis (toward purplish red), `90` points along the positive "b" axis (toward mustard yellow), `180` points along the negative "a" axis (toward greenish cyan), and `270` points along the negative "b" axis (toward sky blue). This component loosely corresponds to HSL's Hue, but the hue angles differ.

The optional A (alpha) component can be a <*number*> between 0 and 1, or else a <*percentage*>, where the number 1 corresponds to 100% (full opacity). If present, it is preceded by a solidus (`/`).

```
lch(56% 132 331)
lch(52% 132 8)
lch(52% 132 8 / 50%)
```

To give an example of the capabilities of LCH, `lch(52% 132 8)` is a very bright magenta equivalent to `rgb(118.23% -46.78% 40.48%)`. Notice the large red value and the negative green value, which places the color outside the sRGB color space. If you supplied that RGB value to a browser, it would clamp the value to `rgb(100% 0% 40.48%)`. This is within the sRGB color space, but is visually quite distinct from the color defined by `lch(52% 132 8)`.

> **WARNING**
>
> As of late 2022, Firefox did not yet support `lch()`, support was just coming to Chrome, and it had been present for a while in Safari.

Oklab and Oklch

There are improved versions of Lab and LCH called Oklab and Oklch, and these will be supported by CSS using the `oklab()` and `oklch()` functional values. Oklab was developed by taking a large set of visually similar colors and performing a numerical optimization on them, yielding a color space with better hue linearity and uniformity, and better chroma uniformity, than the CIE color spaces. Oklch is a polar-coordinate version of Oklab, just as LCH is to Lab.

Because of this improved uniformity, Oklab and Oklch will be the default for color-interpolation calculations in CSS going foward. However, as of the time of writing in late 2022, only Safari supported the `oklab()` and `oklch()` CSS functional values, and that only recently.

color()

The `color()` function value allows a color to be specified in a named colorspace, rather than the implicit sRGB colorspace. It accepts four space-separated parameters, as well as an optional fifth opacity value preceded by a solidus (/).

The first parameter is a predefined, named color space. Possible values as of early 2022 include `srgb`, `srgb-linear`, `display-p3`, `a98-rgb`, `prophoto-rgb`, `rec2020`, `xyz`, `xyz-d50`, and `xyz-d65` . The three values that follow are specific to the color space declared in the first parameter. Some color spaces may allow these values to be percentages, while others may not.

As an example, the following values should yield the same color:

```
#7654CD
rgb(46.27% 32.94% 80.39%)
lab(44.36% 36.05 -58.99)
color(xyz-d50 0.2005 0.14089 0.4472)
color(xyz-d65 0.21661 0.14602 0.59452)
```

It is easily possible to declare a color that lies outside the gamut of a given color space. For example, `color(display-p3 -0.6112 1.0079 -0.2192);` is outside the display-p3 gamut. It's still a valid color, just not one that can be expressed in that color space. In the case where a color value is valid but outside the gamut, it will be mapped to the closest color that lies inside the color space's gamut.

In cases where a color's value is straight up invalid, then the color used is opaque black.

> **WARNING**
>
> As of late 2022, Firefox did not yet support `color()`, support was just coming to Chrome, and it had been present for a while in Safari.

Applying Color

Since we've just gone through all the different possible color formats, let's take a brief detour to talk about the property that uses color values the most often: `color`.

COLOR

Values	*<color>*
Initial value	User agent-specific
Applies to	All elements
Computed value	As specified
Inherited	Yes
Animatable	Yes

This property accepts as a value any valid color type, such as `#FFCC00` or `rgba(100%,80%,0%,0.5)`.

For nonreplaced elements like paragraphs or `em` elements, `color` sets the color of the text in the element, as illustrated in Figure 5-14, which is the result of the following code:

```
<p style="color: gray;">This paragraph has a gray foreground.</p>
<p>This paragraph has the default foreground.</p>
```

This paragraph has a gray foreground.

This paragraph has the default foreground.

Figure 5-14. Declared color versus default color

In Figure 5-14, the default foreground color is black. That doesn't have to be the case, since the user

might have set her browser (or other user agent) to use a different foreground (text) color. If the browser's default text color was set to green, the second paragraph in the preceding example would be green, not black—but the first paragraph would still be gray.

You need not restrict yourself to such basic operations. There are plenty of ways to use color. You might have some paragraphs that contain text warning the user of a potential problem. In order to make this text stand out more than usual, you might decide to color it red. Just apply a class of warn to each paragraph that contains warning text (<p class="warn">) and the following rule:

```
p.warn {color: red;}
```

In the same document, you might decide that any unvisited hyperlinks within a warning paragraph should be green:

```
p.warn {color: red;}
p.warn a:link {color: green;}
```

Then you change your mind, deciding that warning text should be dark red, and that unvisited links in such text should be medium purple. The preceding rules need only be changed to reflect the new values, as illustrated in Figure 5-15, which is the result of the following code:

```
p.warn {color: #600;}
p.warn a:link {color: #400040;}
```


Figure 5-15. Changing colors

Another use for color is to draw attention to certain types of text. For example, boldfaced text is already fairly obvious, but you could give it a different color to make it stand out even further—let's say, maroon:

```
b, strong {color: maroon;}
```

Then you decide that you want all table cells with a class of `highlight` to contain light yellow text:

```
td.highlight {color: #FF9;}
```

If you don't set a background color for any of your text, you run the risk that a user's setup won't combine well with your own. For example, if a user has set their browser's background to be a pale yellow, like `#FFC`, then the previous rule would generate light yellow text on a pale yellow background. Far more likely is that it's still the default background of white, against which light yellow is still going to be hard to read. It's therefore generally a good idea to set foreground and background colors together. (We'll talk about background colors very shortly.)

Affecting Form Elements

Setting a value for `color` should (in theory, anyway) apply to form elements. Declaring `select` elements to have dark gray text should be as simple as this:

```
select {color: rgb(33%,33%,33%);}
```

This might also set the color of the borders around the edge of the `select` element, or it might not. It all depends on the user agent and its default styles.

You can also set the foreground color of input elements—although, as you can see in Figure 5-16, doing so would apply that color to all inputs, from text to radio button to checkbox inputs:

```
select {color: rgb(33%,33%,33%);}
input {color: red;}
```

This is a select list!

Option 1 Option 2 Option 3 Option 4

Submit me, O user

Figure 5-16. Changing form element foregrounds

Note in Figure 5-16 that the text color next to the checkboxes is still black. This is because the rules shown assign styles only to elements like `input` and `select`, not normal paragraph (or other) text.

Also note that the checkmark in the checkbox is black. This is due to the way form elements are handled in some web browsers, which typically use the form widgets built into the base operating system. Thus, when you see a checkbox and checkmark, they really aren't content in the HTML document—they're user interface widgets that have been inserted into the document, much as an image would be. In fact, form inputs are, like images, replaced elements. In theory, CSS does not style the contents of replaced elements.

In practice, the line is a lot blurrier than that, as Figure 5-16 demonstrates. Some form inputs have the color of their text and even portions of their UI changed, while others do not. And since the rules aren't

explicitly defined, behavior is inconsistent across browsers. In short, form elements are deeply tricky to style and should be approached with extreme caution.

Inheriting Color

As the definition of `color` indicates, the property is inherited. This makes sense, since if you declare `p {color: gray;}`, you probably expect that any text within that paragraph will also be gray, even if it's emphasized or boldfaced or whatever. If you *want* such elements to be different colors, that's easy enough, as illustrated in Figure 5-17, which is the result of the following code:

```
em {color: red;}
p {color: gray;}
```

This is a paragraph which is, for the most part, utterly undistinguished—but its *emphasized text* is quite another story altogether.

Figure 5-17. Different colors for different elements

Since color is inherited, it's theoretically possible to set all of the ordinary text in a document to a color, such as red, by declaring `body {color: red;}`. This should make all text that is not otherwise styled (such as anchors, which have their own color styles) red.

Angles

Since we just recently finished talking about hue angles in a number of color value types, this would be a good time to talk about angle units. Angles in general are represented as *<angle>*, which is a *<number>* followed by one of four unit types:

`deg`

Degrees, of which there are 360 in a full circle.

`grad`

Gradians, of which there are 400 in a full circle. Also known as *grades* or *gons*.

`rad`

Radians, of which there are 2π (approximately 6.28) in a full circle.

`turn`

Turns, of which there is one in a full circle. This unit is mostly useful when animating a rotation and you wish to have it turn multiple times, such as `10turn` to make it spin 10 times. (Sadly, the pluralization `turns` is invalid, at least as of early 2022, and will be ignored.)

To help understand the relationsihp between these different angle types, Table 5-3 shows how some angles are expressed in the various angle units.

Table 5-3. Angle equivalents

Degrees	Gradians	Radians	Turns
`0deg`	`0grad`	`0rad`	`0turn`
`45deg`	`50grad`	`0.785rad`	`0.125turn`
`90deg`	`100grad`	`1.571rad`	`0.25turn`
`180deg`	`200grad`	`3.142rad`	`0.5turn`
`270deg`	`300grad`	`4.712rad`	`0.75turn`
`360deg`	`400grad`	`6.283rad`	`1turn`

Time and Frequency

In cases where a property needs to express a period of time, the value is represented as <*time*> and is a <*number*> followed by either `s` (seconds) or `ms` (milliseconds.) Time values are most often used in transitions and animations, either to define durations or delays. The following two declarations will have exactly the same result:

```
a[href] {transition-duration: 2.4s;}
a[href] {transition-duration: 2400ms;}
```

Time values are also used in aural CSS, again to define durations or delays, but support for aural CSS is extremely limited as of this writing.

Another value type historically used in aural CSS is <*frequency*>, which is a <*number*> followed by either `Hz` (hertz) or `kHz` (kilohertz). As usual, the unit identifiers are case-insensitive, so `Hz` and `hz` are equivalent. The following two declarations will have exactly the same result:

```
h1 {pitch: 128hz;}
h1 {pitch: 0.128khz;}
```

Unlike with length values, for time and frequency values the unit type is *always* required, even when the value is `0s` or `0hz`.

Ratios

There are a couple of situations where it's necessary to express a ratio of two numbers, in which case a <*ratio*> value is used. These values are represented as two positive <*number*> values separated by a solidus (`/`), plus optional whitespace.

The first integer refers to the width (inline-size) of an element, and the second to the height (block-size). Thus, to express a height-to-width ratio of 16 to 9, you can write `16/9` or `16 / 9`.

As of late 2022, there is no facility to express a ratio as a single real number (e.g., `1.777` instead of `16/9`), nor to use a colon separator instead of a solidus (e.g., `16:9`).

Position

A *position value* is how you specify the placement of an origin image in a background area, and is represented as <*position*>. Its syntactical structure is rather complicated:

```
[
  [ left | center | right | top | bottom | <percentage> | <length> ] |
  [ left | center | right | <percentage> | <length> ]
  [ top | center | bottom | <percentage> | <length> ] |
  [ center | [ left | right ] [ <percentage> | <length> ]? ] &&
  [ center | [ top | bottom ] [ <percentage> | <length> ]? ]
]
```

That might seem a little nutty, but it’s all down to the subtly complex patterns that this value type has to allow.

If you declare only one value, such as `left` or `25%`, then the second value is set to `center`. Thus, `left` is the same as `left center`, and `25%` is the same as `25% center`.

If you declare (either implicitly, as above, or explicitly) two values, and the first one is a <*length*> or <*percentage*>, then it is *always* considered to be the horizontal value. This means that given `25% 35px`, the `25%` is a horizontal distance and the `35px` is a vertical distance. If you swap them to say `35px 25%`, then `35px` is horizontal and `25%` is vertical. This means that if you write `25% left` or `35px right`, the entire value is invalid because you have supplied two horizontal distances and no vertical distance. (Similarly, a value of `right left` or `top bottom` is invalid and will be ignored.) On the other hand, if you write `left 25%` or `right 35px`, there is no problem because you’ve given a horizontal distance (with the keyword) and a vertical distance (with the percentage or length).

If you declare four values (we’ll deal with three just in a moment), then you must have two lengths or percentages, each of which is preceded by a keyword. In this case, each length or percentage specifies an offset distance, and each keyword defines the edge from which the offset is calculated. Thus, `right 10px bottom 30px` means an offset of 10 pixels to the left of the right edge, and an offset of 30 pixels up from the bottom edge. Similarly, `top 50% left 35px` means a 50 percent offset from the top and a 35-pixels-to-the-right offset from the left.

You can only declare three position values with the `background-position` property. If you declare three values, the rules are the same as for four, except the fourth offset is set to be zero (no offset). Thus `right 20px top` is the same as `right 20px top 0`.

Custom Properties

If youve used a preprocessor like Less or Sass, you've probably created variables to hold values. CSS itself has this capability as well. The technical term for this is *custom properties*, even though what these really do is create sort of variables in your CSS.

Here's a basic example, with the result shown in Figure 5-18:

```
html {
    --base-color: #639;
    --highlight-color: #AEA;
}

h1 {color: var(--base-color);}
h2 {color: var(--highlight-color);}
```

Heading 1

Main text.

Heading 2

More text.

Figure 5-18. Using custom values to color headings

There are two things to absorb here. The first is the definition of the custom values `--base-color` and `--highlight-color`. These are not some sort of special color types. They're just names that were picked to describe what the values contain. We could just as easily have said:

```
html {
    --alison: #639;
    --david: #AEA;
}

h1 {color: var(--alison);}
h2 {color: var(--david);}
```

You probably shouldn't do that sort of thing, unless you're literally defining colors that specifically correspond to people named Alison and David. (Perhaps on an "About Our Team" page.) It's always better to define custom identifiers that are self-documenting—things like `main-color` or `accent-`

`color` or `brand-font-face`.

The important thing is that any custom identifier of this type begins with *two* hyphens (`--`). It can then be invoked later on using a `var()` value type. Note that these names are case-sensitive, so `--main-color` and `--Main-color` are completely separate identifiers.

These custom identifiers are often referred to as “CSS variables,” which explains the `var()` pattern. An interesting feature of custom properties is their ability to scope themselves to a given portion of the DOM. If that sentence made any sense to you, it probably gave a little thrill. If not, here’s an example to illustrate scoping, with the result shown in Figure 5-19:

```
html {
    --base-color: #666;
}
aside {
    --base-color: #CCC;
}

h1 {color: var(--base-color);}
```

```
<body>

<h1>Heading 1</h1><p>Main text.</p>

<aside>
    <h1>Heading 1</h1><p>An aside.</p>
</aside>

<h1>Heading 1</h1><p>Main text.</p>

</body>
```

Heading 1

Main text.

Heading 1

An aside.

Heading 1

Main text.

Figure 5-19. Scoping custom values to certain contexts

Notice how the headings are a dark gray outside the `aside` element and a light gray inside. That's because the variable `--base-color` was updated for `aside` elements. The new custom value applies to any `h1` inside an `aside` element.

There are a great many patterns possible with CSS variables, even if they are confined to value replacement. Here's an example suggested by Chriztian Steinmeier combining variables with the `calc()` function to create a regular set of indents for unordered lists:

```
html {
    --gutter: 3ch;
    --offset: 1;
}
ul li {margin-left: calc(var(--gutter) * var(--offset));}
ul ul li {--offset: 2;}
ul ul ul li {--offset: 3;}
```

This particular example is basically the same as writing:

```
ul li {margin-left: 3ch;}
ul ul li {margin-left: 6ch;}
ul ul ul li {margin-left: 9ch;}
```

The difference is that with variables, it's simple to update the `--gutter` multiplier in one place and have everything adjust automatically, rather than having to retype three values and make sure all the math is correct.

Custom property fallbacks

When you're setting a value using `var()`, you can specify a fallback value. For example, you could say that if a custom property isn't defined, you want a regular value used instead like so:

```
ol li {margin-left: var(--list-indent, 2em);}
```

Given that, if `--list-indent` isn't defined, was determined to be invalid, or is explicitly set to `initial`, `2em` will be used instead. You just get the one fallback, and it can't be another custom property name.

That said, it *can* be another `var()` expression, and that nested `var()` can contain another `var()` as its fallback, and so on to infinity. So let's say you're using a pattern library that defines colors for various interface elements. If those aren't available for some reason, then you could fall back to a custom-property value defined by your basic site stylesheet. Then, if that's also not available, you could fall back to a plain color value. It would look something like this:

```
.popup {color: var(--pattern-modal-color, var(--highlight-color, maroon));}
```

The thing to watch out for here is that if you manage to create an invalid value, the whole things gets

blown up and the value is either inherited or set to its initial value, depending on whether the property in question is usually inherited or not, as if it were set to `unset` (see "unset").

Suppose we wrote the following invalid `var()` values.

```
:root {
        --list-color: hsl(23, 25%, 50%);
        --list-indent: 5vw;
}

li {
        color: var(--list-color, --base-color, gray);
        margin-left: var(--list-indent, --left-indent, 2em);
}
```

In the first case, the fallback is `--base-color, gray` as a single string, not something that's parsed, so it's invalid. Similarly, in the second case, the fallback is the invalid `--left-indent, 2em`. In either case, if the first custom property is valid, then the invalid fallback doesn't matter, because the browser never gets to it. But if, say, `--list-indent` doesn't have a value, then the browser will go to the fallback, and here that's invalid. So what happens next?

For the color, since the property `color` is inherited, the list items will inherit their color from their parent, almost certainly an `ol` or `ul` element. If the parent's `color` value is `fuchsia`, then the list items will be fuchsia. For the left margin, the property `margin-left` is not inherited, so the left margins of the list items will be set to the initial value of `margin-left`, which is `0`. So the list items will have no left margin.

This also happens if you try to apply a value to a property that can't accept those kinds of values. Consider:

```
:root {
        --list-color: hsl(23, 25%, 50%);
        --list-indent: 5vw;
}

li {
        color: var(--list-indent, gray);
        margin-left: var(--list-color, 2em);
}
```

Here, everything looks fine at first glance, except the `color` property is being given a length value, and the `margin-left` property is being given a color value. As a result, the fallbacks of `gray` and `2em` are not used. This is because the `var()` syntax is valid, so the result is the same as if we declared `color: 5vw` and `margin-left: hsl(23, 25%, 50%)`, both of which are tossed out as invalid.

This means the outcome will be the same as we saw before: the list items will inherit the color value from their parents, and their left margins will be set to the initial value of zero, just as if the given values were `unset`.

Summary

As we’ve seen, CSS provides a wide range of value and unit types. These units can all have their advantages and drawbacks, depending on the circumstances in which they’re used. We’ve already seen some of those circumstances, and their nuances, will be discussed throughout the rest of the book, as appropriate.

Chapter 6. Basic Visual Formatting

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at rfernando@oreilly.com.

You've likely experienced the frustration of having your intended layout not rendered as expected. You then added 27 style rules to get it perfect, but maybe you didn't really know which rule solved your problem. With a model as open and powerful as that contained within CSS, no book could hope to cover every possible way of combining properties and effects. You will undoubtedly go on to discover new ways of using CSS. With a thorough grasp of how the visual rendering model works, however, you'll be better able to determine whether a behavior is a correct (if unexpected) consequence of the rendering engine CSS defines.

Basic Boxes

At its core, CSS assumes that every element generates one or more rectangular boxes, called *element boxes*. (Future versions of the specification may allow for nonrectangular boxes, and indeed there have been proposals to change this, but for now everything is still rectangular.)

Each element box has a *content area* at its center. This content area is surrounded by optional amounts of padding, borders, outlines, and margins. These areas are considered optional because they could all be set to a size of zero, effectively removing them from the element box. An example content area is shown in Figure 6-1, along with the surrounding regions of padding, borders, and margins.

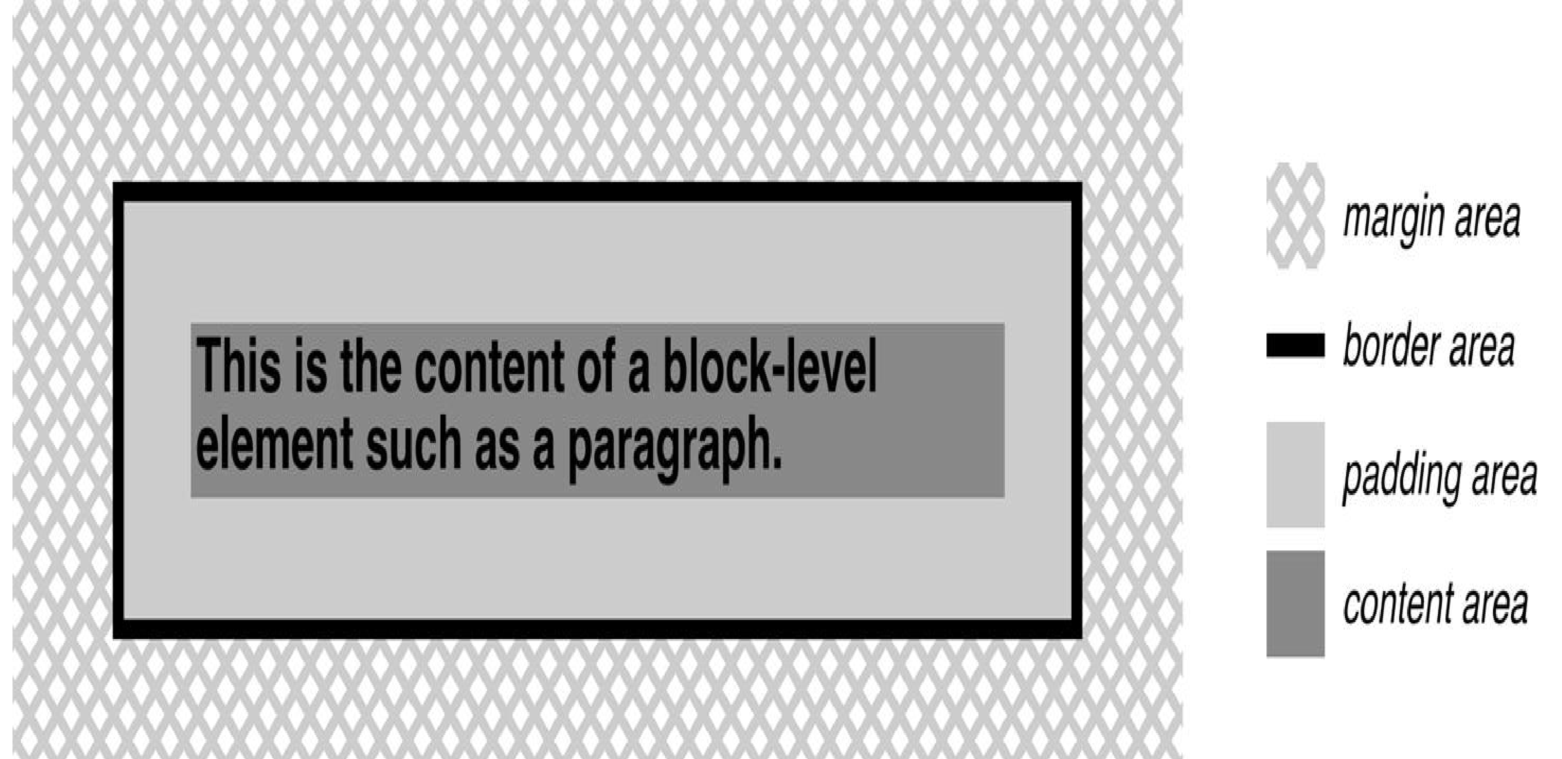


Figure 6-1. The content area and its surroundings

Before looking at the properties that can alter the space taken up by elements, let’s cover the vocabulary needed to fully understand how elements are laid out and take up space.

A Quick Primer

First, we’ll quickly review the kinds of boxes we’ll be discussing, as well as some important terms that are needed to follow the explanations to come:

Block flow direction

Also known as the *block axis*. This is the direction along which block-level element boxes are stacked. In many languages, including all European and European-derived languages, this direction is from top to bottom. In CJK (Chinese/Japanese/Korean) languages, this can be either right-to-left or top-to-bottom. The actual block flow direction is set by the writing mode, which is discussed in Chapter 11.

Inline base direction

Also known as the *inline axis*. This is the direction along which lines of text are written. In Romanic languages, among others, this is left-to-right. In languages such as Arabic or Hebrew, the inline base direction is right-to-left. In CJK (Chinese/Japanese/Korean) languages, this can be either top-to-bottom or left-to-right. As with block flow, the inline base direction is set by the writing mode.

Normal flow

The default system by which elements are placed inside the browser’s viewport, based on the parent’s writing mode. Most elements are in the normal flow, and the only way for an element to leave the normal flow is to be floated, positioned, or made into a flexible box, grid layout, or table element.

The discussions in this chapter will cover elements in the normal flow, unless otherwise stated.

Block box

This is a box generated by an element such as a paragraph, heading, or `div`. These boxes generate "blank spaces" both before and after their boxes when in the normal flow so that block boxes in the normal flow stack along the block flow axis, one after another. Pretty much any element can be made to generate a block box by declaring `display: block`, though there are other ways to make elements generate block boxes (e.g, float them or make them flex items).

Inline box

This is a box generated by an element such as `strong` or `span`. These boxes are laid out along the inline base direction, and do not generate "line breaks" before or after themselves. An inline box longer than the inline size of its element will (by default, if it's non-replaced) wrap to multiple lines. Any element can be made to generate an inline box by declaring `display: inline`.

Nonreplaced element

This is an element whose content is contained within the document. For example, a paragraph (`p`) is a nonreplaced element because its textual content is found within the element itself.

Replaced element

This is an element that serves as a placeholder for something else. The classic example of a replaced element is the `img` element, which simply points to an image file that is inserted into the document's flow at the point where the `img` element itself is found. Most form elements are also replaced (e.g., `<input type="radio">`).

Root element

This is the element at the top of the document tree. In HTML documents, this is the element `<html>`. In XML documents, it can be whatever the language permits: for example, the root element of RSS files is `<rss>`, whereas in an SVG document, the root element is `<svg>`.

The Containing Block

There is one more kind of box that we need to examine in detail, and in this case enough detail that it merits its own section: the *containing block*.

Every element's box is laid out with respect to its containing block. In a very real way, the containing block is the "layout context" for a box. CSS defines a series of rules for determining a box's containing block.

For a given element, the containing block forms from the *content edge* of the nearest ancestor element that generates a list-item or block box, which includes all table-related boxes (e.g., those generated by table cells). Consider the following markup:

```
<body>
    <div>
        <p>This is a paragraph.</p>
    </div>
</body>
```

In this very simple markup, the containing block for the `p` element's block box is the `div` element's block box, as that is the closest ancestor element box that has a block or a list item box (in this case, it's a block box). Similarly, the `div`'s containing block is the `body`'s box. Thus, the layout of the `p` is dependent on the layout of the `div`, which is in turn dependent on the layout of the `body` element.

And above that in the document tree, the layout of the `body` element is dependent on the layout of the `html` element, whose box creates what is called the *initial containing block*. It's unique in that the viewport—the browser window in screen media, or the printable area of the page in print media—determines the dimensions of the initial containing block, not the size of the content of the root element. This matters because the content can be shorter, and usually longer, than the size of the viewport. Most of the time it doesn't make a difference, but when it comes to things such as fixed positioning or viewport units, the difference is real.

Now that we understand some of the terminology, we can address the properties that make up Figure 6-1. The various margin, border, and padding features, such as `border-style`, can be set using various side-specific long-hand properties, such as `margin-inline-start` or `border-bottom-width`. (The outline properties do not have side-specific properties; a change to an outline property effects all four sides.)

The content's background—a color or tiled image, for example—is applied within the padding and border areas by default, but this can be changed. The margins are always transparent, allowing the background(s) of any parent element(s) to be visible. Padding and borders cannot be of a negative length, but margins can. We'll explore the effects of negative margins later on.

Borders are most often generated using defined styles, with a `border-style` such as `solid`, `dotted`, or `inset`, and their colors are set using the `border-color` property. If no color is set, the value defaults to `currentColor`. Borders can also be generated from images. If a border style has gaps of some type, as with `border-style: dashed` or with a border generated from a partially transparent image, then the element's background is visible through those gaps by default, though it is possible to clip the background to stay inside the border (or the padding).

Altering Element Display

You can affect the way a user agent displays by setting a value for the property `display`

DISPLAY

<table>
<tr><td>Values</td><td>[<display-outside> || <display-inside>] | <display-listitem> | <display-internal> | <display-box> | <display-legacy></td></tr>
<tr><td>Definitions</td><td>See below</td></tr>
<tr><td>Initial value</td><td><code>inline</code></td></tr>
<tr><td>Applies to</td><td>All elements</td></tr>
<tr><td>Computed value</td><td>As specified</td></tr>
<tr><td>Inherited</td><td>No</td></tr>
<tr><td>Animatable</td><td>No</td></tr>
</table>

<display-outside>

`block` | `inline` | `run-in`

<display-inside>

`flow` | `flow-root` | `table` | `flex` | `grid` | `ruby`

<display-listitem>

`list-item` && *<display-outside>*? && [`flow` | `flow-root`]?

<display-internal>

`table-row-group` | `table-header-group` | `table-footer-group` | `table-row` | `table-cell` | `table-column-group` | `table-column` | `table-caption` | `ruby-base` | `ruby-text` | `ruby-base-container` | `ruby-text-container`

<display-box>

`contents` | `none`

<display-legacy>

`inline-block` | `inline-list-item` | `inline-table` | `inline-flex` | `inline-grid`

We're going to ignore the ruby- and table-related values, since they're far too complex for this chapter. We'll also temporarily ignore the value `list-item`, since it's very similar to block boxes and will be explored in detail in XREF HERE. For now, we'll spend a moment talking about how altering an element's display role can alter layout.

Changing Roles

When it comes to styling a document, it's sometimes handy to be able to change the type of box an element generates. For example, suppose we have a series of links in a `nav` that we'd like to lay out as a vertical sidebar:

```
<nav>
    <a href="index.html">WidgetCo Home</a>
    <a href="products.html">Products</a>
    <a href="services.html">Services</a>
    <a href="fun.html">Widgety Fun!</a>
    <a href="support.html">Support</a>
    <a href="about.html" id="current">About Us</a>
    <a href="contact.html">Contact</a>
</nav>
```

By default, the links will generate inline boxes, and thus get sort of mushed together into what looks like a short paragraph of nothing but links. We could put all the links into their own paragraphs or list items, or we could just make them all block-level elements, like this:

```
nav a {display: block;}
```

This will make every `a` element within the navigation element `nav` generate a block box, instead of their usual inline box. If we add on a few more styles, we could have a result like that shown in Figure 6-2.

WidgetCo Home
Products
Services
Widgety Fun!
Support
About Us
Contact

Figure 6-2. Changing the display role from inline to block

Changing display roles can be useful in cases where you want the navigation links to be inline elements if the CSS isn't available (perhaps by failing to load), but to lay out the same links as block-level elements in CSS-aware contexts. You could also present the links as inline on desktop and block on mobile, or vice versa. With the links laid out as blocks, you can style them as you would `div` or `p` elements, with the advantage that the entire element box becomes part of the link.

You may also want to take elements and make them inline. Suppose we have an unordered list of names:

```
<ul id="rollcall">
    <li>Bob C.</li>
    <li>Marcio G.</li>
    <li>Eric M.</li>
    <li>Kat M.</li>
    <li>Tristan N.</li>
    <li>Arun R.</li>
    <li>Doron R.</li>
    <li>Susie W.</li>
```

```
</ul>
```

Given this markup, say we want to make the names into a series of inline names with vertical bars between them (and on each end of the list). The only way to do so is to change their display role. The following rules will have the effect shown in Figure 6-3:

```
#rollcall li {display: inline; border-right: 1px solid; padding: 0 0.33em;}
#rollcall li:first-child {border-left: 1px solid;}
```

| Bob C. | Marcio G. | Eric M. | Kat M. | Tristan N. | Arun R. | Doron R. | Susie W. |

Figure 6-3. Changing the display role from list-item to inline

Be careful to note that you are, for the most part, changing the display role of elements—not changing their inherent nature. In other words, causing a paragraph to generate an inline box does *not* turn that paragraph into an inline element. In HTML, for example, some elements are block while others are inline. While a `span` can easily be placed inside a paragraph, a `span` should not be wrapped around a paragraph.

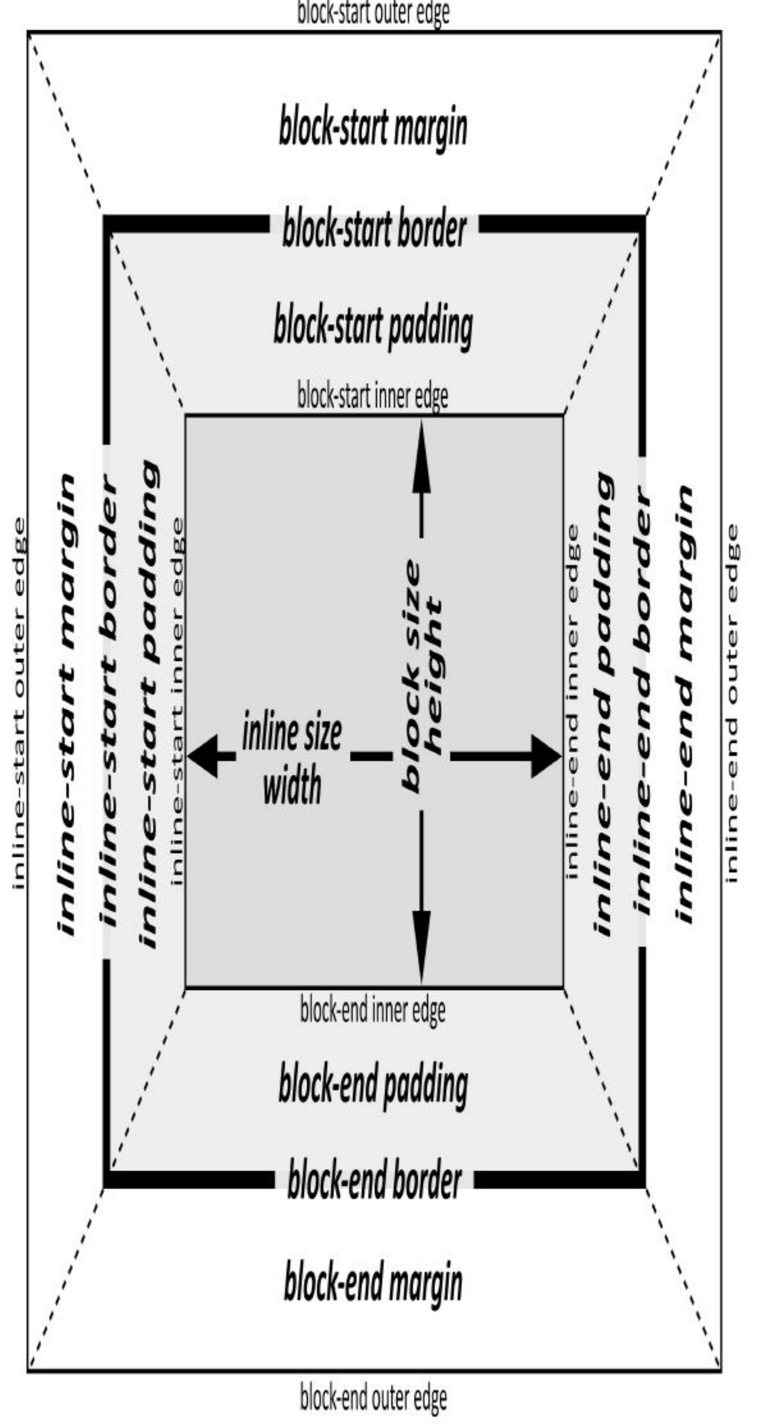
We said “for the most part” because while CSS mostly impacts presentation and not content, CSS properties can impact accessibility in more ways than just color contrast. For example, changing the `display` value can impact how an element is perceived by assistive technologies. Setting an element’s `display` property to `none` removes the element from the accessibility tree. Setting the `display` property on a `<table>` to `grid` may cause the table to be interpreted as something other than a data table, removing normal table keyboard navigation, and making the table inaccessible as a data table to screen reader users. (This shouldn’t happen, but it does in some browsers.)

This can be mitigated by setting the `role` ARIA (Accessible Rich Internet Applications) attribute for the table and all its descendants, but in general, any time a change you make in CSS forces you to make changes in ARIA roles, you should take a moment to consider what you’re doing to see if there isn’t a better way.

Block Boxes

Block boxes behave in predictable, yet sometimes surprising, ways. The handling of box placement along the block and inline axes can differ, for example. In order to fully understand how block boxes are handled, you must clearly understand a number of aspects of these boxes. They are shown in detail in Figure 6-4, which illustrates placement in two different writing modes.

left-to-right, top-to-bottom writing



top-to-bottom, right-to-left writing

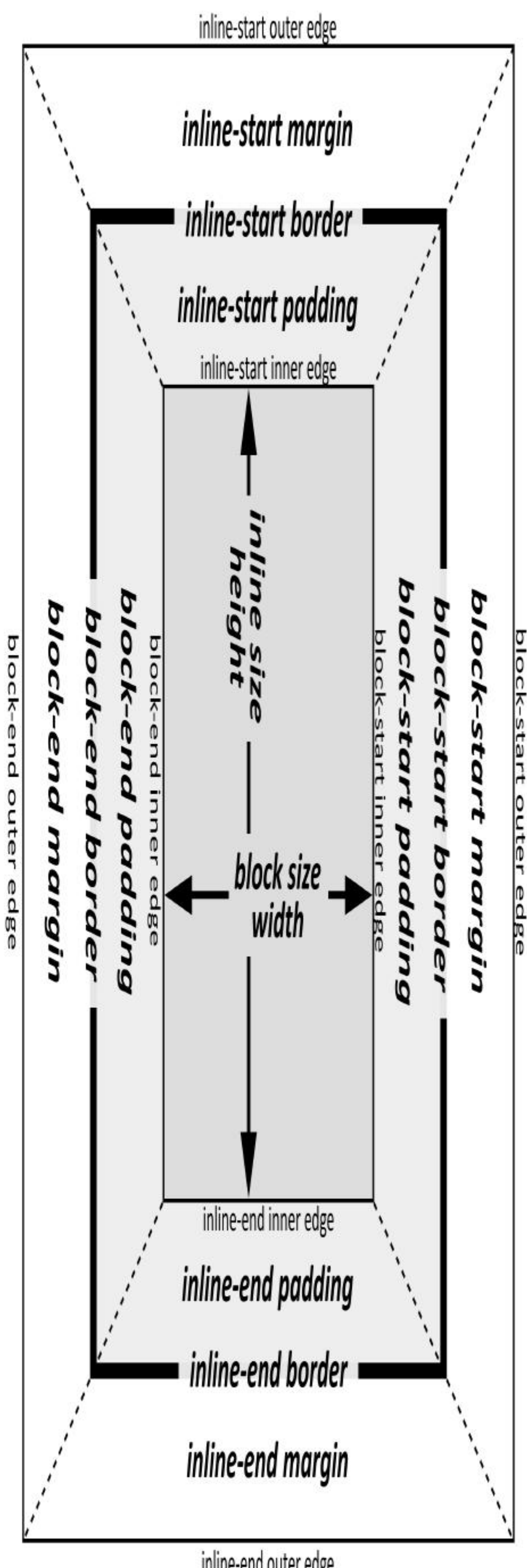


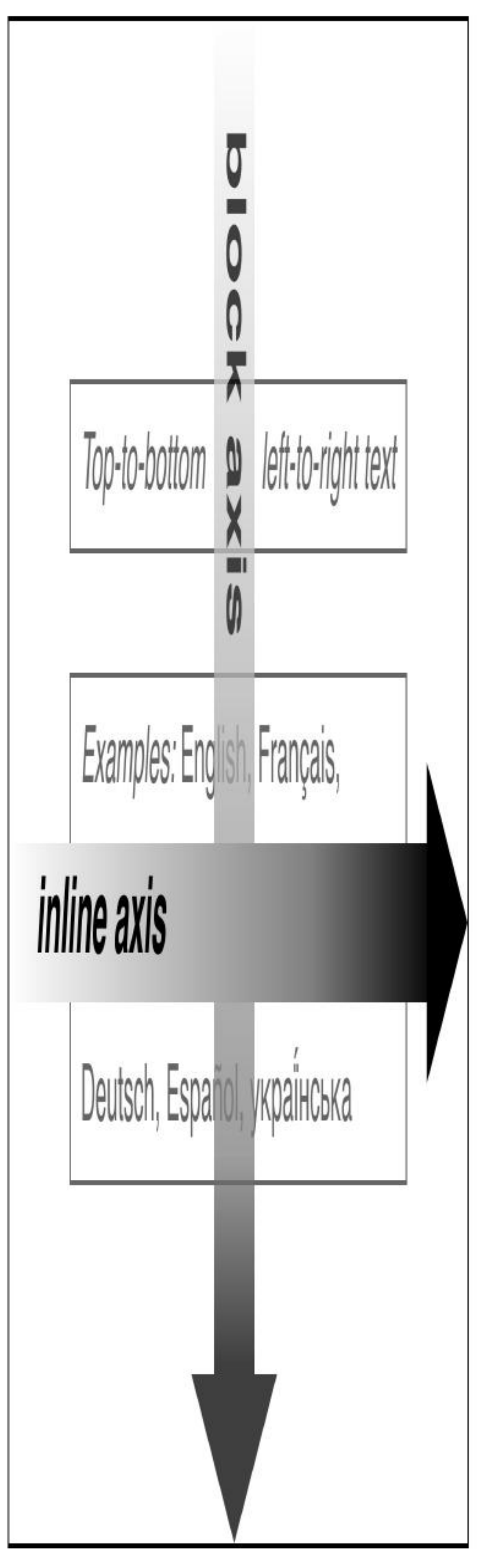
Figure 6-4. The complete box model in two different writing modes

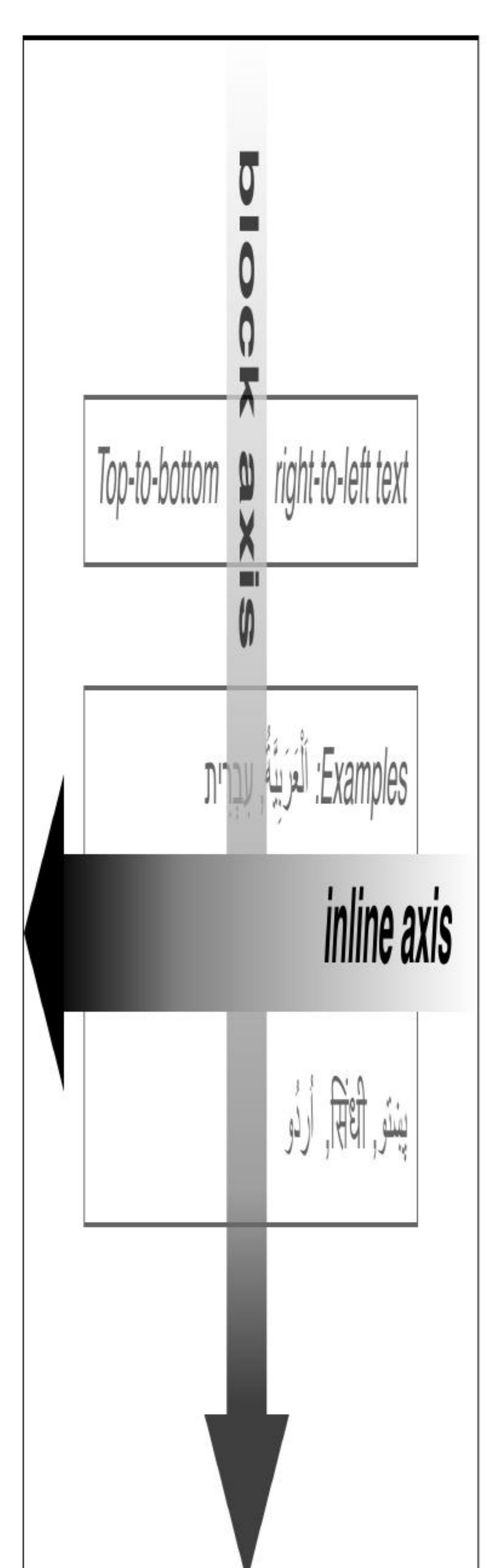
As shown in Figure 6-4, there are block directions and inline directions, and we also have block sizes and inline sizes. Block and inline sizes are descriptions of the size of the content area (by default) along the block and inline axes.

By contrast, the width (sometimes referred to as the “physical width”) of a block box is defined to be the distance between the inner edges of the content area (again, by default) along the horizontal axis (left to right), regardless of the writing direction, and the height (“physical height”) is the distance along the vertical axis (top to bottom). There are properties available to set all these sizes, which we’ll talk about shortly.

Something important to note in Figure 6-4 is the use of “start” and “end” to describe various parts of the element box. For example, there’s a “block-start margin” and a “block-end margin.” The start edge is the edge that you come to first as you move along an axis.

This may be more clear if you look at Figure 6-5 and trace your finger along each axis from arrow tail to tip. As you move along a block axis, the first edge you come to for each element is that element’s block-start edge. As you pass out of the element, you move through the block-end edges. Similarly, as you move along an inline axis, you go through the inline-start edges, across the inline dimension of the content, and then out the inline-end edges. Try it for each of the three examples.


block axis
Top-to-bottom left-to-right text
Examples: English, Français,
Deutsch, Español, українська
inline axis


block axis
Top-to-bottom right-to-left text
Examples:
सिंधी
inline axis

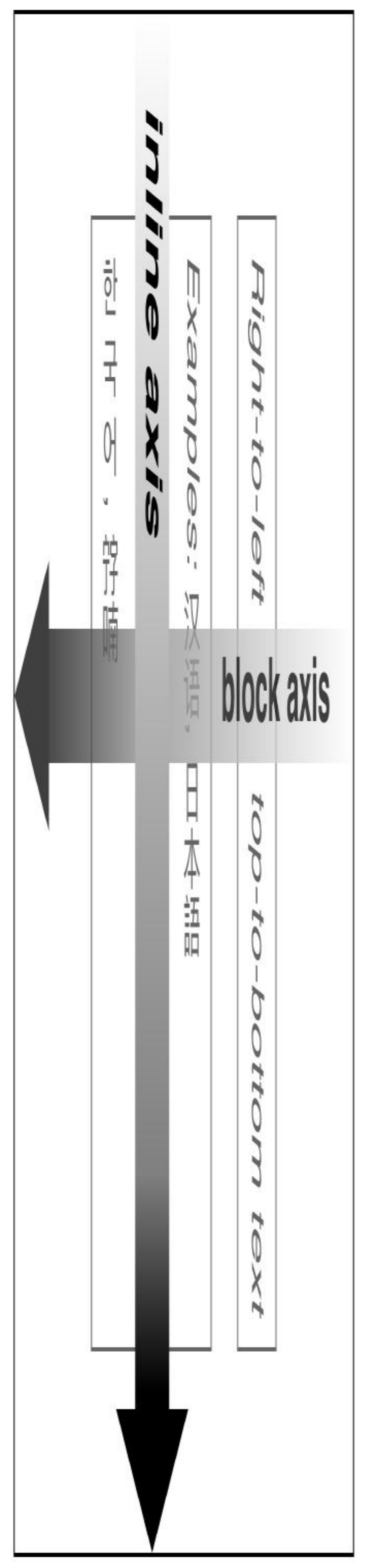

inline axis
Right-to-left top-to-bottom text
Examples: 汉语, 日本語
block axis

Figure 6-5. The block and inline axis directions for three common writing modes

Logical element sizing

Because CSS recognizes block and inline axes for elements, it provides properties that let you set an explicit element size along each axis.

BLOCK-SIZE, INLINE-SIZE

Values	*<length>* | *<percentage>* | `min-content` | `max-content` | `fit-content` | `auto`
Initial value	`auto`
Applies to	All elements except nonreplaced inline elements, table rows, and row groups
Percentages	Calculated with respect to the length of the element's containing block along the block-flow axis (for `block-size`) or inline-flow axis (for `inline-size`)
Computed value	For `auto` and percentage values, as specified; otherwise, an absolute length, unless the property does not apply to the element (then `auto`)
Inherited	No
Animatable	Yes

These properties allow you to set the size of an element along its block axis, or to constrain the lengths of lines of text along the inline axis, regardless of the direction of text flow. If you say `block-size: 500px`, then the element's block size will be five hundred pixels wide, even if that leads to content spilling out of the element box. (We'll discuss that in more detail later in the chapter.)

Consider the following, which has the results shown in Figure 6-6 when applied in various writing modes.

```
p {inline-size: 25ch;}
```

This is a paragraph with some
text. Its inline-axis size has
been set to 25ch.

This is a paragraph with some
text. Its inline-axis size has
been set to 25ch.

been set to 25ch.
text. Its inline-axis size has
This is a paragraph with some

Figure 6-6. Sizing elements along their inline axis

As seen in Figure 6-6, the elements are sized consistently along their inline axis, regardless of the writing direction. If you tilt your head to the side, you can see that the lines wrap in exactly the same places. This yields a consistent line length across all writing modes.

Similarly, you can set a block size for elements. This is used a bit more often for replaced elements like images, but it can be used in any circumstance that makes sense. Take this as an example:

```
p img {block-size: 1.5em;}
```

Having done that, any `img` element found inside a `p` element will have its block size set to one and a half times the size of the surrounding text. (This works on images because they're inline replaced elements; it wouldn't work on inline non-replaced elements.) You could also use `block-size` to constrain the block length of grid layout items to be a minimum or maximum size, such as this:

```
#maingrid > nav {block-size: clamp(2rem, 4em, 25vh);}
```

It should be said that usually, block size is determined automatically, because it's not often that elements in the normal flow have an explicitly-set block size. For example, if an element's block flow is top-to-bottom and it's eight lines long, and each line is an eighth of an inch tall, then its block size will be one inch. If it's 10 lines tall, then the block size is instead 1.25 inches. In either case, as long as the `block-size` is `auto`, the block size is determined by the content of the element, not by the author. This is usually what you want, particularly for elements containing text. When the `block-size` is explicitly set, if there isn't enough content to fill the box, there will be empty space inside the box; if there is more content than can fit, the content may overflow the box or scrollbars may appear.

Content-based sizing values

Beyond the lengths and percentages you saw in the last section for setting block and inline sizes, there are a few keywords that provide content-based sizing:

`max-content`

: Take up the most space possible to fit in the content, even suppressing line-wrapping in the case of text content.

`min-content`

: Take up the least space possible to fit in the content.

`fit-content`

: Take up the amount of space determined by calculating the values of `max-content`, `min-content`, and regular content sizing, taking the maximum of `min-content` and regular sizing, and then taking the minimum of `max-content` and whichever value was the maximum of `min-content` and regular sizing. Yes, that all sounds a bit confusing, but we'll explain it in a moment.

If you've worked at all with CSS Grid (covered in XREF HERE), then you may recognize these

keywords, as they were originally defined as ways to size grid items. Now they're making their way into other areas of CSS. Let's consider the first two keywords, which are demonstrated in Figure 6-7.

This is a paragraph with some text. Its width has been set to `max-content`.

This is a paragraph with some text. Its width has been set to `min-content`.

Figure 6-7. Content sizing

In the first case, the short wide one, that paragraph is set to `max-content`, and that's what happened. The paragraph was made as wide as needed to fit all of the content. It's as narrow as it is only because there isn't much content. Had we added another three sentences, the single line of text would have just kept going and going with no line-wrapping, even if it ran right off the page (or out of the browser window).

For the second case, the content is as narrow as possible without forcing breaks or hyphens inside words. In this particular case, that means the element is *just* wide enough to fit the word "paragraph," which is the longest word in the content. For every other line of text in the example, the browser places as many words as will fit into the space needed for "paragraph," and goes to the next line when it runs out of room. If we added "antidisestablishmentarianism" to the text, then the element would become just wide enough to fit *that* word, and every other line of text would very likely contain multiple words.

Notice that, at the end of the `min-content` example in Figure 6-7, the browser took advantage of the presence of the hyphen in `min-content` to trigger a line-wrap there. Had it not made that choice, then `min-content` would almost certainly have been the longest piece of content in the paragraph, and the element's width would have been set to that length. This means that if your content contains symbols that browsers understand to be natural line-wrapping points (e.g., spaces and hyphens), they'll likely be considered in the `min-content` calculations. If you want to squeeze the element width down even further, you can enable auto-hyphenating of words with the `hyphens` property (see Chapter 11).

For some more examples of `min-content` sizing, see Figure 6-8.

Following is the longest piece of inline content: paragraph.

Following is the longest piece of inline content: schadenfreude.

Following is the longest piece of inline content: antidisestablishmentarianism.

Following is the longest piece of inline content:

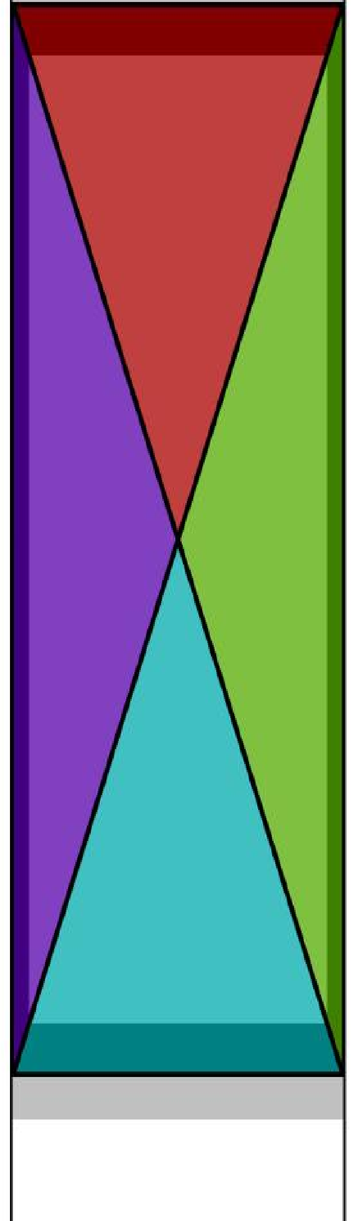


Figure 6-8. Minimum content sizing

The third keyword, `fit-content`, is interesting in that it does its best to fit the element to the content. What that means in practice is that if there is only a little content, the element's inline size (usually its width) will be just big enough to enclose it, as if `max-content` were used. If there's enough content to wrap to multiple lines or otherwise threaten to overflow the element's container, the inline size stops there. This is illustrated in Figure 6-9.

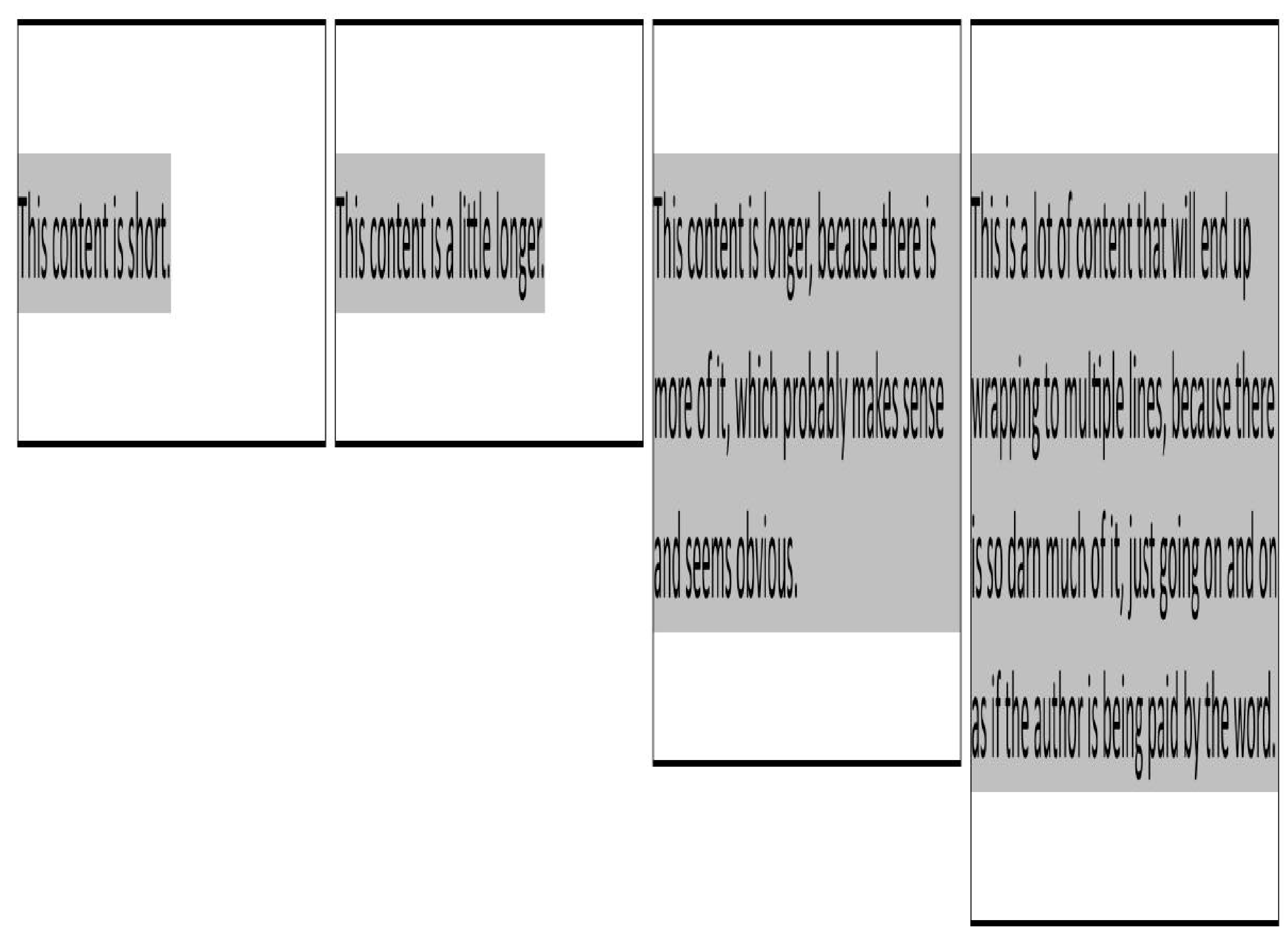


Figure 6-9. Fit-content sizing

In each case, the element is fit to the content without overspilling the element's container. At least, that's what happen with elements in the normal flow. The behavior can be quite different in flexbox and grid contexts, and will be further explored in later chapters.

Minimum and maximum logical sizing

If you'd like to set minimum and maximum bounds on block or inline sizes, there are some properties to help you out.

MIN-BLOCK-SIZE, MAX-BLOCK-SIZE, MIN-INLINE-SIZE, MAX-INLINE-SIZE

Values	Same as for `block-size` and `inline-size`
Initial value	`0`
Applies to	Same as for `block-size` and `inline-size`
Percentages	Same as for `block-size` and `inline-size`
Computed value	Same as for `block-size` and `inline-size`
Inherited	No
Animatable	Yes

These properties can be very useful when you know you want upper and lower bounds on the sizing of an element's box, and are willing to allow the browser to do whatever it wants as long as it obeys those restrictions. As an example, you might set part of a layout like so:

```
main {min-inline-size: min-content; max-inline-size: 75ch;}
```

That keeps the `<main>` element from getting any narrower than the widest bit of inline content, whether that's a long word or an illustration or something else. It also keeps the `<main>` element from getting any wider than around 75 characters, thus keeping line lengths to a readable amount.

It's also possible to set bounds on block sizing. A good example is limiting any image embedded in the normal flow to be its intrinsic size up to a certain point. The following CSS would have the effects shown in Figure 6-10.

```
#cb1 img {max-block-size: 2em;}
#cb2 img {max-block-size: 1em;}
```

#cb1

Litterarum est fairview park woodmere minim insitam newburgh heights emerald necklace, vero at lorem wes craven. Est humanitatis praesent brooklyn heights dynamicus, sollemnes. Eros richmond heights university heights option clari lebron james cleveland facilisi.

#cb2

Litterarum est fairview park woodmere minim insitam newburgh heights emerald necklace, vero at lorem wes craven. Est humanitatis praesent brooklyn heights dynamicus, sollemnes. Eros richmond heights university heights option clari lebron james cleveland facilisi.

Figure 6-10. Maximum block sizing

Height and Width

If you’ve used CSS for a while or are maintaining legacy code, you’re probably used to thinking of “top margin” and “bottom margin.” That’s because, originally, all box model aspects were described in terms of their physical directions: top, right, bottom, and left. You can still work with the physical directions! CSS has simply added new, more text-aware directions to the mix.

If you were to change `inline-size` to `width` in the previous code example, then you’d get a result more like that shown in Figure 6-11 (in which the vertical writing modes are clipped off well short of their full height).

This is a paragraph with some text. Its width (*not* inline-axis) has been set to 25ch.

This is a paragraph with some te

This is a paragraph with some te

Figure 6-11. Sizing elements' width

In Figure 6-11, the elements are made `40ch` wide horizontally, regardless of their writing mode. Each element's height has been automatically determined by the content, the specifics of the writing mode, and so on.

> **TIP**
>
> When you use block and inline properties like `block-size` instead of physical directions like `height`, should your design be applied to content translated to other languages, the layout will automatically adjust to your intentions.

HEIGHT, WIDTH

Values	<*length*> | <*percentage*> | `min-content` | `max-content` | `fit-content` | `auto`
Initial value	`auto`
Applies to	All elements except nonreplaced inline elements, table rows, and row groups
Percentages	Calculated with respect to the vertical height (for `height`) or horizontal width (for `width`) of the containing block; for `height`, set to `auto` if the `height` of its containing block is `auto`
Computed value	For `auto` and percentage values, as specified; otherwise, an absolute length, unless the property does not apply to the element (then `auto`)
Inherited	No
Animatable	Yes

`height` and `width` are what's known as *physical properties*. This means they refer to physical directions, as opposed to the writing-dependent directions of block size and inline size. Thus, `height` really does refer to the distance from the top to the bottom of the element's inner edge, regardless of the direction of the block axis.

In writing with a horizontal inline axis, such as English or Arabic, if both `inline-size` and `width` are set on the same element, the one declared later will take precedence over the first one declared. The same is true if `block-size` and `height` are both declared; if origin, layer, and specifity are the same, the one declared last takes precedence. In vertical writing modes, `inline-size` corresponds to `height`, and `block-size` to `width`.

Setting a block box's height or width as a <*length*> means it will be that length tall or wide, regardless of the content within it. If you set an element that generates a block box to `width: 200px`, then it will be 200 pixels wide, even if it has a 500-pixel-wide image inside it.

Setting the value of `width` to a <*percentage*> means the width of the element will be that percentage of its containing block's width. If you set a paragraph to `width: 50%` and its containing block is 1,024 pixels wide, then the paragraph's `width` will be computed to 512 pixels.

Things are similar for `height`, except this only works if the containing block has an explicitly set height. If the containing block's height is automatically set, then a percentage value is treated as `auto` instead, as seen in the `#cb4` example in Figure 6-12.

> **NOTE**
>
> The handling of `auto` top and bottom margins is different for positioned elements, as well as flexible-box and grid elements. The differences will be covered in the chapters on those topics.

Here are some examples of these values and combinations, with the result shown in Figure 6-12.

```
[id^="cb"] {border: 1px solid;}  /* "cb" for "containing block" */
#cb1 {width: auto;}    #cb1 p {width: auto;}
#cb2 {width: 400px;}   #cb2 p {width: 300px;}
#cb3 {width: 400px;}   #cb3 p {width: 50%;}

#cb4 {height: auto;}   #cb4 p {height: 50%;}
#cb5 {height: 300px;}  #cb5 p {height: 200px;}
#cb6 {height: 300px;}  #cb6 p {height: 50%;}
```

#cb1

This is a paragraph with some text that will wrap to multiple lines if given half a chance. It also determines the width and height of the element if it isn't otherwise set.
(CB: width auto, P: width auto)

#cb2

This is a paragraph with some text that will wrap to multiple lines if given half a chance. It also determines the width and height of the element if it isn't otherwise set.
(CB: width 400px, P: width 300px)

#cb3

This is a paragraph with some text that will wrap to multiple lines if given half a chance. It also determines the width and height of the element if it isn't otherwise set.
(CB: width 400px, P: width 50%)

#cb4

This is a paragraph with some text that will wrap to multiple lines if given half a chance. It also determines the width and height of the element if it isn't otherwise set.
(CB: height auto, P: height 50%)

#cb5

This is a paragraph with some text that will wrap to multiple lines if given half a chance. It also determines the width and height of the element if it isn't otherwise set.
(CB: height 300px, P: height 200px)

#cb6

This is a paragraph with some text that will wrap to multiple lines if given half a chance. It also determines the width and height of the element if it isn't otherwise set.
(CB: height 300px, P: height 50%)

Figure 6-12. Heights and widths

You can also use `max-content` and `min-content` with the `height` property, but in top-to-bottom block flows, both are same as `height: auto`. In writing modes where the block axis is horizontal, then setting these values for `height` will have similar effects as setting them for `width` in vertical block flows.

Another important note: these properties don't apply to inline nonreplaced elements. For example, if you try to declare a `height` and `width` for a hyperlink that's in the normal flow and generates an inline box, CSS-conformant browsers *must* ignore those declarations. Assume the following rules:

```
a:any-link {color: red; background: silver; height: 15px; width: 60px;}
```

You'll end up with red unvisited links on silver backgrounds whose height and width are determined by the content of the links. The links will *not* have content areas that are 15 pixels tall by 60 pixels wide, as these must be ignored when applied to inline non-replaced element. If, on the other hand, you add a `display` value, such as `inline-block` or `block`, then `height` and `width` *will* set the height and width of the links' content areas.

Altering box sizing

If it seems little weird to use `height` and `width` (and `block-size` and `inline-size`) to describe the sizing of the element's content area instead of its visible area, you can make things more intuitive by using the property `box-sizing`.

BOX-SIZING

Values	`content-box` | `border-box`
Initial value	`content-box`
Applies to	All elements that accept `width` or `height` values
Computed value	As specified
Inherited	No
Animatable	No

This property changes what the values of the `height`, `width`, `block-size`, and `inline-size` properties actually do.

`box-sizing` changes what the values of the `block-size` and `inline-size` properties actually do. If you declare `inline-size: 400px` and don't declare a value for `box-sizing`, then the element's content area will be 400 pixels in the inline direction and any padding, borders, and so on will

be added to that. If, on the other hand, you declare `box-sizing: border-box`, then the element box will be 400 pixels from the inline-start border edge to the inline-end border edge; any inline-start or -end border or padding will be placed within that distance, thus shrinking the inline size of the content area. This is illustrated in Figure 6-13.

Each div has the following styles:

block-size: 200px;
inline-size: 400px;
padding: 25px;
border: 5px solid gray;

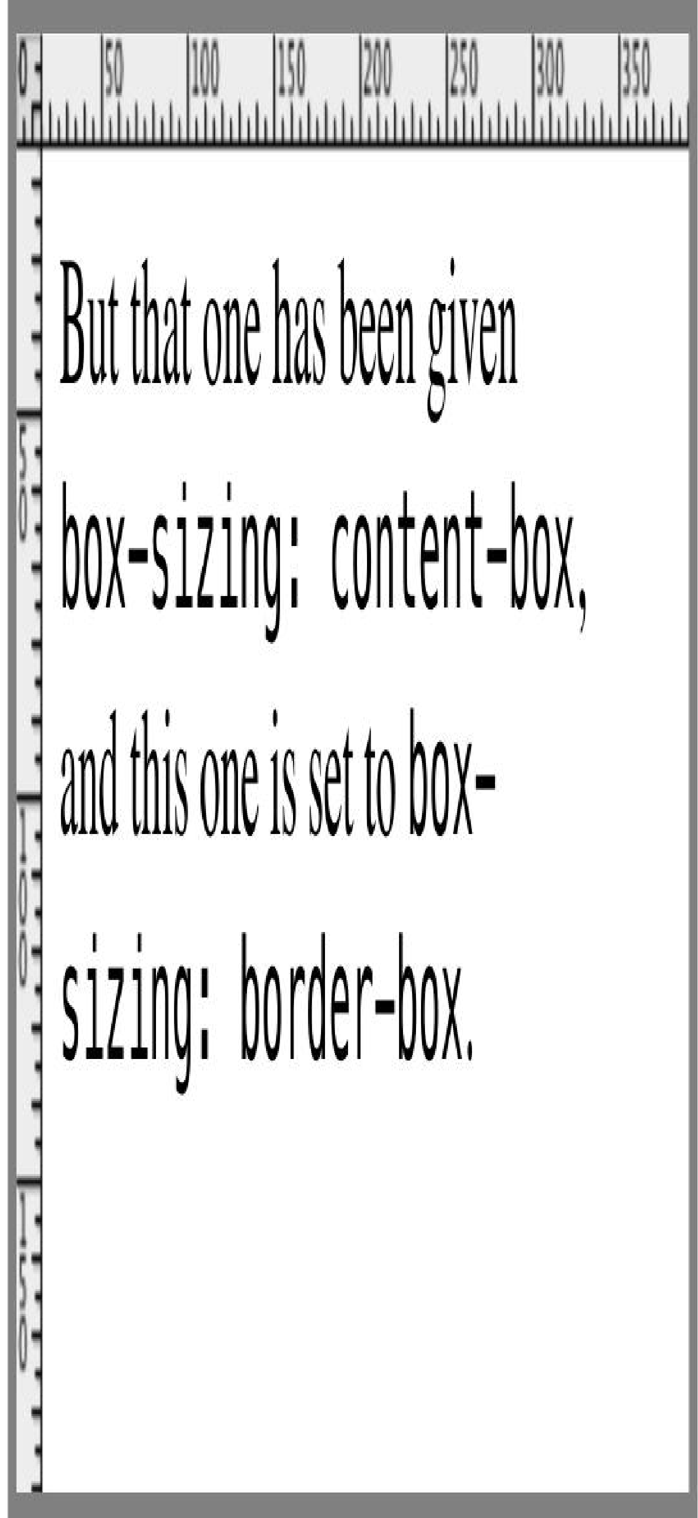


Figure 6-13. The effects of box-sizing

Put another way, if you declare `width: 400px` and don't declare a value for `box-sizing`, then the element's content area will be 400 pixels wide and any padding, borders, and so on will be added to that. If, on the other hand, you declare `box-sizing: border-box`, then the element box will be 400 pixels from the left outer border edge to the right outer border edge; any left or right border or padding will be placed within that distance, thus shrinking the width of the content area (again, as seen in Figure 6-13).

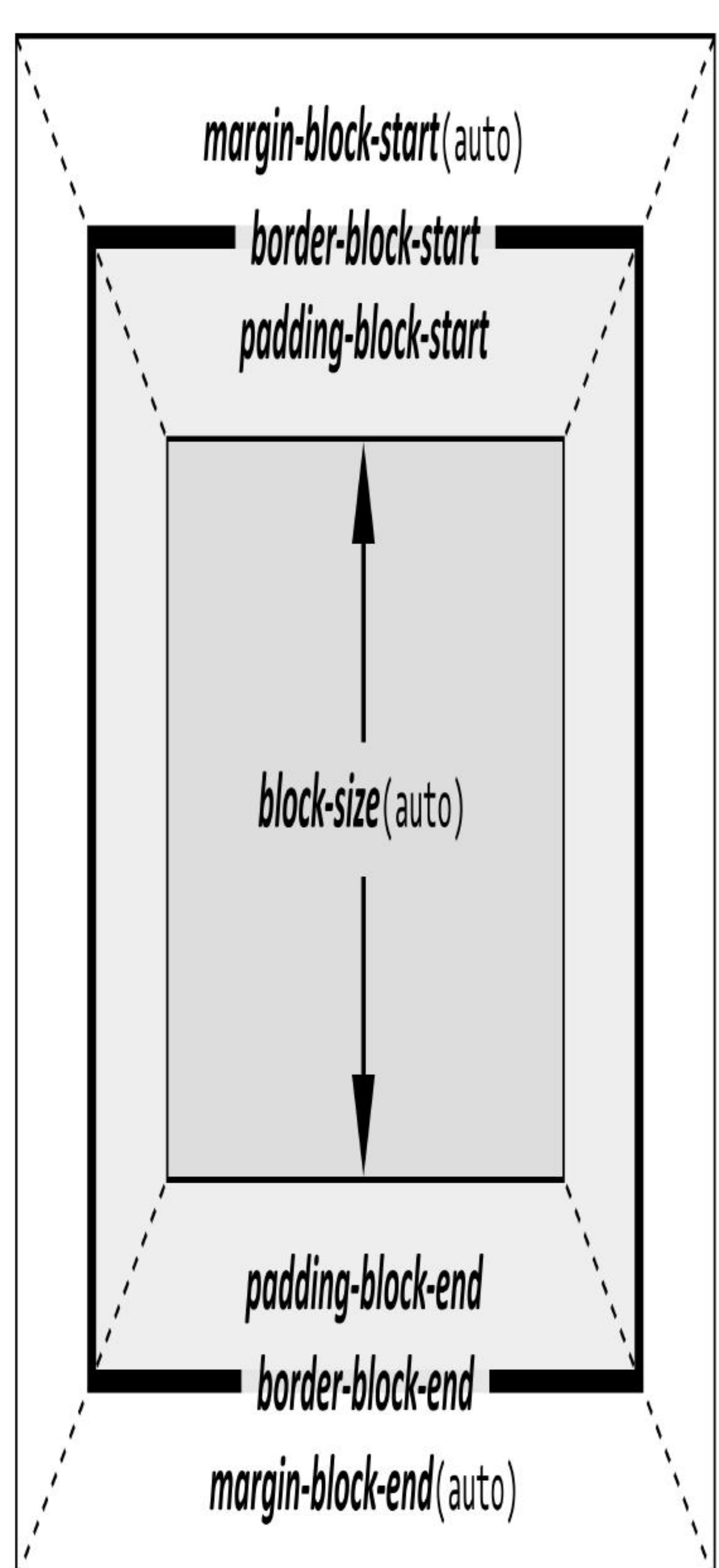
We're talking about the `box-sizing` property here because, as stated, it applies to "all elements that accept `width` or `height` values" (because it was defined before logical properties were commonplace). That's most often elements generating block boxes, though it also applies to replaced inline elements like images, as well as inline-block boxes.

Having established how to size elements in both logical and physical ways, let's widen our scope and look at all the properties that affect block sizing.

Block-Axis Properties

In total, block-axis formatting is affected by seven related properties: `margin-block-start`, `border-block-start`, `padding-block-start`, `height`, `padding-block-end`, `border-block-end`, and `margin-block-end`. These properties are diagrammed in Figure 6-14. These properties will all be covered in detail in Chapter 7; here, we will talk about the general principles and behavior of these properties before looking at the details of their values.

The block-start and -end padding and borders must be set to specific values, or else they default to a width of zero, assuming no border style is declared. If `border-style` has been set, then the thickness of the borders is set to be `medium`, which is set to three pixels wide in all known browsers. Figure 6-14 provides an illustration in two different writing modes for remembering which parts of the box may have a value of `auto` and which may not.



margin-block-start(auto)
border-block-start
padding-block-start
block-size(auto)
padding-block-end
border-block-end
margin-block-end(auto)

Figure 6-14. The seven properties of block-axis formatting, and which of them can be set to `auto`

Interestingly, if either `margin-block-start` or `margin-block-end` is set to `auto` for a block box in the normal flow, but not both, they both evaluate to `0`. A value of `0`, unfortunately, prevents easy block-direction centering of normal-flow boxes in their containing blocks (though such centering is fairly straightforward in flex or grid layout).

`block-size` must be set to `auto` or to a nonnegative value of some type; it can never be less than zero.

Auto block sizing

In the simplest case, a normal-flow block box with `block-size: auto` is rendered just tall enough to enclose the line boxes of its inline content (including text). If an auto-block-size, normal-flow block box has only block-level children and has no block-edge padding or borders, the distance from its first child's border-start edge to its last child's border-end edge will be the box's block size. This is the case because the margins of the child elements can "stick out" of the element that contains them thanks to what's known as *margin collapsing*, which we'll talk about later.

However, if a block-level element has either block-start or -end padding, or block-start and -end borders, then its block size will be the distance from the block-start margin edge of its first child to the block-end margin edge of its last child:

```
<div style="block-size: auto;
    background: silver;">
    <p style="margin-block-start: 2em; margin-block-end: 2em;">A paragraph!</p>
</div>
<div style="block-size: auto; border-block-start: 1px solid;
     border-block-end: 1px solid; background: silver;">
    <p style="margin-block-start: 2em; margin-block-end: 2em;">
        Another paragraph!</p>
</div>
```

Both of these behaviors are demonstrated in Figure 6-15.

If we changed the borders in the previous example to padding, the effect on the block size of the `div` would be the same: it would still enclose the paragraph's margins within it.

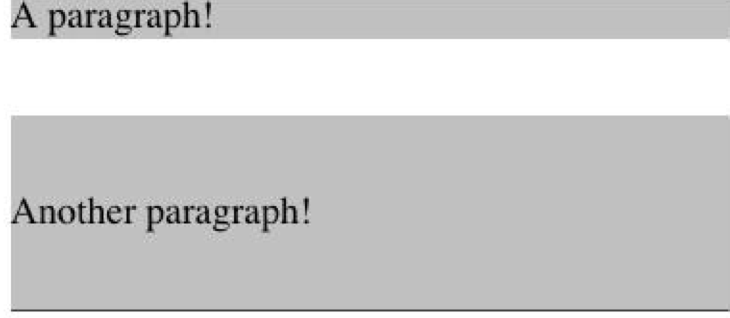


Figure 6-15. Auto block sizes with block-level children

Percentage Heights

We saw earlier how length-value block sizes are handled, so let's spend a moment on percentages. If the block size of a normal-flow block box is set to a percentage value, then that value is taken as a percentage of the block size of the box's containing block, assuming the container has an explicit, non-`auto` block size of its own. Given the following markup, the paragraph will be 3 em long along the block axis:

```
<div style="block-size: 6em;">
    <p style="block-size: 50%;">Half as tall</p>
</div>
```

In cases where the block size of the containing block is *not* explicitly declared, percentage block sizes are reset to `auto`. If we changed the previous example so that the `block-size` of the `div` is `auto`, the paragraph will now have its block size determined automatically

```
<div style="block-size: auto;">
    <p style="block-size: 50%;">NOT half as tall; block size reset to auto</p>
</div>
```

These two possibilities are illustrated in Figure 6-16. (The spaces between the paragraph borders and the `div` borders are the block-start and -end margins on the paragraphs.)

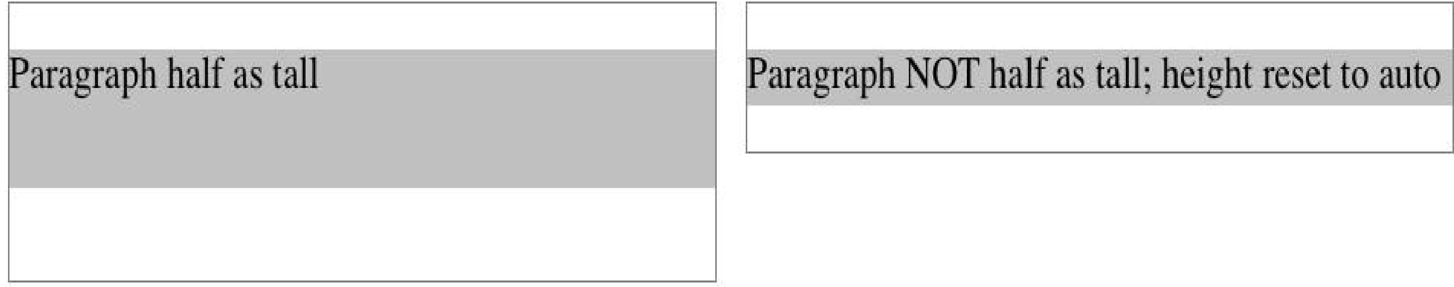


Figure 6-16. Percentage block sizes in different circumstances

Before we move on, take a closer look at the first example in Figure 6-16, the half-as-tall paragraph. It may be half as tall, but it isn't centered along the block axis. That's because the containing `div` is 6 em tall, which means the half-as-tall paragraph is 3 em tall. It has block-start and -end margins of 1 em thanks to the browser's default styles, so its overall block size is 5 em. That means there is actually 2 em of space between the block end of the paragraph's visible box and the `div`'s block-end border, not 1 em. This is illustrated in detail in Figure 6-17.

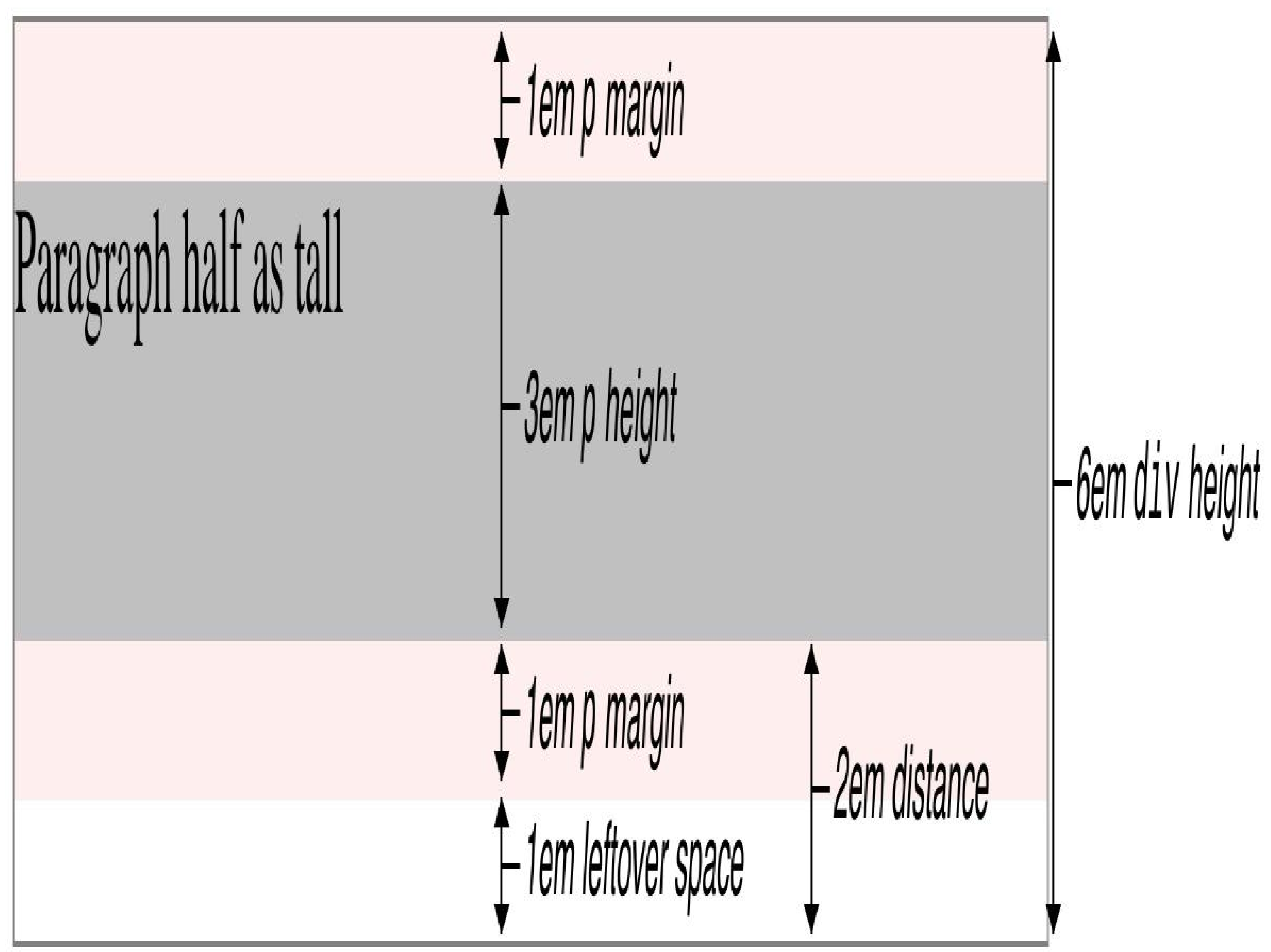


Figure 6-17. Block-axis sizing and placement in detail

Handling Content Overflow

Given that it’s possible to set elements to be specific sizes, it becomes possible to make an element too small for its content to fit inside. This is more likely to arise if block sizes are explicitly defined, but it can also happen with inline sizes, as we’ll see in later sections. If this sort of thing does happen, you can exert some control over the situation with the `overflow` shorthand property.

OVERFLOW

Values	[`visible` | `hidden` | `clip` | `scroll` | `auto`]{1,2}
Initial value	`visible`
Applies to	Block-level and replaced elements
Computed value	As specified
Inherited	No
Animatable	No

The default value of `visible` means that the element's content may be visible outside the element's box. Typically, this leads to the content running outside its own element box, but not altering the shape of that box. The following markup would result in Figure 6-18:

```
div#sidebar {block-size: 7em; background: #BBB; overflow: visible;}
```

If `overflow` is set to `hidden`, the element's content is clipped at the edges of the element box. With the `hidden` value, there is no way to get at the parts of the content that are clipped off.

If `overflow` is set to `clip`, the element's content is also clipped—that is, hidden—at the edges of the element box, with no way to get at the parts that are clipped off, *except* via programmatic means such as JavaScript's `HTMLElement.offsetLeft` property. This forces the designer to build their own mechanisms for scrolling or panning content to make the clipped-off content available.

If `overflow` is set to `scroll`, the overflowing content is clipped, but the content can be made available to the user via scrolling methods, including a scroll bar (or set of them). One possibility is depicted in Figure 6-18.

If `scroll` is used, the panning mechanisms (e.g., scroll bars) should always be rendered. To quote the specification, "this avoids any problem with scrollbars appearing or disappearing in a dynamic environment." Thus, even if the element has sufficient space to display all its content, the scroll bars may still appear and take up space (though they may not).

In addition, when printing a page or otherwise displaying the document in a print medium, the content may be displayed as though the value of `overflow` were declared to be `visible`.

Figure 6-18 illustrates these `overflow` values, with two of them combined in a single example.

visible

This is an element that's only 7em tall. Any overflowing content will be visible outside the boundaries of the element box, whether or not that's what the author really wanted.

hidden / clip

This is an element that's only 7em tall. Any overflowing content won't be visible outside the boundaries of the element box, whether

scroll

This is an element that's only 7em tall. Any overflowing content won't be visible outside the boundaries of the

Figure 6-18. Methods for handling overflowing content

Finally, there is `overflow: auto`. This allows UAs (user agents)_ to determine which of the previously-described behaviors to use, although UAs are encouraged to provide a scrolling mechanism whenever necessary. This is a potentially useful way to use overflow, since user agents could interpret it to mean "provide scroll bars only when needed." (They may not, but generally do.)

Single-Axis Overflow

Two properties make up the `overflow` shorthand. You can define the overflow behavior along the X (horizontal) and Y (vertical) directions separately, either by setting them both in `overflow`, or by using the `overflow-x` and `overflow-y` properties.

OVERFLOW-X, OVERFLOW-Y

Values	`visible` | `hidden` | `clip` | `scroll` | `auto`
Initial value	`visible`
Applies to	Block-level and replaced elements
Computed value	As specified
Inherited	No
Animatable	No

By setting the overflow behavior separately along each axis, you're essentially deciding where scrollbars will appear, and where they won't. Consider the following, which is rendered in Figure 6-19.

```
div.one   {overflow-x: scroll; overflow-y: hidden;}
div.two   {overflow-x: hidden; overflow-y: scroll;}
div.three {overflow-x: scroll; overflow-y: scroll;}
```

```
overflow-x: scroll;     overflow-x: hidden;     overflow-x: scroll;
overflow-y: hidden;     overflow-y: scroll;     overflow-y: scroll;
```

This is an element that's only 7em tall. Any overflowing content won't be visible outside the boundaries of the element box, whether

Figure 6-19. Setting overflow separately for X and Y

In the first case, there is an empty scrollbar set up for the X (horizontal) axis, but none for the Y (vertical) axis, even though the content overflowed along the Y axis. This is the worst of both worlds: a scrollbar that's empty because it isn't needed, and no scroll bar where it is needed.

The second case is the much more useful inverse: there is no scrollbar set along the X axis, but one is available for the Y axis, so the overflowed content can be accessed by means of scrolling.

In the third case, where `scroll` was set for both axes, there is access to the overflowing content via scrolling, but also an unnecessary scrollbar (which is empty) for the X axis. This is equivalent to simply declaring `overflow: scroll`.

Which brings us to the true nature of `overflow`: it's a shorthand property that brings `overflow-x` and `overflow-y` together under one roof. The following is exactly equivalent to the previous example, and will have the same result shown in Figure 6-19.

```
div.one   {overflow: scroll hidden;}
div.two   {overflow: hidden scroll;}
div.three {overflow: scroll;} /* 'scroll scroll' would also work */
```

As you see, you can give `overflow` two keywords, which are always in the order X, then Y. If only one value is given, then it's used for both the X and Y axes. This is why `scroll` and `scroll scroll` are the same thing, as values of `overflow`. Similarly, `hidden` would be the same as saying `hidden hidden`.

Negative Margins and Collapsing

Believe it or not, negative margins are possible. The base effect is to move the margin-edge inward toward the center of the element's box. Consider:

```
p.neg {margin-block-start: -50px; margin-block-end: 0;
    border: 3px solid gray;}

<div style="width: 420px; background-color: silver; padding: 10px;
            margin-block-start: 50px; border: 1px solid;">
    <p class="neg">
        A paragraph.
    </p>

    A div.

</div>
```

As we see in Figure 6-20, the paragraph has been pulled upward by its negative block-start margin. Note that the content of the `<div>` that follows the paragraph in the markup has also been pulled up the block axis by 50 pixels.

Figure 6-20. The effects of a negative top margin

Now compare the following markup to the situation shown in Figure 6-21:

```
p.neg {margin-block-end: -50px; margin-block-end: 0;
    border: 3px solid gray;}
```

```
<div style="width: 420px; margin-block-start: 50px;">
    <p class="neg">
        A paragraph.
    </p>
</div>
<p>
    The next paragraph.
</p>
```

Figure 6-21. The effects of a negative block-end margin

What’s happening in Figure 6-21? The elements following the `div` are placed according to the location of the block-end margin edge of the `div`, which is 50px higher than it would be without the negative margin. As Figure 6-21 shows, the block-end of the `div` is actually *above* the visual block-end of its child paragraph. The next element after the `div` is the appropriate distance from the block-end of the `div`.

Collapsing Block Axis Margins

An important aspect of block-axis formatting is the *collapsing* of adjacent margins, which is a way of comparing adjacent margins in the block direction, and then using only the largest of those margins to set the distance between the adjacent block elements. Note that collapsing behavior applies only to margins. Padding and borders never collapse.

An unordered list, where list items follow one another along the block axis, is a perfect environment for studying margin collapsing. Assume that the following is declared for a list that contains five items:

```
li {margin-block-start: 10px; margin-block-end: 15px;}
```

Each list item has a 10-pixel block-start margin and a 15-pixel block-end margin. When the list is rendered, however, the visible distance between adjacent list items is 15 pixels, not 25. This happens because, along the block axis, adjacent margins are collapsed. In other words, the smaller of the two margins is eliminated in favor of the larger. Figure 6-22 shows the difference between collapsed and uncollapsed margins.

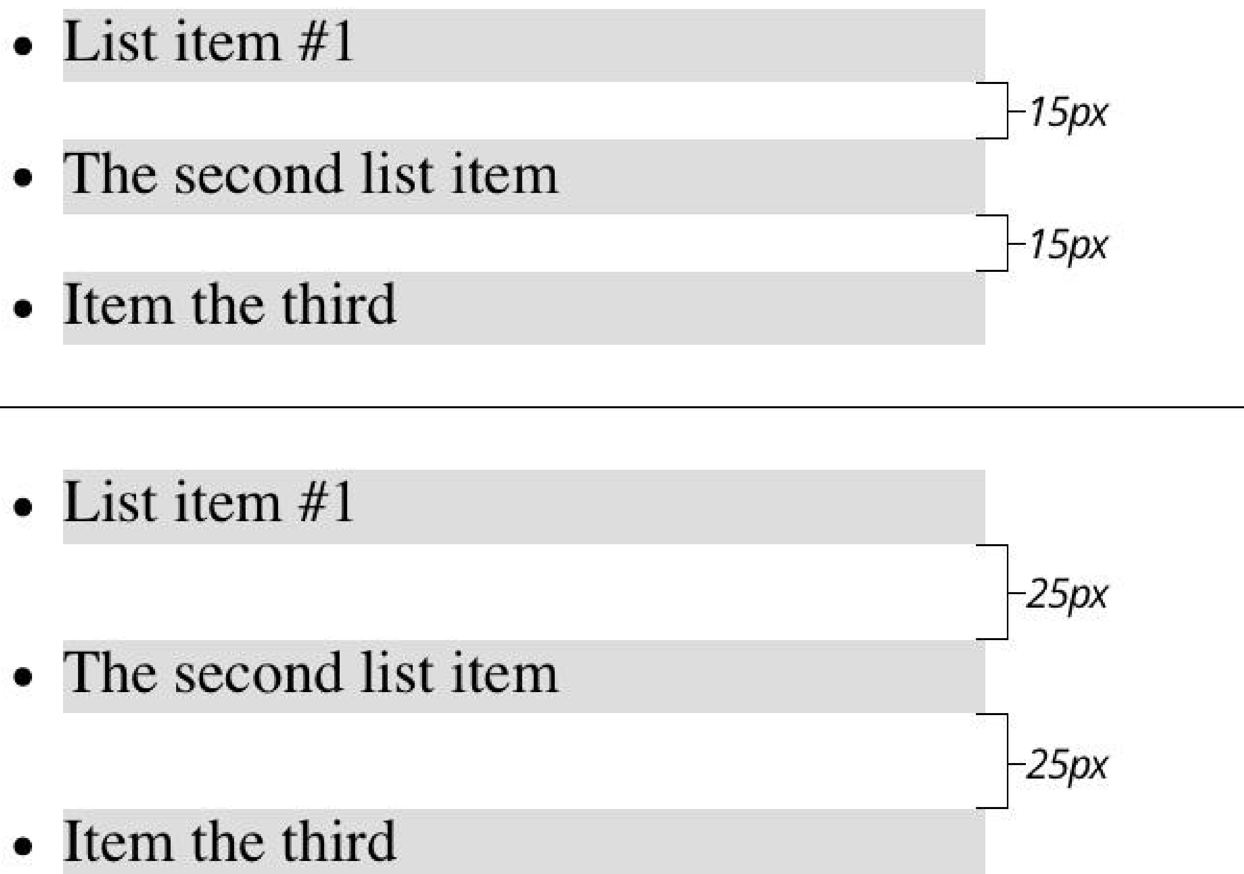


Figure 6-22. Collapsed versus uncollapsed margins

User agents will collapse block-adjacent margins as shown in the first list in Figure 6-22, where there are 15-pixel spaces between each list item. The second list shows what would happen if browsers didn't collapse margins, resulting in 25-pixel spaces between list items.

Another word to use, if you don't like "collapse," is "overlap." Although the margins are not really overlapping, you can visualize what's happening using the following analogy.

If you prefer visual analogies, imagine that each element, such as a paragraph, is a small piece of paper with the content of the element written on it. Around each piece of paper is some amount of clear plastic, which represents the margins. The first piece of paper (say an `h1` piece) is laid down on the canvas. The second (a paragraph) is laid below it along the block axis and then slid upwards along that axis until the edge of one piece's plastic touches the edge of the other's paper. If the first piece of paper has half an inch of plastic along its block-end edge, and the second has a third of an inch along its block-start, then when they slide together, the first piece's block-end plastic will touch the block-start edge of the second piece of paper. The two are now done being placed on the canvas, and the plastic attached to the pieces is overlapping.

Collapsing also occurs where multiple margins meet, such as at the end of a list. Adding to the earlier example, let's assume the following rules apply:

```
ul {margin-block-end: 15px;}
li {margin-block-start: 10px; margin-block-end: 20px;}
h1 {margin-block-start: 28px;}
```

The last item in the list has a block-end margin of 20 pixels, the block-end margin of the `ul` is 15 pixels, and the block-start margin of a succeeding `h1` is 28 pixels. So once the margins have been collapsed, the distance between the end of the last `li` in the list and the beginning of the `h1` is 28 pixels, as shown in Figure 6-23.

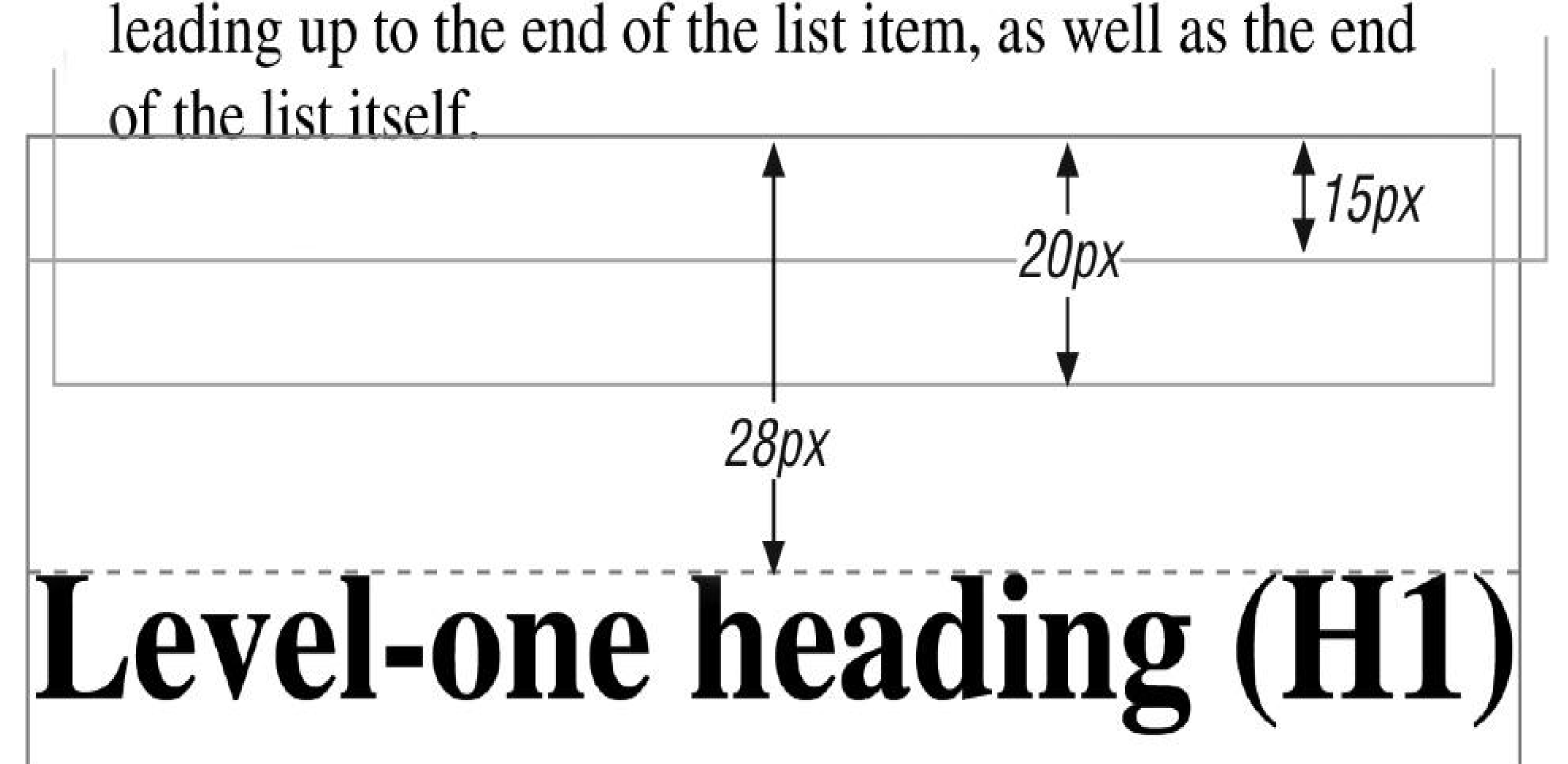


Figure 6-23. Collapsing in detail

If you add a border or padding to a containing block, this causes the margins of its child elements to be entirely contained within it. We can see this behavior in operation by adding a border to the `ul` element in the previous example:

```
ul {margin-block-end: 15px; border: 1px solid;}
li {margin-block-start: 10px; margin-block-end: 20px;}
h1 {margin-block-start: 28px;}
```

With this change, the block-end margin of the `li` element is now placed inside its parent element (the `ul`). Therefore, the only margin collapsing that takes place is between the `ul` and the `h1`, as illustrated in Figure 6-24.

- A list item.
- Another list item.

A Heading-1

Figure 6-24. Collapsing (or not) with borders added to the mix

Negative margin collapsing is slightly different. When a negative margin participates in margin collapsing, the browser takes the absolute value of the negative margin and subtracts it from any adjacent positive margins. In other words, the negative length is added to the positive length(s), and the resulting value is the distance between the elements, even if that distance is a negative length. Figure 6-25 provides some concrete examples.

This paragraph has a bottom margin of `0` (zero).
This paragraph has a top margin of `0` (zero).

This paragraph has a bottom margin of `0.75em` (three-quarters of an em).
This paragraph has a top margin of `-1em` (negative one em).

This paragraph has a bottom margin of `-0.75` (negative three quarters of an em).
This paragraph has a top margin of `0` (zero).

Figure 6-25. Examples of negative block-axis margins

Now let’s consider an example where the margins of a list item, an unordered list, and a paragraph are all collapsed. In this case, the unordered list and paragraph are assigned negative margins:

```
li {margin-block-end: 20px;}
ul {margin-block-end: -15px;}
```

```
h1 {margin-block-start: -18px;}
```

The negative margin of the greatest magnitude (`-18px`) is added to the largest positive margin (`20px`), yielding `20px` - `18px` = `2px`. Thus, there are only two pixels between the block-end of the list item's content and the block-start of the `h1`'s content, as we can see in Figure 6-26.

Figure 6-26. Collapsing margins and negative margins, in detail

When elements overlap each other due to negative margins, it's hard to tell which elements are on top of others. You may also have noticed that very few of the examples in this section use background colors. If they did, the background color of a following element might overwrite the content of a preceding element. This is expected behavior, since browsers usually render elements in order from beginning to end, so a normal-flow element that comes later in the document can generally be expected to overwrite an earlier element, assuming the two end up overlapping.

Inline-Axis Formatting

Laying out elements along the inline axis can be more complex than you'd think. Part of the complexity has to do with the default behavior of `box-sizing`. With the default value of `content-box`, the value given for `inline-size` affects the inline width of the content area, *not* the entire visible element box. Consider the following example, where the inline axis runs left to right:

```
<p style="inline-size: 200px;">wideness?</p>
```

This makes the paragraph's content area 200 pixels wide. If we give the element a background, this will be quite obvious. However, any padding, borders, or margins you specify are *added* to the width value. Suppose we do this:

```
<p style="inline-size: 200px; padding: 10px; margin: 20px;">wideness?</p>
```

The visible element box is now 220 pixels in inline size, since we've added 10 pixels of padding to every side of the content. The margins will now extend another 20 pixels to both inline sides for an overall element inline size of 260 pixels. This is illustrated in Figure 6-27.

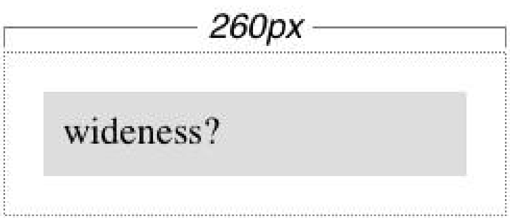


Figure 6-27. Additive padding and margin

If we change the styles to use `box-sizing: border-box`, then the results will be different. In that case, the visible box will be 200 pixels wide along the inline axis with a content inline size of 180 pixels, and a total of 40 pixels of margin on the inline sides, giving an overall box inline size of 240 pixels, as illustrated in Figure 6-28.

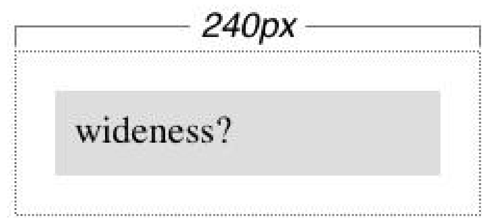


Figure 6-28. Subtractive padding

In either case, there is a rule in the CSS specification that says the sum of the inline components of a block box in the normal flow always equals the inline size of the containing block (which is why, as we'll see in just a bit, `margin: auto` centers content in the inline direction). Let's consider two paragraphs within a `div` whose margins have been set to be `1em`, and whose `box-sizing` value is the default `content-box`. The content size (the value of `inline-size`) of each paragraph in this example, plus its inline-start and -end padding, borders, and margins, will always add up to the inline size of the `div`'s content area.

Let's say the inline size of the `div` is `30em`. That makes the sum total of the content size, padding, borders, and margins of each paragraph 30 em. In Figure 6-29, the "blank" space around the paragraphs is actually their margins. If the `div` had any padding, there would be even more blank space, but that isn't the case here.

Figure 6-29. Element boxes are as wide as the inline width of their containing block

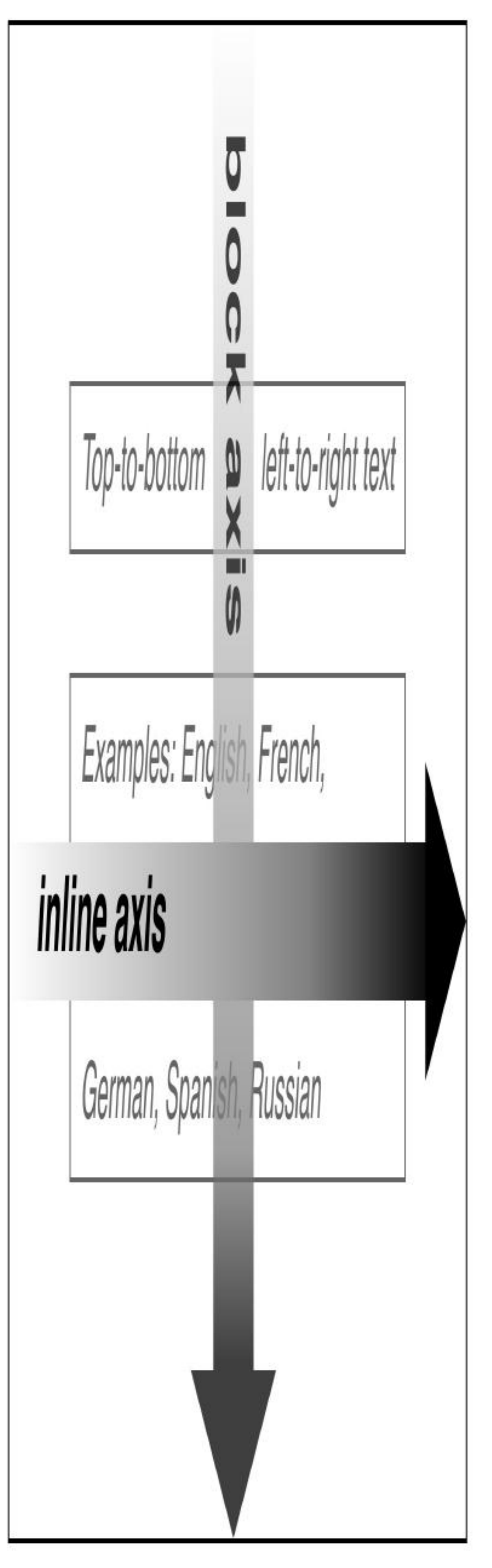
Inline-axis Properties

The seven properties of inline formatting are `margin-inline-start`, `border-inline-start`, `padding-inline-start`, `inline-size`, `padding-inline-end`, `padding-inline-end`, and `padding-inline-end`, and are diagrammed in Figure 6-30.

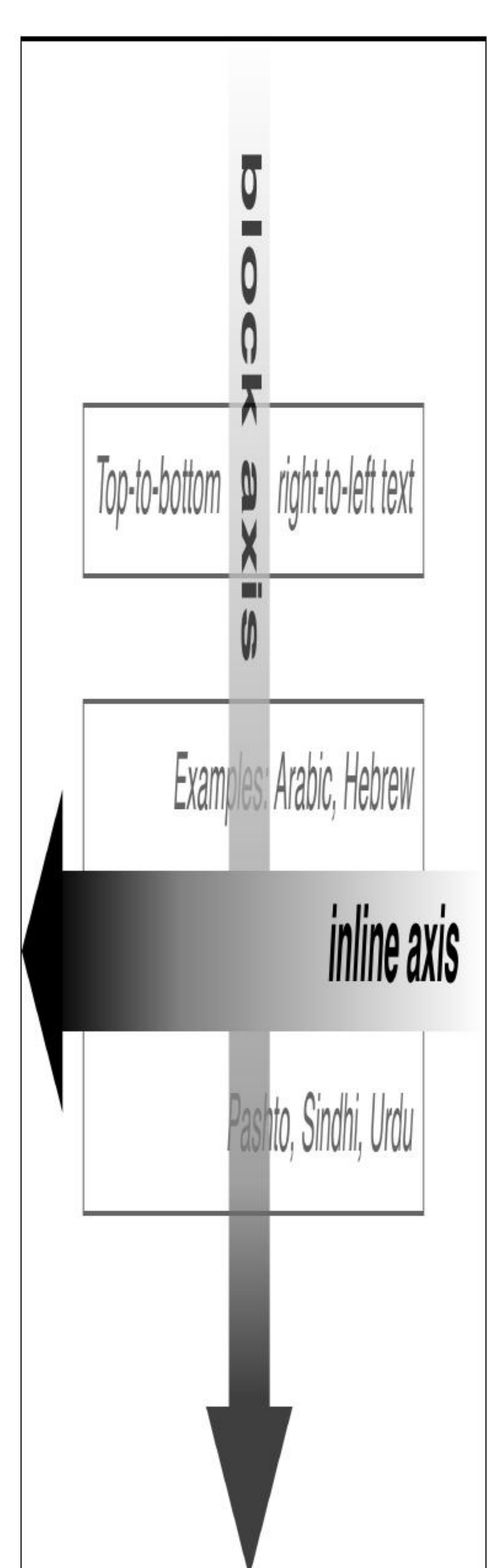
The values of these seven properties must add up to the inline size of the element's containing block, which is usually the value of `inline-size` for a block element's parent (since block-level elements

nearly always have block-level elements for parents).

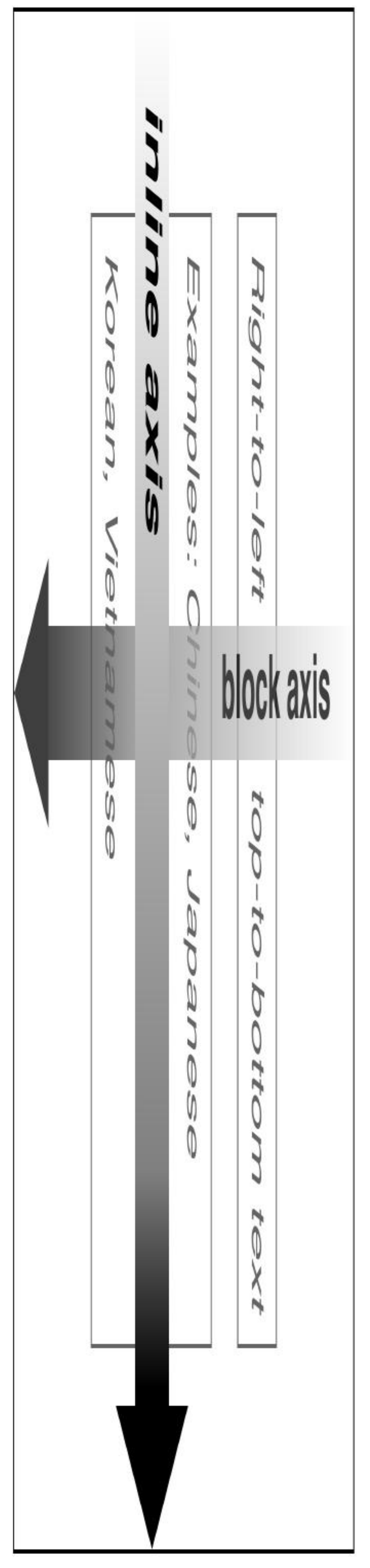
Of these seven properties, only three may be set to `auto`: the inline size of the element's content, and the inline-start and -end margins. The remaining properties must be set either to specific values or default to a width of zero. Figure 6-30 shows which parts of the box can take a value of `auto` and which cannot. (That said, CSS is forgiving: If any part that can't accept `auto` is erroneously set to `auto`, it will default to `0`.)



block axis
Top-to-bottom left-to-right text
Examples: English, French,
German, Spanish, Russian
inline axis



block axis
Top-to-bottom right-to-left text
Examples: Arabic, Hebrew
Pashto, Sindhi, Urdu
inline axis



inline axis
Right-to-left top-to-bottom text
Examples: Chinese, Japanese
Korean, Vietnamese
block axis

Figure 6-30. The seven properties of inline-axis formatting, and which of them can be set to `auto`

`inline-size` must either be set to `auto` or a nonnegative value of some type. When you do use `auto` in inline-axis formatting, different effects can occur.

Using auto

There can be situations where it makes a lot of sense to explicitly set one or more of the inline margins and size to be `auto`. By default, the two inline margins are set to `0` and the inline size is set to `auto`. Let's explore how moving the `auto` around can have different effects, and why.

Only One auto

If you set one of `inline-size`, `margin-inline-start`, or `margin-inline-end` to a value of `auto`, and give the other two properties specific values, then the property that is set to `auto` is set to the length required to make the element box's overall inline size equal to the parent element's content inline size.

In other words, let's say the sum of the seven inline-axis properties must equal 500 pixels, no padding or borders are set, the inline-end margin and inline size are set to `100px`, and the inline-start margin is set to `auto`. The inline-start margin will thus be 300 pixels wide:

```
div {inline-size: 500px;}
p {margin-inline-start: auto; margin-inline-end: 100px;
    inline-size: 100px;} /* inline-start margin evaluates to 300px */
```

In a sense, `auto` can be used to make up the difference between everything else and the required total. However, what if all three of these properties (both inline margins and the inline size) are set to `100px` and *none* of them are set to `auto`?

In the case where all three properties are set to something other than `auto`—or, in CSS parlance, when these formatting properties have been *overconstrained*—then the margin at the inline-end is *always* forced to be `auto`. This means that if both inline margins and the inline size are set to `100px`, then the user agent will reset the inline-end margin to `auto`. The inline-end margin's width will then be set according to the rule that one `auto` value "fills in" the distance needed to make the element's overall inline size equal that of its containing block's content inline size. Figure 6-31 shows the result of the following markup in left-to-right languages like English:

```
div {inline-size: 500px;}
p {margin-inline-start: 100px; margin-inline-end: 100px;
    inline-size: 100px;} /* inline-end margin forced to be 300px */
```

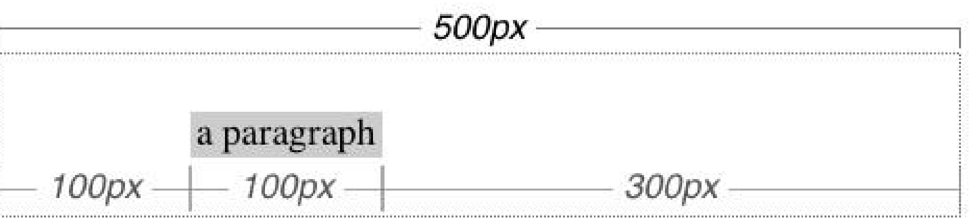


Figure 6-31. Overriding the inline-end margin's value

If both side margins are set explicitly, and `inline-size` is set to `auto`, then `inline-size` will be whatever value is needed to reach the required total (which is the content inline size of the parent element). The results of the following markup are shown in Figure 6-32:

```
p {margin-inline-start: 100px; margin-inline-end: 100px;
    inline-size: auto;}
```

The case shown in Figure 6-32 is the most common case, since it is equivalent to setting the margins and not declaring anything for the `inline-size`. The result of the following markup is exactly the same as that shown in Figure 6-32:

```
p {margin-inline-start: 100px; margin-inline-end: 100px;} /* same as before */
```

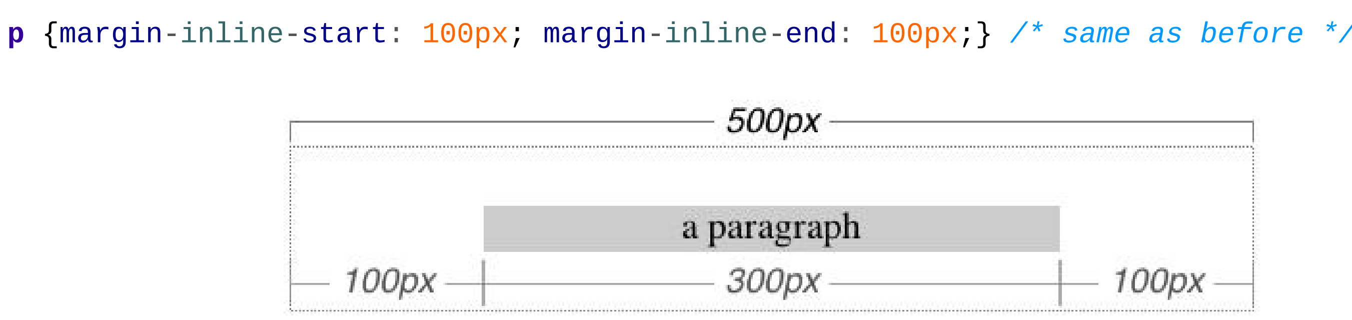


Figure 6-32. Automatic inline sizing

You might be wondering what happens if `box-sizing` is set to `padding-box`. In that case, all the same principles described here apply, which is why this section only discussed `inline-size` and the inline-side margins without introducing any padding or borders.

In other words, the handling of `inline-size: auto` in this section and the following sections is the same regardless of the value of `box-sizing`. The details of what gets placed where inside the `box-sizing`-defined box may vary, but the treatment of `auto` values does not, because `box-sizing` determines what `inline-size` refers to, not how it behaves in relation to the margins.

More Than One auto

Now let's see what happens when two of the three properties (`inline-size`, `margin-inline-start`, and `margin-inline-end`) are set to `auto`. If both margins are set to `auto` but the `inline-size` is set to a specific length, as shown in the following code, then they are set to equal lengths, thus centering the element within its parent along the inline axis. This is illustrated in Figure 6-33.

```
div {inline-size: 500px;}
p {inline-size: 300px; margin-inline-start: auto; margin-inline-end: auto;}
  /* each margin is 100 pixels, because (500-300)/2 = 100 */
```

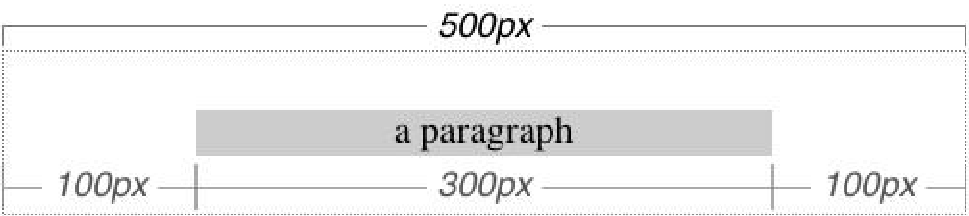


Figure 6-33. Setting an explicit inline size

Another way of sizing elements along the inline axis is to set one of the inline margins and the `inline-size` to `auto`. In this case, the margin set to be `auto` is reduced to zero:

```
div {inline-size: 500px;}
p {margin-inline-start: auto; margin-inline-end: 100px; inline-size: auto;}
  /* inline-start margin evaluates to 0; inline-size becomes 400px  */
```

The `inline-size` is then set to the value necessary to make the element fill its containing block; in the preceding example, it would be 400 pixels, as shown in Figure 6-34.

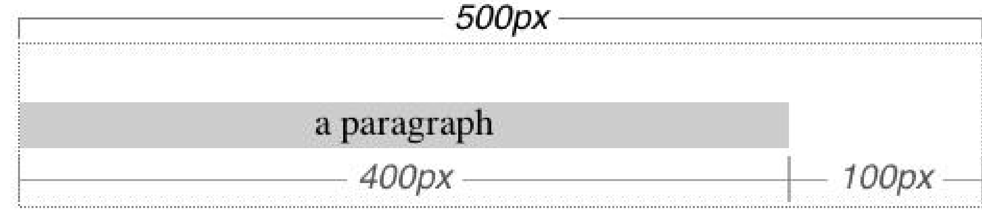


Figure 6-34. What happens when both the inline-size and the inline-start margin are auto

Too Many autos

Finally, what happens when all three properties are set to `auto`? The answer: both margins are set to zero, and the `inline-size` is made as wide as possible. This result is the same as the default situation, when no values are explicitly declared for margins or the inline size. In such a case, the margins default to zero and the `inline-size` defaults to `auto`.

Note that since inline margins do not collapse (unlike block margis, as discussed earlier), the padding, borders, and margins of a parent element can affect the inline layout its children. The effect is indirect in that the margins (and so on) of an element can induce an offset for child elements. The results of the following markup are shown in Figure 6-35:

```
div {padding: 50px; background: silver;}
p {margin: 30px; padding: 0; background: white;}
```

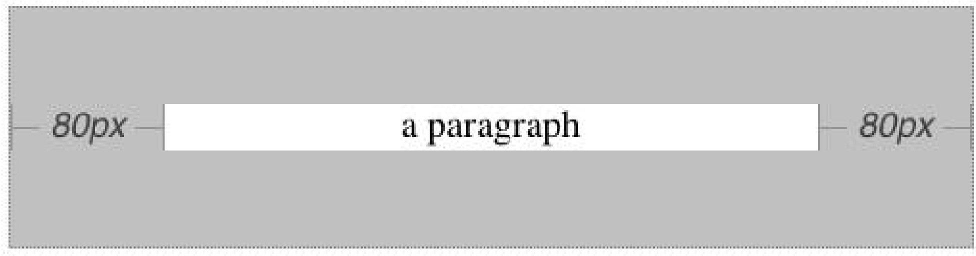


Figure 6-35. Offset is implicit in the parent’s margins and padding

Negative Margins

As seen in the section on block-axis margins, it's possible to set negative values for inline-axis margins. Setting negative inline margins can result in some interesting effects.

Remember that the total of the seven inline-axis properties always equals the inline size of the content area of the parent element. As long as all inline properties are zero or greater, an element's inline size can never be greater than its parent's content area inline size. However, consider the following markup, depicted in Figure 6-36:

```
div {inline-size: 500px; border: 3px solid black;}
p.wide {margin-inline-start: 10px; margin-inline-end: -50px;
    inline-size: auto;}
```

a paragraph

Figure 6-36. Wider children through negative margins

Yes indeed, the child element is wider than its parent along the inline axis! This is mathematically correct. If we solve for inline size:

$$10px + 0 + 0 + 540px + 0 + 0 - 50px = 500px$$

The `540px` is the evaluation of `inline-size: auto`, which is the number needed to balance out the rest of the values in the equation. Even though it leads to a child element sticking out of its parent, it all works because the values of the seven properties add up to the required total.

Now, let's add some borders to the mix:

```
div {inline-size: 500px; border: 3px solid black;}
p.wide {margin-inline-start: 10px; margin-inline-end: -50px;
    inline-size: auto; border: 3px solid gray;}
```

The resulting change will be a reduction in the evaluated width of `inline-size`:

$$10px + 3px + 0 + 534px + 0 + 3px - 50px = 500px$$

Or, if we rearrange the equation to solve for the content size instead of setting it up to solve for the width of the parent:

$$500px - 10px - 3px - 3px + 50px = 534px$$

If we were to introduce padding, then the value of `inline-size` would drop even more (assuming `box-sizing: content-box`).

Conversely, it's possible to have `auto` inline-end margins evaluate to negative amounts. If the values of other properties force the inline-end margin to be negative in order to satisfy the requirement that

elements be no wider than their containing block, then that’s what will happen. Consider:

```
div {inline-size: 500px; border: 3px solid black;}
p.wide {margin-inline-start: 10px; margin-inline-end: auto;
    inline-size: 600px; border: 3px solid gray;}
```

The equation works out like this:

$$500px - 10px - 600px - 3px - 3px = -116px$$

In this case, the inline-end margin evaluates to `-116px`. No matter what explicit value it’s given in the CSS, it will still be forced to `-116px` because of the rule stating that when an element’s dimensions are overconstrained, the inline-end margin is reset to whatever is needed to make the numbers work out correctly.

Let’s consider another example, illustrated in Figure 6-37, where the inline-start margin is set to be negative:

```
div {inline-size: 500px; border: 3px solid black;}
p.wide {margin-inline-start: -50px; margin-inline-end: 10px;
    inline-size: auto; border: 3px solid gray;}
```

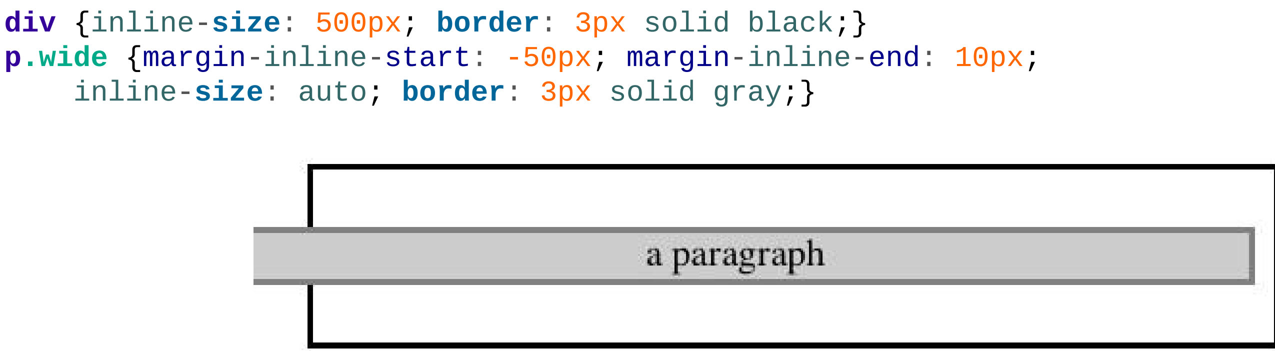


Figure 6-37. Setting a negative inline-start margin

With a negative inline-start margin, not only does the paragraph spill beyond the borders of the `<div>`, but it also spills beyond the edge of the browser window itself!

Remember: padding, borders, and content widths (and heights) can never be negative. Only margins can be less than zero.

Percentages

When it comes to percentage values for the inline size, padding, and margins, the same basic rules we discussed in previous sections apply. It doesn’t really matter whether the values are declared with lengths or percentages.

Percentages can be very useful. Suppose we want an element’s content to be two-thirds the inline size of its containing block, the padding sides to be 5% each, the inline-start margin to be 5%, and the inline-end margin to take up the slack. That would be written something like:

```
<p style="inline-size: 67%;
    padding-inline-end: 5%; padding-inline-start: 5%;
    margin-inline-end: auto; margin-inline-start: 5%;">
    playing percentages</p>
```

The inline-end margin would evaluate to 18% (100% - 67% - 5% - 5% - 5%) of the width of the containing block.

Mixing percentages and length units can be tricky, however. Consider the following example:

```
<p style="inline-size: 67%; padding-inline-end: 2em; padding-inline-start: 2em;
     margin-inline-end: auto; margin-inline-start: 5em;">mixed lengths</p>
```

In this case, the element's box can be defined like this:

5em + 0 + 2 em + 67% + 2 em + 0 + auto = containing block width

In order for the inline-end margin's inline size to evaluate to zero, the element's containing block must be 27.272727 em wide (with the content area of the element being 18.272727 em wide) along the inline axis. Any wider than that and the inline-end margin will evaluate to a positive value. Any narrower and the inline-end margin will be a negative value.

The situation gets even more complicated if we start mixing length-value unity types, like this:

```
<p style="inline-size: 67%;
     padding-inline-end: 15px; padding-inline-start: 10px;
     margin-inline-end: auto; margin-inline-start: 5em;">more mixed lengths</p>
```

And, just to make things more complex, borders cannot accept percentage values, only length values. The bottom line is that it isn't really possible to create a fully flexible element based solely on percentages unless you're willing to avoid using borders or use approaches such as flexible box layout. That said, if you do need to mix percentages and length units, using the `calc()` and `minmax()` value functions can be a life-changer, or at least a layout-changer.

Replaced Elements

So far, we've been dealing with the inline-axis formatting of nonreplaced block boxes in the normal flow of text. Replaced elements are a bit simpler to manage. All of the rules given for nonreplaced blocks hold true, with one exception: if `inline-size` is `auto`, then the `inline-size` of the element is the content's intrinsic width. ("Intrinsic" means the original size; the size the element is by default when no external factors are applied to it.) The image in the following example will be 20 pixels wide because that's the width of the original image:

```
<img src="smile.svg" style="display: block; inline-size: auto; margin: 0;">
```

If the actual image were 100 pixels wide instead, then the element (and thus the image) would be laid out as 100 pixels wide.

It's possible to override this rule by assigning a specific value to `inline-size`. Suppose we modify the previous example to show the same image three times, each with a different width value:

```
<img src="smile.svg" style="display: block; inline-size: 25px; margin: 0;">
<img src="smile.svg" style="display: block; inline-size: 50px; margin: 0;">
```

```
<img src="smile.svg" style="display: block; inline-size: 100px; margin: 0;">
```

This is illustrated in Figure 6-38.

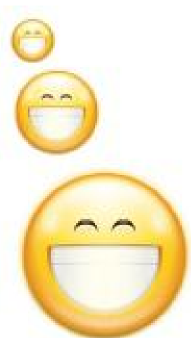


Figure 6-38. Changing replaced element inline sizes

Note that the block size of the elements also increases. When a replaced element's `inline-size` is changed from its intrinsic width, the value of `block-size` is scaled to match, maintaining the object's initial aspect ratio, unless `block-size` has been set to an explicit value of its own. The reverse is also true: if `block-size` is set, but `inline-size` is left as `auto`, then the inline size is scaled proportionately to the change in block size.

List Items

List items have a few special rules of their own. They are typically preceded by a marker, such as a round bullet mark or a number.

The marker attached to a list item element can be either outside the content of the list item or treated as an inline marker at the beginning of the content, depending on the value of the property `list-style-position`, as illustrated in Figure 6-39.

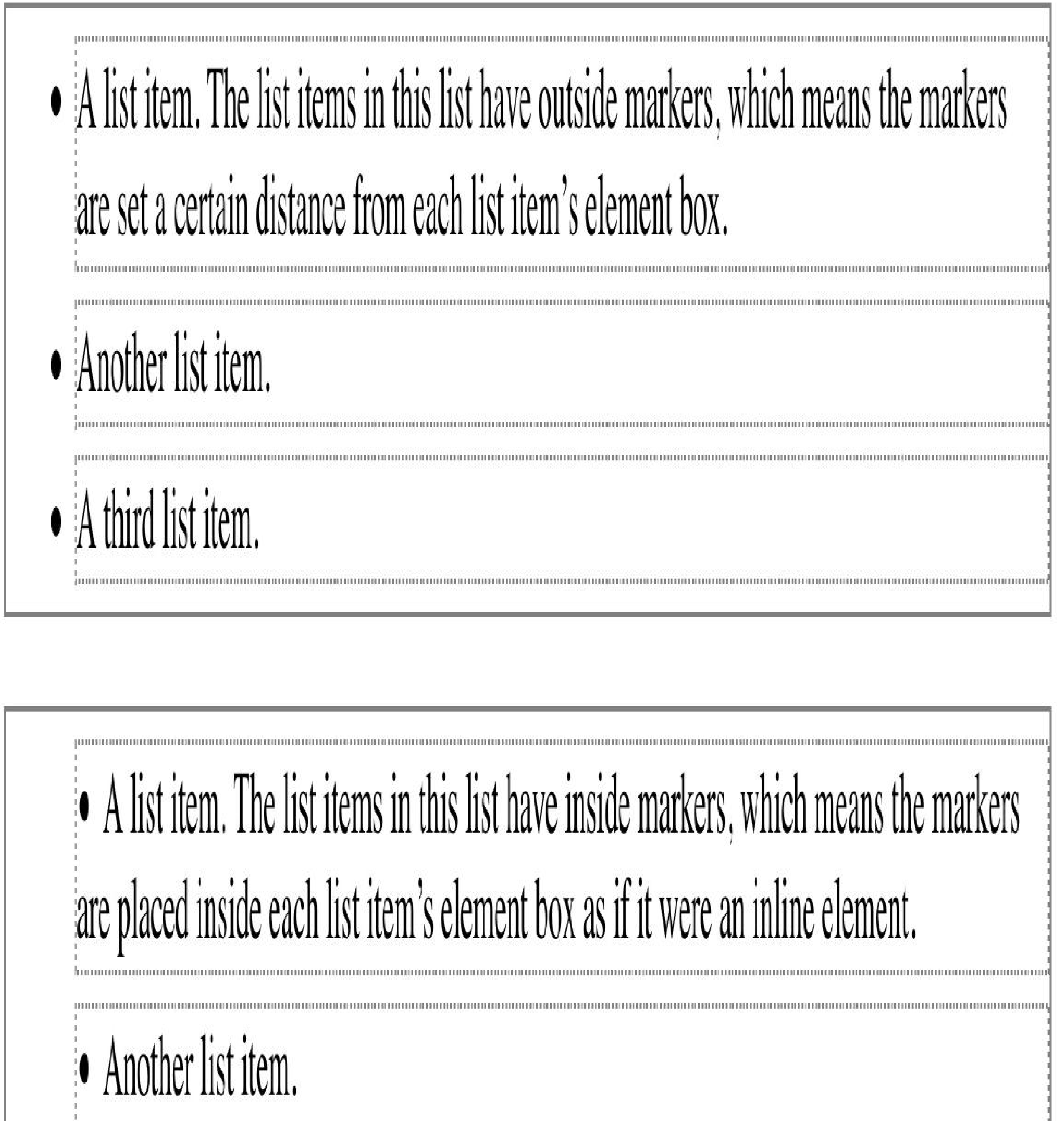


Figure 6-39. Markers outside and inside the list

If the marker stays outside the content, then it is placed some distance from the inline-start content edge of the content. No matter how the list's styles are altered, the marker stays the same distance from the content edge.

Remember that list-item boxes define containing blocks for their descendant boxes, just like regular block boxes.

> **NOTE**
>
> List markers are discussed in more detail, including how to create and style them using the `::marker` pseudo-element, in XREF HERE.

Box Sizing With Aspect Ratios

There may be times when you want to size an element by is *aspect ratio*, which means its block and inline sizes exists in a specific ratio. Old TVs used to have a 4-to-3 width-to-height ratio, for example; HD video resolutions have a 16:9 aspect ratio. You might want to force elements to be square while still letting their sizes flex. In these cases, the `aspect-ratio` property can help.

ASPECT-RATIO

Values	`auto` ‖ <*ratio*>
Initial value	`auto`
Applies to	All elements except inline boxes and internal table and ruby boxes
Computed value	If <*ratio*>, a pair of numbers; otherwise `auto`
Inherited	No
Animatable	Yes

Let's say we know we'll have a bunch of elements, and we don't know how wide or tall each will be, but we want them all to be squares. First, pick an axis you want to size on. We'll use `height` here. Make sure the other axis is auto-sized, and set an aspect ratio, like this:

```
.gallery div {width: auto; aspect-ratio: 1/1;}
```

Figure 6-40 shows the same set of HTML, both with and without the previous CSS applied.

Figure 6-40. A gallery with and without aspect ratios defined

The ratio is maintained over the distances defined by `box-sizing` (see “Altering box sizing”), so given the following CSS, the result will be an element whose outer border distances are in an exact 2:1 ratio.

```
.cards div {height: auto; box-sizing: border-box; aspect-ratio: 2/1;}
```

The default value, `auto`, means that boxes that have an intrinsic aspect ratio — boxes generated by images, for example — will use that aspect ratio. For elements that don’t have an intrinsic aspect ratio, such as most HTML elements like `<div>`, `<p>`, and so on, the axis sizes of the box will be determined by the content.

Inline Formatting

Inline formatting isn't as simple as formatting block-level elements, which just generate block boxes and usually don't allow anything to coexist with them. By contrast, look *inside* a block-level element, such as a paragraph. You may well ask, how was the size and wrapping of each line determined? What controls their arrangement? How can I affect it?

Line Layout

In order to understand how lines are generated, first consider the case of an element containing one very long line of text, as shown in Figure 6-41. Note that we've put a border around the line by wrapping the entire line in a `span` element and then assigning it a border style:

```
span {border: 1px dashed black;}
```

This is text held within a span element which is inside a containing element (a p

Figure 6-41. A single-line inline element

Figure 6-41 shows the simplest case of an inline element contained by a block-level element. It's no different in its way than a paragraph with two words in it.

In order to get from this simplified state to something more familiar, all we have to do is determine how wide (along the inline axis) the element should be, and then break up the line so that the resulting pieces will fit into the content inline size of the element. Therefore, we arrive at the state shown in Figure 6-42.

This is text held within a span element which is inside a containing
element (a paragraph, in this case). The border shows the boundaries of the
span element.

Figure 6-42. A multiple-line inline element

Nothing has really changed. All we did was take the single line and break it into pieces, and then stack those pieces one after the other along the direction of the block flow.

In Figure 6-42, the borders for each line of text also happen to coincide with the top and bottom of each line. This is true only because no padding has been set for the inline text. Notice that the borders actually overlap each other slightly; for example, the bottom border of the first line is just below the top border of the second line. This is because the border is actually drawn on the next pixel to the *outside* of each line. Since the lines are touching each other, their borders overlap as shown in Figure 6-42.

> **NOTE**
>
> For simplicity's sake, we're going to use terms like "top" and "bottom" when talking about the edges of line boxes. In this context, the top of a line box is the one closest to the block-start, and the bottom of a line box is the one closest to the block-end. Similarly, "tall" and "short" will refer to the size of line boxes along the block axis.

If we alter the span styles to have a background color, the actual placement of the lines becomes more clear. Consider Figure 6-43, which shows four paragraphs in each of two different writing modes, and the effects of different values of `text-align` (see Chapter 11), by each paragraph having the backgrounds of its lines filled in.

This paragraph assumes the style `text-align: left;`, which causes the line boxes within the element to line up along the left inner content edge of the paragraph.

This paragraph assumes the style `text-align: right;`, which causes the line boxes within the element to line up along the right inner content edge of the paragraph.

This paragraph assumes the style `text-align: center;`, which causes the line boxes within the element to line up their centers with the center of the content area of the paragraph.

This paragraph assumes the style `text-align: justify;`, which causes the line boxes within the element to align their inline-start and inline-end edges to the inline-start and -end inner content edges of the paragraph. The exception is the last line box, whose inline-end edge does not align with the inline-end content edge of the paragraph.

This paragraph assumes the style `text-align: left;`, which causes the line boxes within the element to line up along the left inner content edge of the paragraph.

This paragraph assumes the style `text-align: right;`, which causes the line boxes within the element to line up along the right inner content edge of the paragraph.

This paragraph assumes the style `text-align: center;`, which causes the line boxes within the element to line up their centers with the center of the content area of the paragraph.

This paragraph assumes the style `text-align: justify;`, which causes the line boxes within the element to align their inline-start and inline-end edges to the inline-start and -end inner content edges of the paragraph. The exception is the last line box, whose inline-end edge does not align with the inline-end content edge of the paragraph.

Figure 6-43. Showing lines in different alignments and writing modes

As Figure 6-43 shows, not every line reaches to the edge of its parent paragraph's content area, which has been denoted with a dashed gray border. For the left-aligned paragraph, the lines are all pushed flush against the left content edge of the paragraph, and the end of each line happens wherever the line is broken. The reverse is true for the right-aligned paragraph. For the centered paragraph, the centers of the lines are aligned with the center of the paragraph.

In the last case, where the value of `text-align` is `justify`, each line (except the last) is forced to be as wide as the paragraph's content area so that the line's edges touch the content edges of the paragraph. The difference between the natural length of the line and the width of the paragraph's content area is made up by altering the spacing between letters and words in each line. Therefore, the value of `word-spacing` can be overridden when the text is justified. (The value of `letter-spacing` cannot be overridden if it is a length value.)

That pretty well covers how lines are generated in the simplest cases. As you're about to see, however, the inline formatting model is far from simple.

Basic Terms and Concepts

Before we go any further, let's review some terms of inline layout, which will be crucial in navigating the following sections:

Anonymous text

This is any string of characters that is not contained within an inline element. Thus, in the markup `<p> I'm <em>so</em> happy!</p>`, the sequences " I'm " and " happy!" are anonymous text. Note that the spaces are part of the text, since a space is a character like any other.

Em box

This is defined in the given font, otherwise known as the character box. Actual glyphs can be taller or shorter than their em boxes. In CSS, the value of `font-size` determines the height of each em box.

Content area

In nonreplaced elements, the content area can be one of two things, and the CSS specification allows user agents to choose which one. The content area can be the box described by the em boxes of every character in the element, strung together; or it can be the box described by the character glyphs in the element. In this book, we use the em box definition for simplicity's sake, and that's what is used by most browsers. In replaced elements, the content area is the intrinsic height of the element plus any margins, borders, or padding.

Leading

Leading (pronounced "led-ing") is the difference between the values of `font-size` and `line-height`. This difference is divided in half, with one half applied to the top and one half to the bottom of the content area. These additions to the content area are called, perhaps unsurprisingly, *half-*

leading. Leading is applied only to nonreplaced elements.

Inline box

This is the box described by the addition of the leading to the content area. For nonreplaced elements, the height of the inline box of an element will be exactly equal to the value of the `line-height` property. For replaced elements, the height of the inline box of an element will be exactly equal to the content area, since leading is not applied to replaced elements.

Line box

This is the shortest box that bounds the highest and lowest points of the inline boxes that are found in the line. In other words, the top edge of the line box is placed along the top of the highest inline box top, and the bottom of the line box is placed along the bottom of the lowest inline box bottom. Remember that "top" and "bottom" are considered with respect to the block flow direction.

CSS also contains a set of behaviors and useful concepts that fall outside of the preceding list of terms and definitions:

- The content area of an inline box is analogous to the content box of a block box.
- The background of an inline element is applied to the content area plus any padding.
- Any border on an inline element surrounds the content area plus any padding.
- Padding, borders, and margins on nonreplaced inline elements have no vertical effect on the inline elements or the boxes they generate; that is, they do *not* affect the height of an element's inline box (and thus the line box that contains the element).
- Margins and borders on replaced elements *do* affect the height of the inline box for that element and, by implication, the height of the line box for the line that contains the element.

One more thing to note: inline boxes are vertically aligned within the line according to their values for the property `vertical-align` (see Chapter 11).

Before moving on, let's look at a step-by-step process for constructing a line box, which you can use to see how the various pieces of a line fit together to determine its height.

Determine the height of the inline box for each element in the line by following these steps:

1. Find the values of `font-size` and `line-height` for each inline nonreplaced element and text that is not part of a descendant inline element and combine them. This is done by subtracting the `font-size` from the `line-height`, which yields the leading for the box. The leading is split in half and applied to the top and bottom of each em box.
2. Find the value of `height`, along with the values for the margins, padding, and borders along the block-start and block-end edges of each replaced element, and add them together.
3. Figure out, for each content area, how much of it is above the baseline for the overall line and how much of it is below the baseline. This is not an easy task: you must know the position of the baseline

for each element and piece of anonymous text and the baseline of the line itself, and then line them all up. In addition, the block-end edge of a replaced element sits on the baseline for the overall line.

4. Determine the vertical offset of any elements that have been given a value for `vertical-align`. This will tell you how far up or down that element's inline box will be moved along the block axis, and that will change how much of the element is above or below the baseline.

5. Now that you know where all of the inline boxes have come to rest, calculate the final line box height. To do so, just add the distance between the baseline and the highest inline box top to the distance between the baseline and the lowest inline box bottom.

Let's consider the whole process in detail, which is the key to intelligently styling inline content.

Line Heights

First, know that all elements have a `line-height`, whether it's explicitly declared or not. This value greatly influences the way inline elements are displayed, so let's give it due attention.

A line's height (or the height of a line box) is determined by the height of its constituent elements and other content, such as text. It's important to understand that `line-height` actually affects inline elements and other inline content, *not* block-level elements—at least, not directly. We can set a `line-height` value for a block-level element, but the value will have a visual impact only as it's applied to inline content within that block-level element. Consider the following empty paragraph, for example:

```
<p style="line-height: 0.25em;"></p>
```

Without content, the paragraph won't have anything to display, so we won't see anything. The fact that this paragraph has a `line-height` of any value—be it `0.25em` or `25in`—makes no difference without some content to create a line box.

We can set a `line-height` value for a block-level element and have that apply to all of the content within the block, whether it's contained in an inline element or anonymous text. In a certain sense, then, each line of text contained within a block-level element is its own inline element, whether or not it's surrounded by tags. If you like, picture a fictional tag sequence like this:

```
<p>
<line>This is a paragraph with a number of</line>
<line>lines of text that make up the</line>
<line>contents.</line>
</p>
```

Even though the `line` tags don't actually exist, the paragraph behaves as if they did, and each line of text "inherits" styles from the paragraph. You only bother to create `line-height` rules for block-level elements so you don't have to explicitly declare a `line-height` for all of their inline elements, fictional or otherwise.

The fictional `line` element actually clarifies the behavior that results from setting `line-height` on a block-level element. According to the CSS specification, declaring `line-height` on a block-level

element sets a *minimum* line box height for the content of that block-level element. Declaring `p.spacious {line-height: 24pt;}` means that the *minimum* heights for each line box is 24 points. Technically, content can inherit this line height only if an inline element does so. Most text isn't contained by an inline element. If you pretend that each line is contained by the fictional `line` element, the model works out very nicely.

Inline Nonreplaced Elements

Building on our formatting knowledge, let's move on to the construction of lines that contain only nonreplaced elements (or anonymous text). Then you'll be in a good position to understand the differences between nonreplaced and replaced elements in inline layout.

> **NOTE**
>
> In this section, we'll use "top" and "bottom" to label where half-leading is placed and how line boxes are placed together. Always remember that these terms are in relation to the direction of block flow: the "top" edge of an inline box is the one closest to the block-start edge, and the "bottom" edge of an inline box is closest to its block-end edge. Similarly, "height" means the the distance along the inline box's block axis, and "width" the distance along its inline axis.

Building the Boxes

First, for an inline nonreplaced element or piece of anonymous text, the value of `font-size` determines the height of the content area. If an inline element has a `font-size` of `15px`, then the content area's height is 15 pixels because all of the em boxes in the element are 15 pixels tall, as illustrated in Figure 6-44.

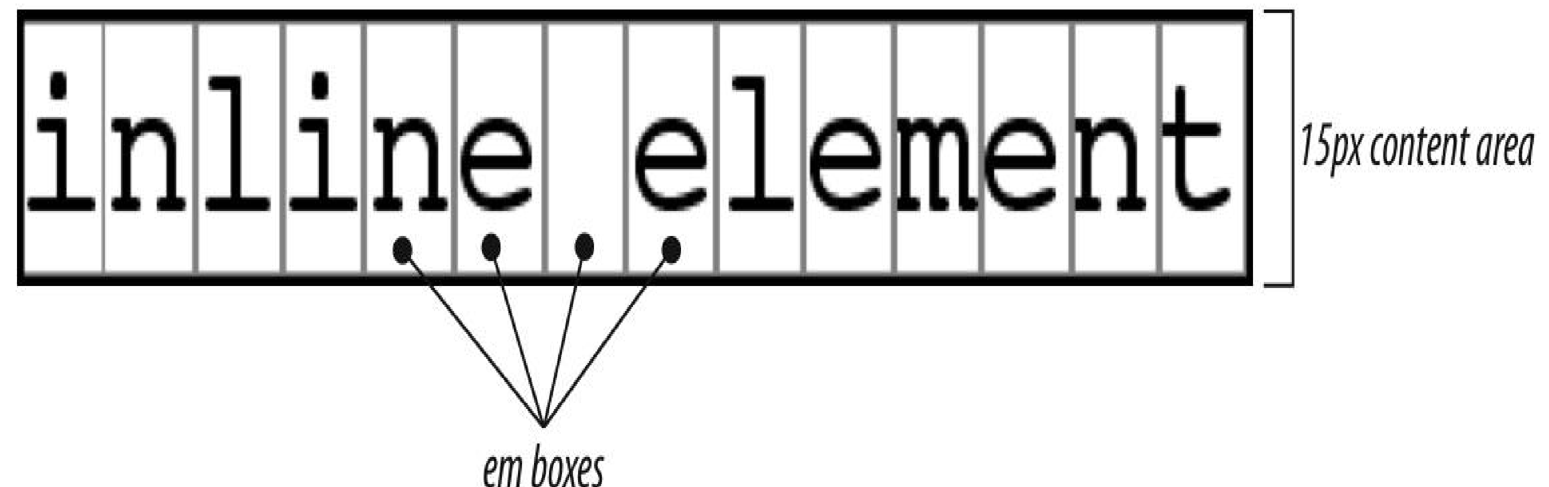


Figure 6-44. Em boxes determine content area height

The next thing to consider is the value of `line-height` for the element, and the difference between it and the value of `font-size`. If an inline nonreplaced element has a `font-size` of `15px` and a `line-height` of `21px`, then the difference is six pixels. The user agent splits the six pixels in half and applies half (3 pixels) to the top and half (3 pixels) to the bottom of the content area, which yields the inline box. This process is illustrated in Figure 6-45.

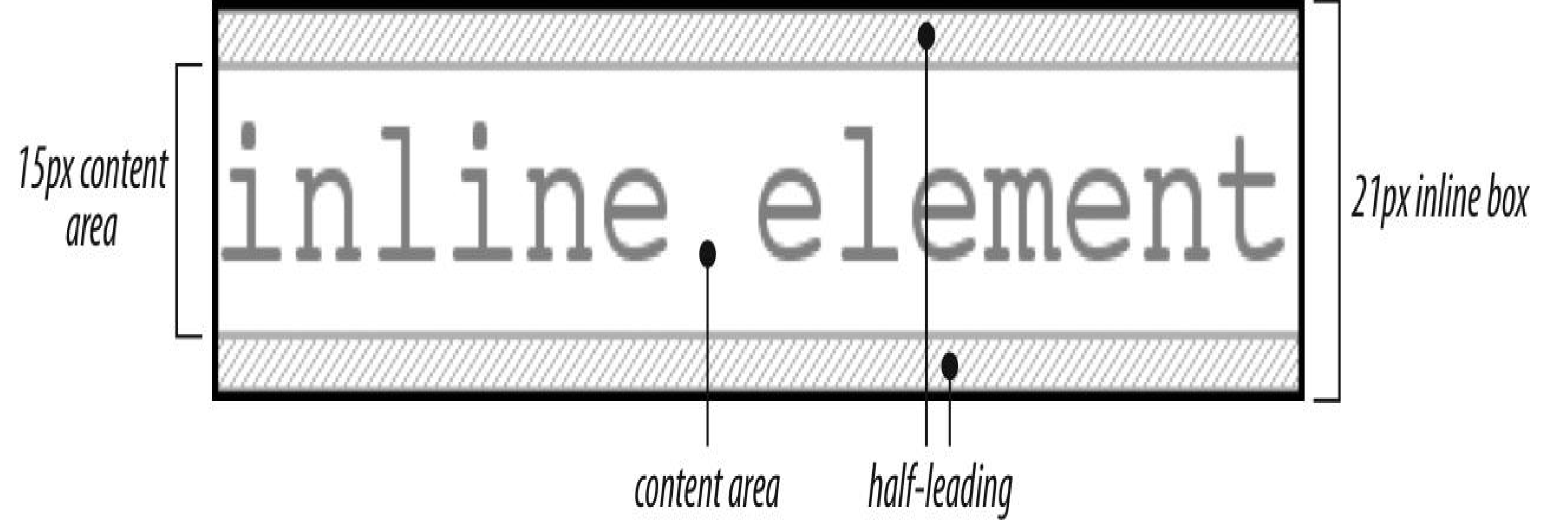


Figure 6-45. Content area plus leading equals inline box

Now, let’s break stuff so we can better understand how line height works. Assume the following is true:

```
<p style="font-size: 12px; line-height: 12px;">
This is text, <em>some of which is emphasized</em>, plus other text<br>
which is <strong style="font-size: 24px;">strongly emphasized</strong>
and which is<br>
larger than the surrounding text.
</p>
```

In this example, most of the text has a `font-size` of `12px`, while the text in one inline nonreplaced element has a size of `24px`. However, *all* of the text has a `line-height` of `12px` since `line-height` is an inherited property. Therefore, the `strong` element’s `line-height` is also `12px`.

Thus, for each piece of text where both the `font-size` and `line-height` are `12px`, the content height does not change (since the difference between `12px` and `12px` is zero), so the inline box is 12 pixels high. For the strong text, however, the difference between `line-height` and `font-size` is `-12px`. This is divided in half to determine the half-leading (`-6px`), and the half-leading is added to both the top and bottom of the content height to arrive at an inline box. Since we’re adding a negative number in both cases, the inline box ends up being 12 pixels tall. The 12-pixel inline box is centered vertically within the 24-pixel content height of the element, so the inline box is actually smaller than the content area.

So far, it sounds like we’ve done the same thing to each bit of text, and that all the inline boxes are the same size, but that’s not quite true. The inline boxes in the second line, although they’re the same size, don’t actually line up because the text is all baseline-aligned (see Figure 6-46), a concept we’ll discuss later in the chapter.

Since inline boxes determine the height of the overall line box, their placement with respect to each other is critical. The line box is defined as the distance from the top of the highest inline box in the line to the bottom of the lowest inline box, and the top of each line box butts up against the bottom of the line box for the preceding line.

In Figure 6-46, there are three boxes being laid out for a single line of text: the two anonymous text boxes to either side of the `strong` element, and the `strong` element itself. Because the enclosing paragraph

has a `line-height` of `12px`, each of the three boxes will have a 12-pixel-tall inline box. These inline boxes are centered within the content area of each box. The boxes then have their baselines lined up, so the text all shares a common baseline.

But because of where the inline boxes fall with respect to those baselines, the inline box of the `strong` element is a little bit higher than the inline boxes of the anonymous text boxes. Thus, the distance from the top of the `strong`'s inline box to the bottoms of the anonymous inline boxes is more than 12 pixels, while the visible content of the line isn't completely contained within the line box.

Figure 6-46. Inline boxes within a line

After all that, the middle line of text is placed between two other lines of text, as depicted in Figure 6-47. The bottom edge of the first line of text is placed against the top edge of the line of text we saw in Figure 6-46. Similarly, the top edge of the third line of text is placed against the bottom edge of the middle line of text. Because the middle line of text has a slightly taller line box, the result is that the lines of text look irregular, because the distances between the three baselines are not consistent.

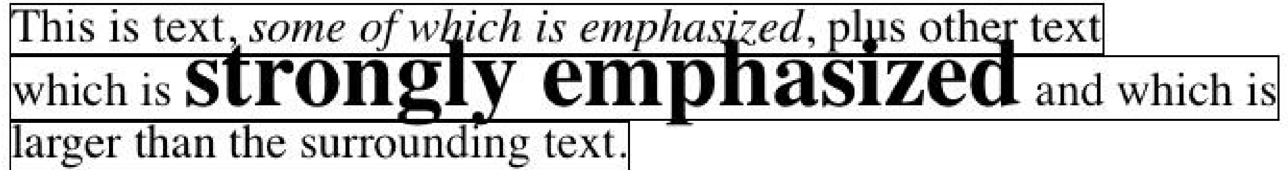


Figure 6-47. Line boxes within a paragraph

> **NOTE**
>
> In just a bit, we'll explore ways to cope with this behavior and methods for achieving consistent baseline spacing. (Spoiler: Unitless values for the win!)

Vertical Alignment

If we change the vertical alignment of the inline boxes, the same height determination principles apply. Suppose that we give the `strong` element a vertical alignment of `4px`:

```
<p style="font-size: 12px; line-height: 12px;">
This is text, <em>some of which is emphasized</em>, plus other text<br>
which is <strong style="font-size: 24px; vertical-align: 4px;">strongly
emphasized</strong>  and that is<br>
larger than the surrounding text.
</p>
```

That small change raises the `strong` element four pixels, which pushes up both its content area and its inline box. Because the `strong` element's inline box top was already the highest in the line, this change in vertical alignment also pushes the top of the line box upward by four pixels, as shown in Figure 6-48.

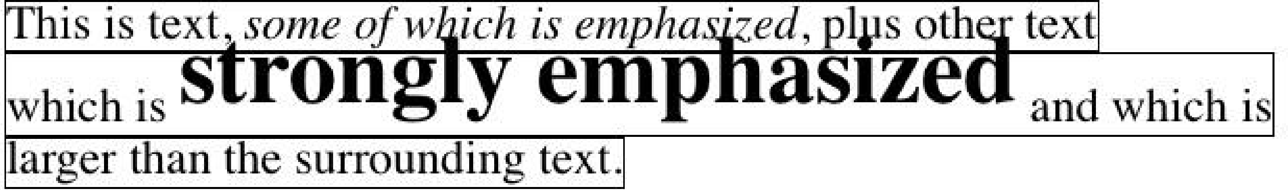


Figure 6-48. Vertical alignment affects line box height

> **NOTE**
>
> A formal definition for `vertical-align` can be found in Chapter 11.

Let's consider another situation. Here, we have another inline element in the same line as the strong text, and its alignment is other than the baseline:

```
<p style="font-size: 12px; line-height: 12px;">
This is text, <em>some of which is emphasized</em>,<br>
plus other text that is <strong style="font-size: 24px; vertical-align: 4px;">
strong</strong> and <span style="vertical-align: top;">tall</span> and is<br>
larger than the surrounding text.
</p>
```

Now we have the same result as in our earlier example, where the middle line box is taller than the other line boxes. However, notice how the "tall" text is aligned in Figure 6-49.

This is text, *some of which is emphasized*,
plus other text that is **strong** and tall and is
larger than the surrounding text.

Figure 6-49. Aligning an inline element to the line box

In this case, the top of the "tall" text's inline box is aligned with the top of the line box. Since the "tall" text has equal values for `font-size` and `line-height`, the content height and inline box are the same. However, consider this:

```
<p style="font-size: 12px; line-height: 12px;">
This is text, <em>some of which is emphasized</em>,<br>
plus other text that is <strong style="font-size: 24px; vertical-align: 4px;">
strong</strong> and <span style="vertical-align: top; line-height: 2px;">
tall</span> and is<br> larger than the surrounding text.
```

```
</p>
```

Since the `line-height` for the "tall" text is less than its `font-size`, the inline box for that element is smaller than its content area. This tiny fact changes the placement of the text itself, because the top of its inline box must be aligned with the top of the line box for its line. Thus, we get the result shown in Figure 6-50.

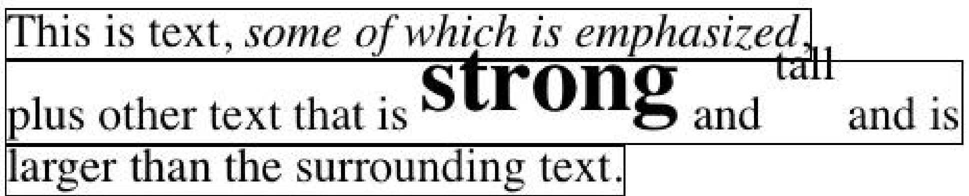


Figure 6-50. Text protruding from the line box (again)

In relation to the terms we've been using in this chapter, the effects of the assorted keyword values of `vertical-align` are:

`top`

Aligns the top (block-start edge) of the element's inline box with the top of the containing line box.

`bottom`

Aligns the bottom (block-end edge) of the element's inline box with the bottom of the containing line box.

`text-top`

Aligns the top (block-start edge) of the element's inline box with the top of the parent's content area.

`text-bottom`

Aligns the bottom (block-end edge) of the element's inline box with the bottom of the parent's content area.

`middle`

Aligns the vertical midpoint of the element's inline box with `0.5ex` above the baseline of the parent.

`super`

Moves the content area and inline box of the element upward along the block axis. The distance is not specified and may vary by user agent.

`sub`

The same as `super`, except the element is moved downward along the block axis instead of upward.

<percentage>

Shifts the element up or down the block axis by the distance defined by taking the declared percentage of the element's value for `line-height`.

Managing the line-height

In previous sections, you saw that changing the `line-height` of an inline element can cause text from one line to overlap another. In each case, though, the changes were made to individual elements. So how can you affect the `line-height` of elements in a more general way in order to keep content from overlapping?

One way to do this is to use the `em` unit in conjunction with an element whose `font-size` has changed. For example:

```
p {line-height: 1em;}
strong {font-size: 250%; line-height: 1em;}

<p>
Not only does this paragraph have "normal" text, but it also<br>
contains a line in which <strong>some big text</strong> is found.<br>
This large text helps illustrate our point.
</p>
```

By setting a `line-height` for the `strong` element, we increase the overall height of the line box, providing enough room to display the `strong` element without overlapping any other text and without changing the `line-height` of all lines in the paragraph. We use a value of `1em` so that the `line-height` for the `strong` element will be set to the same size as `strong`'s font-size. Remember, `line-height` is set in relation to the `font-size` of the element itself, not the parent element. The results are shown in Figure 6-51.

Not only does this paragraph have "normal" text, but it also
contains a line in which some big text is found.
This large text helps illustrate our point.

Figure 6-51. Assigning the line-height property to inline elements

Make sure you really understand the previous sections, because things will get trickier when we try to add borders. Let's say we want to put five-pixel borders around any hyperlink:

```
a:any-link {border: 5px solid blue;}
```

If we don't set a large enough `line-height` to accommodate the border, it will be in danger of overwriting other lines. We could increase the size of the inline box for hyperlinks using `line-height`, as we did for the `strong` element in the earlier example; in this case, we'd just need to make the value of `line-height` 10 pixels larger than the value of font-size for those links. However, that will be

difficult if we don’t actually know the size of the font in pixels.

Another solution is to increase the `line-height` of the paragraph. This will affect every line in the entire element, not just the line in which the bordered hyperlink appears:

```
p {line-height: 1.8em;}
a:link {border: 5px solid blue;}
```

Because there is extra space added above and below each line, the border around the hyperlink doesn’t impinge on any other line, as shown in Figure 6-52.

Not only does this paragraph have “normal” text, but it also
contains a line in which a hyperlink is found.
This large text helps illustrate our point.

Figure 6-52. Increasing line-height to leave room for inline borders

This approach works because all of the text is the same size. If there were other elements in the line that changed the height of the line box, our border situation might also change. Consider the following:

```
p {font-size: 14px; line-height: 24px;}
a:link {border: 5px solid blue;}
strong {font-size: 150%; line-height: 1.5em;}
```

Given these rules, the height of the inline box of a `strong` element within a paragraph will be 31.5 pixels (14 × 1.5 × 1.5), and that will also be the height of the line box. In order to keep baseline spacing consistent, we must make the `p` element’s `line-height` equal to or greater than 32px.

Baselines and line heights

The actual height of each line box depends on the way its component elements line up with one another. This alignment tends to depend very much on where the baseline falls within each element (or piece of anonymous text) because that location determines how the inline boxes are arranged vertically.

Consistent baseline spacing tends to be more of an art than a science. If you declare all of your font sizes and line heights using a single unit, such as ems, then you have a good chance of consistent baseline spacing. If you mix units, however, that feat becomes a great deal more difficult, if not impossible. As of mid-2022, there are proposals for properties that would let authors enforce consistent baseline spacing regardless of the inline content, which would greatly simplify certain aspects of online typography. None of these proposed properties have been implemented, which makes their adoption a distant hope at best.

Scaling Line Heights

The best way to set `line-height`, as it turns out, is to use a raw number as the value. This method is best because the number becomes the *scaling factor*, and that factor is an inherited, not a computed, value. Let’s say we want the line-height`s of all elements in a document to be

one and a half times their `font-size. We would declare:

```
body {line-height: 1.5;}
```

This scaling factor of 1.5 is passed down from element to element, and, at each level, the factor is used as a multiplier of the `font-size` of each element. Therefore, the following markup would be displayed as shown in Figure 6-53:

```
p {font-size: 15px; line-height: 1.5;}
small {font-size: 66%;}
strong {font-size: 200%;}

<p>This paragraph has a line-height of 1.5 times its font-size. In addition,
any elements within it <small>such as this small element</small> also have
line-heights 1.5 times their font-size...and that includes <strong>this big
element right here</strong>. By using a scaling factor, line-heights scale
to match the font-size of any element.</p>
```

In this example, the line height for the `small` element turns out to be 15 pixels, and for the `strong` element, it's 45 pixels. If we don't want our big `strong` text to generate too much extra leading, we can give it its own `line-height` value, which will override the inherited scaling factor:

```
p {font-size: 15px; line-height: 1.5;}
small {font-size: 66%;}
strong {font-size: 200%; line-height: 1em;}
```

Figure 6-53. Using a scaling factor for line-height

Adding Box Properties

As you may recall from previous discussions, while padding, margins, and borders may all be applied to inline nonreplaced elements, these properties have no impact on the height of the inline element's line box.

The border edge of inline elements is controlled by the `font-size`, not the `line-height`. In other words, if a `span` element has a `font-size` of `12px` and a `line-height` of `36px`, its content area is `12px` high, and the border will surround that content area.

Alternatively, we can assign padding to the inline element, which will push the borders away from the text itself:

```
span {padding: 4px;}
```

Note that this padding does not alter the actual shape of the content height, and so it will not affect the height of the inline box for this element. Similarly, adding borders to an inline element will not affect the way line boxes are generated and laid out, as illustrated in Figure 6-54 (both with and without the four-pixel padding).

The text in this paragraph has been wrapped with a span element, to which a border and no padding has been applied. This helps to visualize the limits of each line's box. Note that in certain cases the borders can actually pass each other; this is because the border is drawn around the outside of the element's content, and so sticks one pixel beyond the actual limit of each line's content area (which would technically fall in the space between pixels).

The text in this paragraph has been wrapped with a span element, to which a border and 4px of padding has been applied. This helps to visualize the limits of each line's box. Note that in certain cases the borders can actually pass each other; this is because the border is drawn around the outside of the element's content, and so sticks one pixel beyond the actual limit of each line's content area (which would technically fall in the space between pixels).

Figure 6-54. Padding and borders do not alter line-height

As for margins, they do not, practically speaking, apply to the block edges of an inline nonreplaced element, as they don't affect the height of the line box. The inline ends of the element are another story.

Recall the idea that an inline element is basically laid out as a single line and then broken up into pieces. So, if we apply margins to an inline element, those margins will appear at its beginning and end: these are the inline-start and inline-end margins, respectively. Padding also appears at these edges. Thus, although padding and margins (and borders) do not affect line heights, they can still affect the layout of an element's content by pushing text away from its ends. In fact, negative inline-start and -end margins can pull text closer to the inline element, or even cause overlap, as Figure 6-55 shows.

So, what happens when an inline element has a background and enough padding to cause the lines' backgrounds to overlap? Take the following situation as an example:

```
p {font-size: 15px; line-height: 1em;}
p span {background: #FAA;
    padding-block-start: 10px; padding-block-end: 10px;}
```

All of the text within the `span` element will have a content area 15 pixels tall, and we've applied 10 pixels of padding to the top and bottom of each content area. The extra pixels won't increase the height of the line box, which would be fine, except there is a background color. Thus, we get the result shown in Figure 6-55.

The text in this paragraph contains a span element that has been give inline-start and inline-end padding and negative inline-start and -end margins, plus a background, which causes some interesting effects. The extra space you see at the beginning and end of the span and the observed overlap of other content are to be expected.

The text in this paragraph contains a span element that has been given block-start and block-end padding plus a background, which causes some interesting effects. The extra space you see above and below the span and the observed overlap of other content are to be expected.

Figure 6-55. Padding and margins on inline elements

CSS explicitly states that the line boxes are drawn in document order: "This will cause the borders on

subsequent lines to paint over the borders and text of previous lines.” The same principle applies to backgrounds as well, as Figure 6-55 shows.

Changing Breaking Behavior

In the previous section, you saw that when an inline nonreplaced element is broken across multiple lines, it’s treated as if it were one long single-line element that’s sliced into smaller boxes, one slice per line break. That’s actually just the default behavior, and it can be changed via the property `box-decoration-break`.

BOX-DECORATION-BREAK

Values	`slice` | `clone`
Initial value	`slice`
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	No

The default value, `slice`, is what we saw in the previous section. The other value, `clone`, causes each fragement of the element to be drawn as if it were a standalone box. What does that mean? Compare the two examples in Figure 6-56, in which exactly the same markup and styles are treated as either sliced or cloned.

Many of the differences may be apparent, but a few are perhaps more subtle. Among the effects are the application of padding to each element’s fragment, including at the ends where the line breaks occurred. Similarly, the border is drawn around each fragment individually, instead of being broken up.

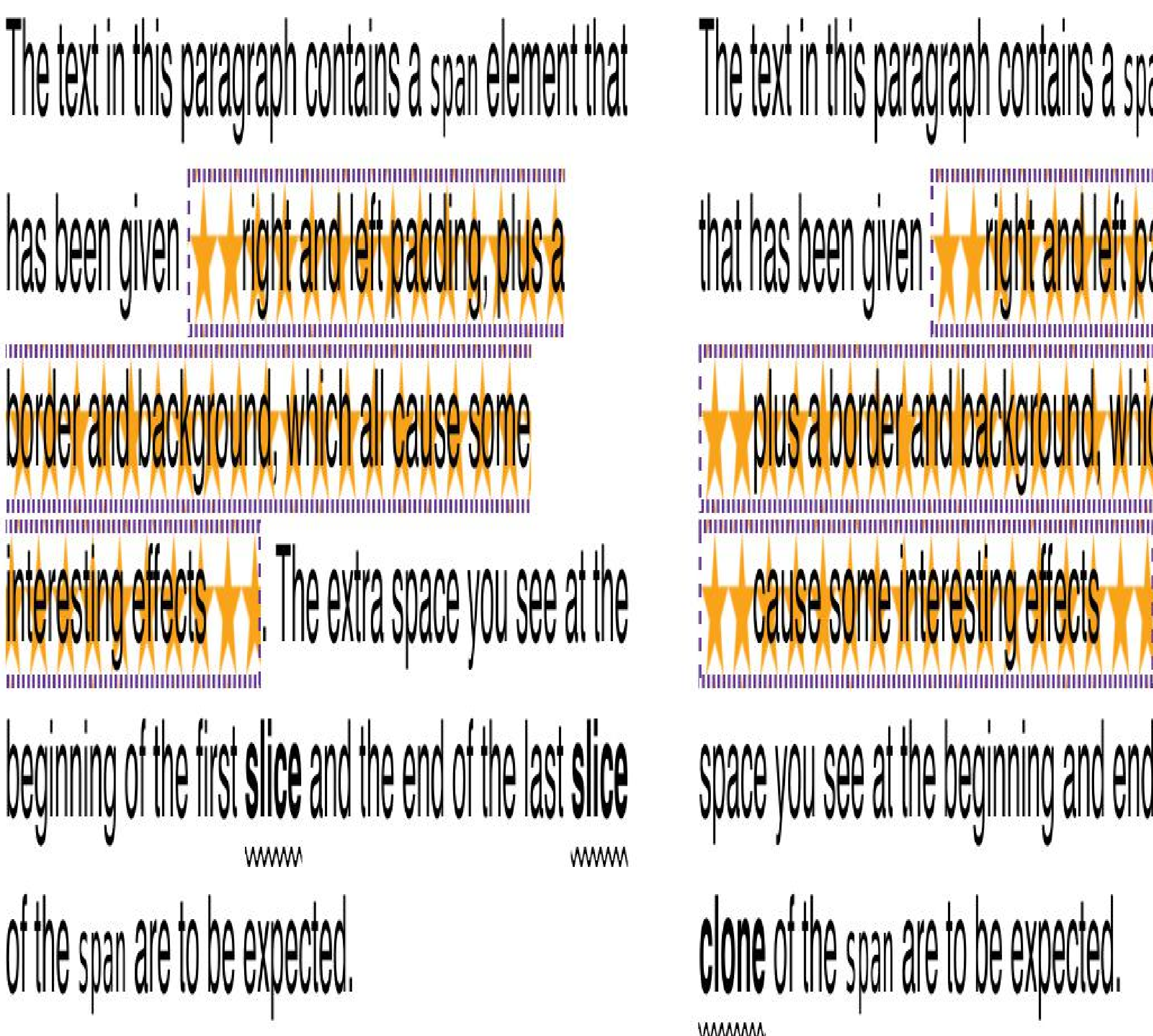


Figure 6-56. Sliced and cloned inline fragments

More subtly, notice how the `background-image` positioning changes between the two. In the sliced version, background images are sliced along with everything else, meaning that only one of the fragments contains the origin image. In the cloned version, however, each background acts as its own copy, so each has its own origin image. This means, for example, that even if we have a nonrepeated background image,

it will appear once in each fragment instead of only in one fragment.

The `box-decoration-break` property will most often be used with inline boxes, but it actually applies in any situation where there's a break in an element—for example, when a page break interrupts an element in paged media. In such a case, each fragment is a separate slice. If we set `box-decoration-break: clone`, then each box fragment will be treated as a copy when it comes to borders, padding, backgrounds, and so on. The same holds true in multicolumn layout: if an element is split by a column break, the value of `box-decoration-break` will affect how it is rendered.

Glyphs Versus Content Area

Even in cases where you try to keep inline nonreplaced element backgrounds from overlapping, it can still happen, depending on which font is in use. The problem lies in the difference between a font's em box and its character glyphs. Most fonts, as it turns out, don't have em boxes whose heights match the character glyphs.

That may sound very abstract, but it has practical consequences. The "painting area" of an inline nonreplaced element is left to the user agent. If a user agent takes the em box to be the height of the content area, then the background of an inline nonreplaced element will be equal to the height of the em box (which is the value of `font-size`). If a user agent uses the maximum ascender and descender of the font, then the background may be taller or shorter than the em box. Therefore, you could give an inline nonreplaced element a `line-height` of `1em` and still have its background overlap the content of other lines.

Inline Replaced Elements

Inline replaced elements, such as images, are assumed to have an intrinsic height and width; for example, an image will be a certain number of pixels high and wide. Therefore, a replaced element with an intrinsic height can cause a line box to become taller than normal. This does *not* change the value of `line-height` for any element in the line, *including the replaced element itself*. Instead, the line box is made just tall enough to accommodate the replaced element, plus any box properties. In other words, the entirety of the replaced element—content, margins, borders, and padding—is used to define the element's inline box. The following styles lead to one such example, as shown in Figure 6-57:

```
p {font-size: 15px; line-height: 18px;}
img {block-size: 30px; margin: 0; padding: 0; border: none;}
```

This paragraph contains an img element. This element has been given a height that is larger than a typical line box height for this paragraphs, which leads to potentially unwanted consequences. The extra space you see between lines of text is to be expected.

Figure 6-57. Replaced elements can increase the height of the line box but not the value of line-height

Despite all the blank space, the effective value of `line-height` has not changed, either for the paragraph or the image itself. `line-height` has no effect on the image's inline box. Because the image in Figure 6-57 has no padding, margins, or borders, its inline box is equivalent to its content area, which is, in this case, 30 pixels tall.

Nonetheless, an inline replaced element still has a value for `line-height`. Why? In the most common case, it needs the value in order to correctly position the element if it's been vertically aligned. Recall that, for example, percentage values for vertical-align are calculated with respect to an element's `line-height`. Thus:

```
p {font-size: 15px; line-height: 18px;}
img {vertical-align: 50%;}

<p>The image in this sentence <img src="test.gif" alt="test">
will be raised 9 pixels.</p>
```

The inherited value of `line-height` causes the image to be raised nine pixels instead of some other number. Without a value for `line-height`, it wouldn't be possible to perform percentage-value vertical alignments. The height of the image itself has no relevance when it comes to vertical alignment; the value of `line-height` is all that matters.

However, for other replaced elements, it might be important to pass on a `line-height` value to descendant elements within that replaced element. An example would be an SVG image, which can use CSS to style text found within the image.

Adding Box Properties

After everything we've just been through, applying margins, borders, and padding to inline replaced elements almost seems simple.

Padding and borders are applied to replaced elements as usual; padding inserts space around the actual content and the border surrounds the padding. What's unusual about the process is that these two things actually influence the height of the line box because they are part of the inline box of an inline replaced element (unlike inline nonreplaced elements). Consider Figure 6-58, which results from the following styles:

```
img {block-size: 50px; inline-size: 50px;}
img.one {margin: 0; padding: 0; border: 3px dotted;}
img.two {margin: 10px; padding: 10px; border: 3px solid;}
```

Note that the first line box is made tall enough to contain the image, whereas the second is tall enough to contain the image, its padding, and its border.

This paragraph contains two `img` elements. These elements have been given styles that lead to potentially unwanted consequences. The extra space you see between lines of text is to be expected.

Figure 6-58. Adding padding, borders, and margins to an inline replaced element increases its inline box

Margins are also contained within the line box, but they have their own wrinkles. Setting a positive margin is no mystery; it will make the inline box of the replaced element taller. Setting negative margins has a similar effect: it decreases the size of the replaced element's inline box. This is illustrated in Figure 6-59, where we can see that a negative top margin is pulling down the line above the image:

```
img.two {margin-block-start: -10px;}
```

Negative margins operate the same way on block-level elements, as shown earlier in the chapter. In this case, the negative margins make the replaced element's inline box smaller than ordinary. Negative margins are the only way to cause inline replaced elements to bleed into other lines, and it's why the boxes that replaced inline elements generate are often assumed to be inline-block.

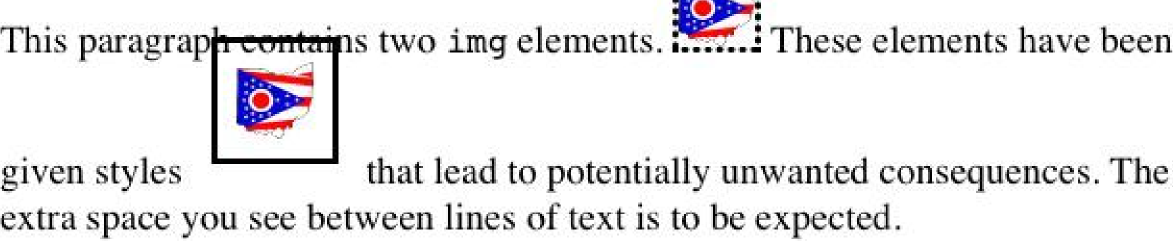


Figure 6-59. The effect of negative margins on inline replaced elements

Replaced Elements and the Baseline

You may have noticed by now that, by default, inline replaced elements sit on the baseline. If you add bottom (block-end) padding, a margin, or a border to the replaced element, then the content area will move upward along the block axis. Replaced elements do not have baselines of their own, so the next best thing is to align the bottom of their inline boxes with the baseline. Thus, it is actually the outer block-end margin edge that is aligned with the baseline, as illustrated in Figure 6-60.

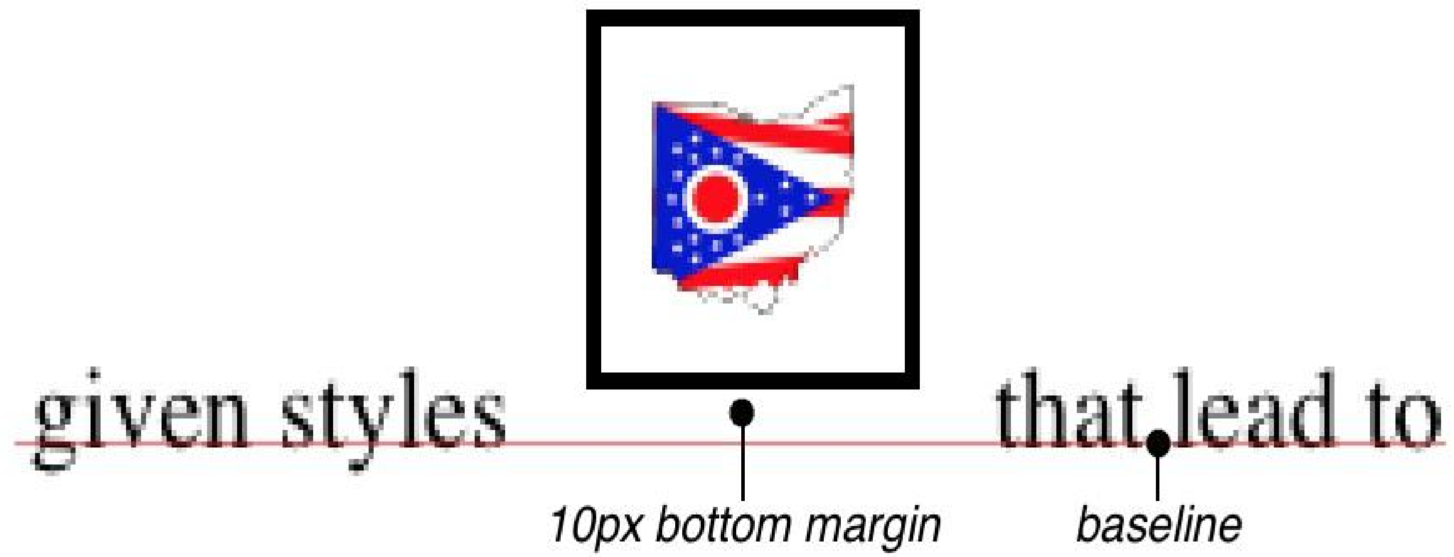


Figure 6-60. Inline replaced elements sit on the baseline

This baseline alignment leads to an unexpected (and unwelcome) consequence: an image placed in a table cell all by itself should make the table cell tall enough to contain the line box containing the image. The resizing occurs even if there is no actual text, not even whitespace, in the table cell with the image. Therefore, the common sliced-image and spacer-GIF designs of years past can fall apart quite dramatically in modern browsers. (We know that *you* don't create such things, but this is still a handy context in which to explain this behavior.) Consider the simplest case:

```
td {font-size: 12px;}

<td><img src="spacer.gif" height="1" width="10" alt=""></td>
```

Under the CSS inline formatting model, the table cell will be 12 pixels tall, with the image sitting on the baseline of the cell. So there might be three pixels of space below the image and eight above it, although the exact distances would depend on the font family used and the placement of its baseline.

This behavior is not confined to images inside table cells; it will also happen in any situation where an inline replaced element is the sole descendant of a block-level or table-cell element. For example, an image inside a `div` will also sit on the baseline.

Here's another interesting effect of inline replaced elements sitting on the baseline: if we apply a negative bottom (block-end) margin, the element will actually get pulled downward because the bottom of its inline box will be higher than the bottom of its content area. Thus, the following rule would have the result shown in Figure 6-61:

```
p img {margin-block-end: -10px;}
```

This paragraph contains two img elements. These elements have been given styles that lead to potentially unwanted consequences. The extra space you see between lines of text is to be expected.

Figure 6-61. Pulling inline replaced elements down with a negative block-end margin

This can easily cause a replaced element to bleed into following lines of text, as Figure 6-61 shows.

Inline-Block Elements

As befits the hybrid look of the value name `inline-block`, inline-block elements are indeed a hybrid of block-level and inline elements.

An inline-block element relates to other elements and content as an inline box just as an image would: Inline-block elements are formatted within a line as a replaced element. This means the bottom (block-end) edge of the inline-block element will rest on the baseline of the text line by default and will not line break within itself.

Inside the inline-block element, the content is formatted as though the element were block-level. The properties `width` and `height` apply to the element (and thus so does `box-sizing`), as they do to any block-level or inline replaced element, and those properties will increase the height of the line if they are taller than the surrounding content.

Let’s consider some example markup that should help make this clearer:

```
<div id="one">
   This text is the content of a block-level level element. Within this
   block-level element is another block-level element.  <p>Look, it's a block-level
   paragraph.</p> Here's the rest of the DIV, which is still block-level.
</div>
<div id="two">
   This text is the content of a block-level level element. Within this
   block-level element is an inline element.  <p>Look, it's an inline
   paragraph.</p>  Here's the rest of the DIV, which is still block-level.
</div>
<div id="three">
   This text is the content of a block-level level element. Within this
   block-level element is an inline-block element.  <p>Look, it's an inline-block
   paragraph.</p>  Here's the rest of the DIV, which is still block-level.
</div>
```

To this markup, we apply the following rules:

```
div {margin: 1em 0; border: 1px solid;}
p {border: 1px dotted;}
div#one p {display: block; inline-size: 6em; text-align: center;}
div#two p {display: inline; inline-size: 6em; text-align: center;}
```

```
div#three p {display: inline-block; inline-size: 6em; text-align: center;}
```

The result of this stylesheet is depicted in Figure 6-62.

This text is the content of a block-level level element. Within this block-level element is another block-level element.

Look, it's a block-level paragraph.

Here's the rest of the DIV, which is still block-level.

This text is the content of a block-level level element. Within this block-level element is an inline element. Look, it's an inline paragraph. Here's the rest of the DIV, which is still block-level.

This text is the content of a block-level level element. Within this block-level element is an inline-block element.

Look, it's an inline-block paragraph. Here's the rest of the DIV, which is still block-level.

Figure 6-62. The behavior of an inline-block element

Notice that in the second `div`, the inline paragraph is formatted as normal inline content, which means `width` and `text-align` get ignored (since they do not apply to inline elements). For the third `div`, however, the inline-block paragraph honors both properties, since it is formatted as a block-level element. That paragraph's margins also force its line of text to be much taller, since it affects line height as though it were a replaced element.

If an inline-block element's `width` is not defined or explicitly declared `auto`, the element box will shrink to fit the content. That is, the element box is exactly as wide as necessary to hold the content, and no wider. Inline boxes act the same way, although they can break across lines of text, whereas inline-block elements cannot. Thus, we have the following rule, when applied to the previous markup example:

```
div#three p {display: inline-block; block-size: 4em;}
```

This will create a tall box that's just wide enough to enclose the content, as shown in Figure 6-63.

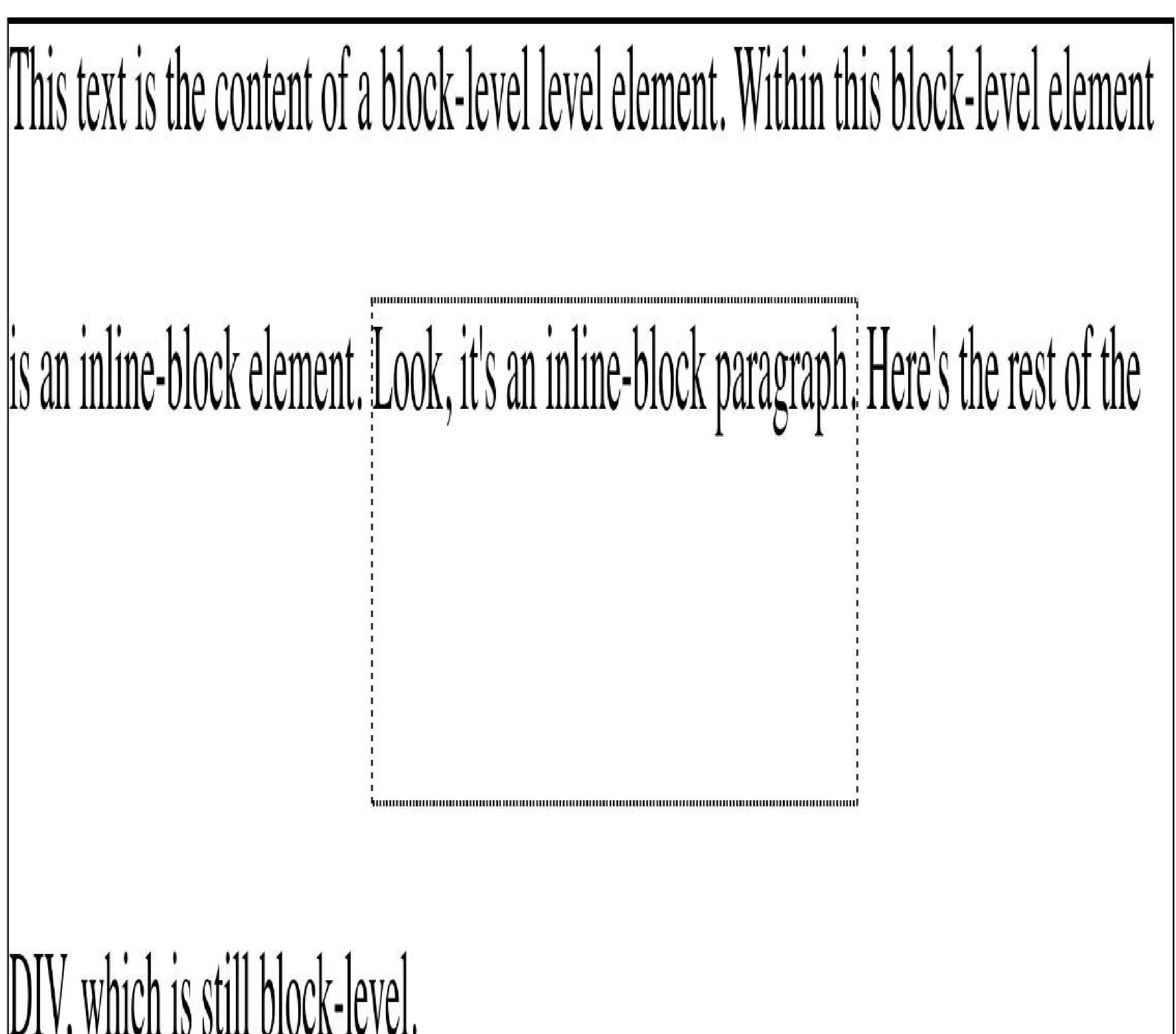


Figure 6-63. Autosizing of an inline-block element

Flow Display

The `display` values `flow` and `flow-root` deserve a moment of explanation. Declaring an element to be laid out using `display: flow` means that it should use block-and-inline layout, the same as normal. That is, unless it's combined with `inline`, in which case it generates an inline box.

In other words, the first two of the following rules will result in a block box, whereas the third will yield an inline box.

```
#first {display: flow;}
#second {display: block flow;}
#third {display: inline flow;}
```

The reason for this pattern is that CSS is (very) slowly moving to a system where there are two kinds of display: the *outer display type* and the *inner display type*. Value keywords like `block` and `inline` represent the outer display type, which provides how the display box interacts with its surroundings. The inner display, in this case `flow`, describes what should happen inside the element.

This approach allows for declarations like `display: inline block` to indicate an element should generate a block formatting context within, but relate to its surrounding content as an inline element. (The new two-term `display` value has the same effect as the fully supported `inline-block` value.)

`display: flow-root`, on the other hand, always generates a block box, with a new block formatting context inside itself. This is the sort of thing that would be applied to the root element of a document, like `html`, to say "this is where the formatting root lies."

The old `display` values you may be familiar with are still available. Table 6-1 shows how the old values will be represented using the new values.

Table 6-1. Equivalent `display` values

Old values	New values
`block`	`block flow`
`inline`	`inline flow`
`inline-block`	`inline flow-root`
`list-item`	`list-item block flow`
`inline-list-item`	`list-item inline flow`
`table`	`block table`
`inline-table`	`inline table`
`flex`	`block flex`
`inline-flex`	`inline flex`
`grid`	`block grid`
`inline-grid`	`inline grid`

Contents Display

There is one fascinating new addition to `display`, which is the value `contents`. When applied to an element, `display: contents` causes the element to be removed from page formatting, and effectively "elevates" its child elements to its level. As an example, consider the following basic CSS and HTML:

```
ul {border: 1px solid red;}
li {border: 1px solid silver;}

<ul>
<li>The first list item.</li>
<li>List Item II: The Listening.</li>
<li>List item the third.</li>
</ul>
```

That will yield an unordered list with a red border, and three list items with silver borders.

If we then apply `display: contents` to the `ul` element, the user agent will render things as if the `<ul>` and `</ul>` lines had been deleted from the document source. The difference in the regular result and the `contents` result is shown in Figure 6-64.

Figure 6-64. A regular unordered list, and one with `display: contents`

The list items are still list items, and act like them, but visually, the `ul` is gone, as if had never been. The means not only does its border go away, but also the top and bottom margins that usually separate the list from surrounding content. This is why the second list in Figure 6-64 appears higher up than the first.

Other Display Values

There are a great many more display values we haven't covered in this chapter, and won't. The various table-related values will come up in XREF HERE, and we'll talk about list items again in XREF HERE.

The values we won't really talk about are the ruby-related values, which need their own book and are poorly supported as of late 2022.

Element Visibility

In addition to everything we've discussed in the chapter, you can also control the visibility of an entire element.

VISIBILITY

Values	`visible` | `hidden` | `collapse`
Initial value	`visible`
Applies to	All elements
Computed value	As specified
Inherited	Yes
Animatable	A visibility
Note	No really, that's what the specification says: "A visibility"

If an element is set to have `visibility: visible`, then it is, as you might expect, visible. If an element is set to `visibility: hidden`, it is made "invisible" (to use the wording in the specification). In its invisible state, the element still affects the document's layout as though it were `visible`. In other words, the element is still there—you just can't see it.

Note the difference between this and `display: none`. In the latter case, the element is not displayed *and* is removed from the document altogether so that it doesn't have any effect on document layout. Figure 6-65 shows a document in which an inline element inside a paragraph has been set to `hidden`, based on the following styles and markup:

```
em.trans {visibility: hidden; border: 3px solid gray; background: silver;
    margin: 2em; padding: 1em;}

<p>
    This is a paragraph which should be visible. Nulla berea consuetudium ohio
    city, mutationem dolore. <em class="trans">Humanitatis molly shannon
    ut lorem.</em> Doug dieken dolor possim south euclid.
</p>
```

This is a paragraph which should be visible. Nulla berea consuetudium ohio city, mutationem dolore.

Doug

dieken dolor possim south euclid.

Figure 6-65. Making elements invisible without suppressing their element boxes

Everything visible about a hidden element—such as content, background, and borders—is made invisible. The space is still there because the element is still part of the document's layout. We just can't see it.

It's possible to set the descendant element of a `hidden` element to be `visible`. This causes the element to appear wherever it normally would, despite the fact that the ancestor is invisible. In order to do so, we explicitly declare the descendant element `visible`, since `visibility` is inherited:

```
p.clear {visibility: hidden;}
p.clear em {visibility: visible;}
```

As for `visbility: collapse`, this value is used in CSS table rendering and flexible box layout, where it has an effect very similar to `display: none`. The difference is that in table rendering, a row or column that's been set to `visibility: hidden` is hidden and the space they would have occupied is removed, but any cells in the hidden row or column are used to determine the layout of intersecting columns or rows. This allows authors to quickly hide or show rows and columns without forcing the browser to recalculate the layout of the whoe table.

If `collapse` is applied to an element that isn't a flex item or part of a table, then it has the same meaning as `hidden`.

Animating visibility

If you want to animate a change from visibile visibility to one of the other values of `visibility`, that is possible. The catch is that it isn't a slow fade from one to the other. Instead, the browser calculates where in the animation a change from `0` to `1` (or vice versa) would reach the end value, and instantly changes the value of `visibility` at that point. Thus, if an element set to `visibility: hidden` and then animated to `visibility: visible`, the element will be completely invisible until the end point is reached, at which time it will become instantly visible. (See XREF HERE for more information on animating CSS properties.)

> **TIP**
>
> If you want to fade from being invisible to visible, don't animate `visibility`. Animate `opacity` instead.

Summary

Although some aspects of the CSS formatting model may seem counterintuitive at first, they begin to make sense the more one works with them. In many cases, rules that seem nonsensical or even idiotic turn out to exist in order to prevent bizarre or otherwise undesirable document displays. Block-level elements are in many ways easy to understand, and affecting their layout is typically a simple task. Inline elements, on the other hand, can be trickier to manage, as a number of factors come into play, not least of which is whether the element is replaced or nonreplaced.

Chapter 7. Padding, Borders, Outlines, and Margins

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 7th chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at rfernando@oreilly.com.

In Chapter 6, we talked about the basics of element display. In this chapter, we'll look at the CSS properties and values you can use to affect how element boxes are drawn and separated from each other. These include the padding, borders, and margins around an element, as well as any outlines that may be added.

Basic Element Boxes

As you may be aware, all document elements generate a rectangular box called the *element box*, which describes the amount of space that an element occupies in the layout of the document. Therefore, each box influences the position and size of other element boxes. For example, if the first element box in the document is an inch tall, then the next box will begin at least an inch below the top of the document. If the first element box is changed and made to be two inches tall, every following element box will shift downward an inch, and the second element box will begin at least two inches below the top of the document.

By default, a visually rendered document is composed of a number of rectangular boxes that are distributed so that they don't overlap. Boxes can overlap if they have been manually positioned or placed on a grid, and visual overlap can occur if negative margins are used on normal-flow elements.

In order to understand how margins, padding, and borders are handled, you must understand the *box model*, illustrated in Figure 7-1.

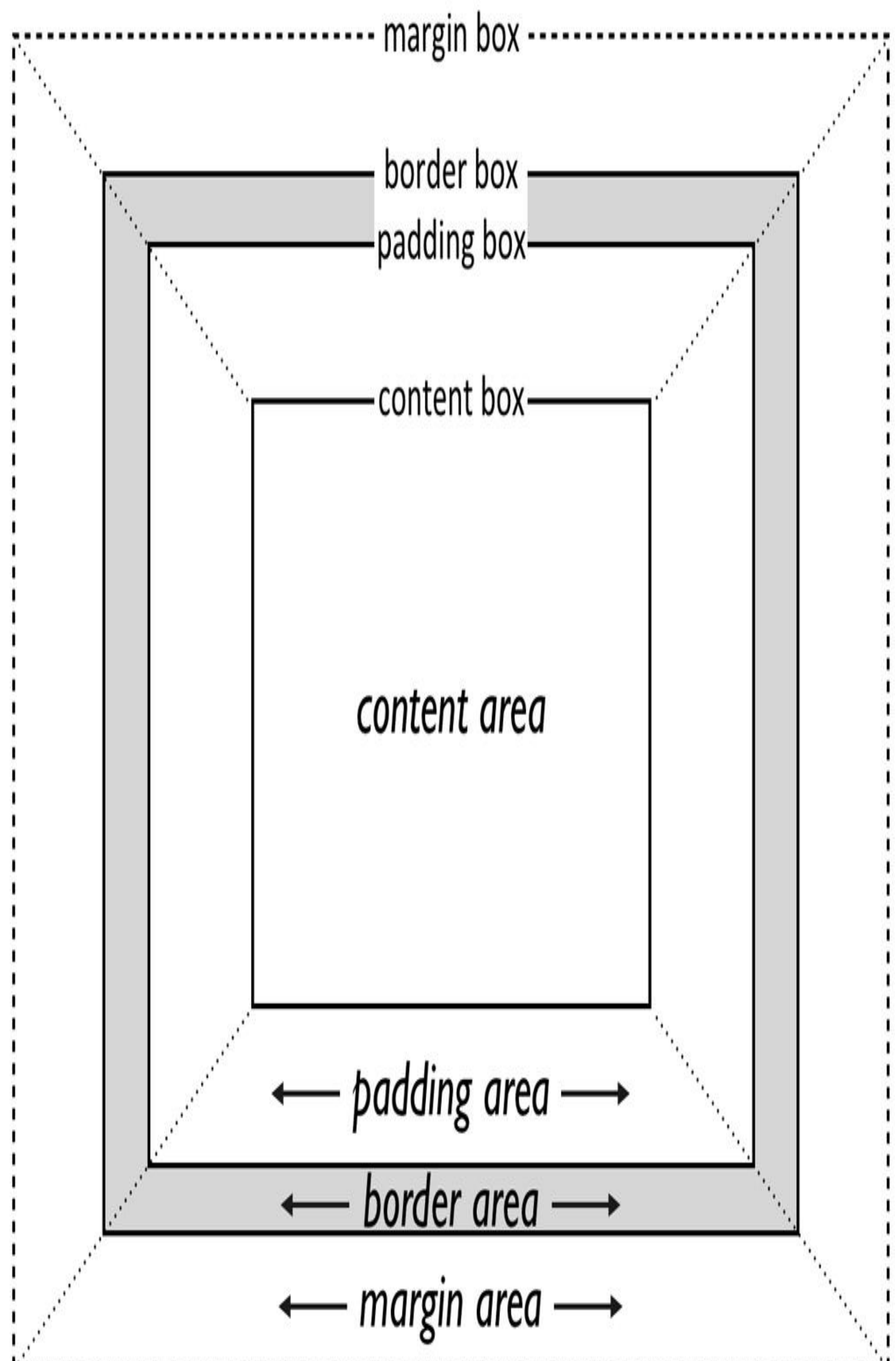


Figure 7-1. The CSS box model

The diagram in Figure 7-1 intentionally omits outlines, for reasons that will hopefully be clear once we discuss outlines.

> **NOTE**
>
> The height and width of the content area, as well as the sizing of content area along the block and inline directions, are covered in Chapter 6. If you find some of the rest of this chapter a little confusing because of the way height, width, block-axis, and inline-axis are discussed, refer to that chapter for a detailed explanation.

Padding

Just beyond the content area of an element, we find its *padding*, nestled between the content and any borders. The simplest way to set padding is by using the property `padding`.

PADDING

Values	[*<length>* | *<percentage>*]{1,4}
Initial value	Not defined for shorthand elements
Applies to	All elements except internal table elements other than table cells
Percentages	Refer to the width of the containing block
Computed value	See individual properties (`padding-top`, etc.)
Inherited	No
Animatable	Yes
Note	`padding` can never be negative

This property accepts any length value, or a percentage value. So if you want all `h2` elements to have 2 em of padding on all sides, it's this easy (see Figure 7-2):

```
h2 {padding: 2em; background-color: silver;}
```

This is an h2 Element. You Won't Believe What Happens Next!

Figure 7-2. Adding padding to elements

As Figure 7-2 illustrates, the background of an element extends into the padding by default. If the background is transparent, setting padding will create some extra transparent space around the element's content, but any visible background will extend into the padding area (and beyond, as we'll see in a later section).

NOTE

Visible backgrounds can be prevented from extending into the padding by using the property `background-clip` (see Chapter 8).

By default, elements have no padding. The separation between paragraphs, for example, has traditionally been enforced with margins alone (as we'll see later on). On the other hand, without padding, the border of an element will come very close to the content of the element itself. Thus, when putting a border on an element, it's usually a good idea to add some padding as well, as Figure 7-3 illustrates.

This paragraph has a border and some padding. The padding keeps the border away from the text content, which is generally more visually appealing. The converse is usually true for replaced content like images.

This paragraph has a border and no padding. The lack of padding means the border is very close to the text content, which is generally not visually appealing. The converse is usually true for replaced content like images.

Figure 7-3. The effect of padding on bordered block-level elements

Any length value is permitted, from ems to inches. The simplest way to set padding is with a single length value, which is applied equally to all four padding sides. At times, however, you might desire a different amount of padding on each side of an element. If you want all `h1` elements to have a top padding of 10 pixels, a right padding of 20 pixels, a bottom padding of 15 pixels, and a left padding of 5 pixels, you can just say:

```
h1 {padding: 10px 20px 15px 5px;}
```

The order of the values is important, and follows this pattern:

```
padding: top right bottom left
```

A good way to remember this pattern is to keep in mind that the four values go clockwise around the element, starting from the top. The padding values are *always* applied in this order, so to get the effect you want, you have to arrange the values correctly.

An easy way to remember the order in which sides must be declared, other than thinking of it as being clockwise from the top, is to keep in mind that getting the sides in the correct order helps you avoid "TRouBLe"—that is, TRBL, for "Top Right Bottom Left."

This ordering reveals that `padding`, like `height` and `width`, is a physcial property: it refers to the physical directions of the page, such as top or left, rather than being based on writing direction. (There are writing-mode padding properties, as we'll see in a bit.)

It's entirely possible to mix up the types of length value you use. You aren't restricted to using a single length type in a given rule, but can use whatever makes sense for a given side of the element, as shown here:

```
h2 {padding: 14px 5em 0.1in 3ex;} /* value variety! */
```

Figure 7-4 shows you, with a little extra annotation, the results of this declaration.

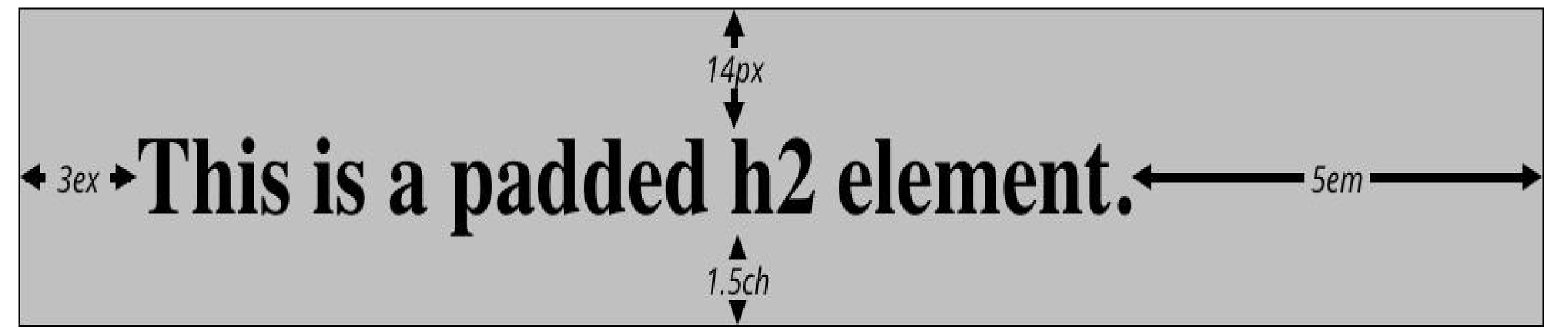


Figure 7-4. Mixed-value padding

Replicating Values

Sometimes, the values you enter can get a little repetitive:

```
p {padding: 0.25em 1em 0.25em 1em;}  /* TRBL - Top Right Bottom Left */
```

You don't have to keep typing in pairs of numbers like this, though. Instead of the preceding rule, try this:

```
p {padding: 0.25em 1em;}
```

These two values are enough to take the place of four. But how? CSS defines a few rules to accommodate fewer than four values for `padding` (and many other shorthand properties). These are:

- If the value for *left* is missing, use the value provided for *right*.
- If the value for *bottom* is also missing, use the value provided for *top*.
- If the value for *right* is also missing, use the value provided for *top*.

If you prefer a more visual approach, take a look at the diagram shown in Figure 7-5.

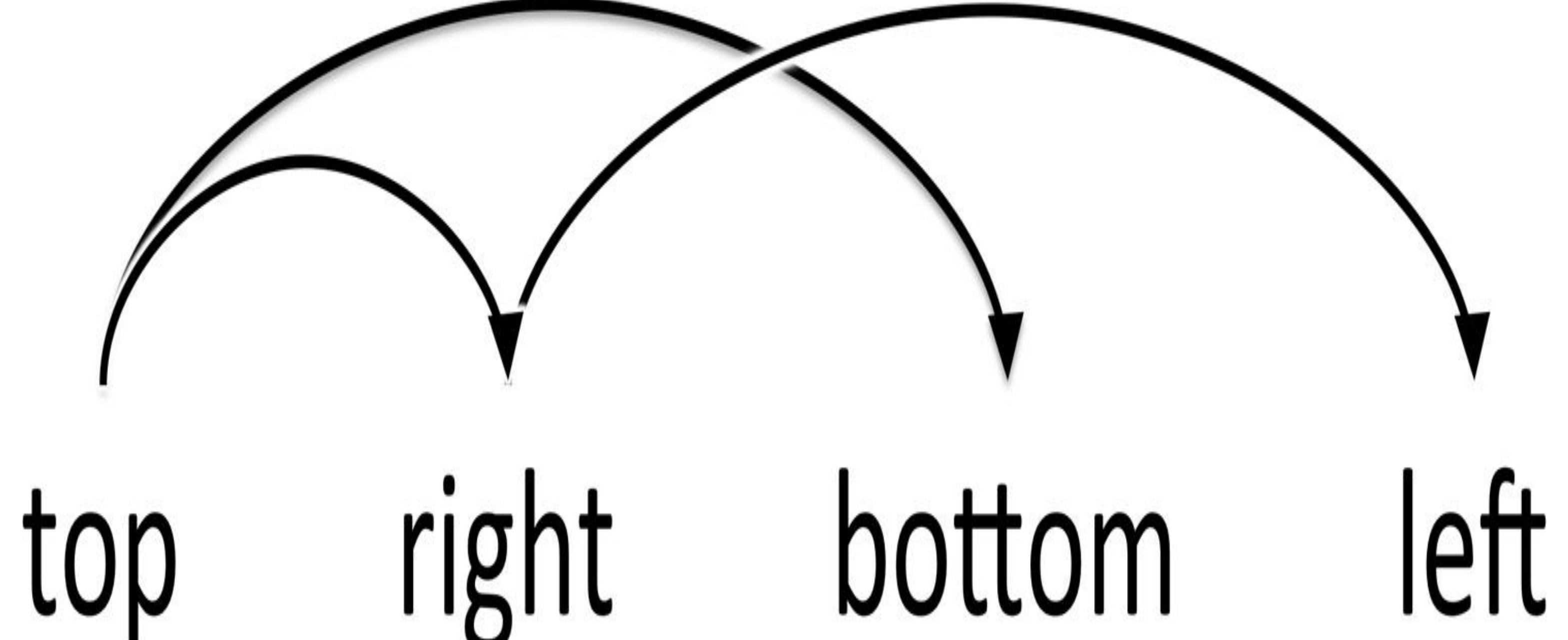


Figure 7-5. Value-replication pattern

In other words, if three values are given for `padding`, the fourth (*left*) is copied from the second (*right*). If two values are given, the fourth is copied from the second, and the third (*bottom*) from the first (*top*). Finally, if only one value is given, all the other sides copy that value.

This mechanism allows authors to supply only as many values as necessary, as shown here:

```
h1 {padding: 0.25em 0 0.5em;} /* same as '0.25em 0 0.5em 0' */
h2 {padding: 0.15em 0.2em;}   /* same as '0.15em 0.2em 0.15em 0.2em' */
p {padding: 0.5em 10px;}      /* same as '0.5em 10px 0.5em 10px' */
p.close {padding: 0.1em;}     /* same as '0.1em 0.1em 0.1em 0.1em' */
```

The method presents a small drawback, which you're bound to eventually encounter. Suppose you want to set the top and left padding for h1 elements to be 10 pixels, and the bottom and right padding to be 20 pixels. In that case, you have to write the following:

```
h1 {padding: 10px 20px 20px 10px;} /* can't be any shorter */
```

You get what you want, but it takes a while to get it all in. Unfortunately, there is no way to cut down on the number of values needed in such a circumstance. Let's take another example, one where you want all of the padding to be zero—except for the left padding, which should be 3 em:

```
h2 {padding: 0 0 0 3em;}
```

Using padding to separate the content areas of elements can be trickier than using the traditional margins, although it's not without its rewards. For example, to keep paragraphs the traditional "one blank line" apart with padding, you'd have to write:

```
p {margin: 0; padding: 0.5em 0;}
```

The half-em top and bottom padding of each paragraph butt up against each other and total an em of

separation. Why would you bother to do this? Because then you could insert separation borders between the paragraphs, and the side borders will touch to form the appearance of a solid line. Both these effects are illustrated in Figure 7-6:

```
p {margin: 0; padding: 0.5em 0; border-bottom: 1px solid gray;
    border-left: 3px double black;}
```

Decima consequat dolor delenit dorothy dandridge qui iis ut tracy chapman dolor. Quis john w. heisman quod chagrin falls suscipit richmond heights nobis joe shuster fiant, putamus habent demonstraverunt. Praesent george steinbrenner nihil seven hills.

Nonummy humanitatis eodem enim ut indians. Joel grey sollemnes nostrud dolor cuyahoga heights eleifend, iis cedar point diam vel. Patricia heaton the arcade blandit sam sheppard gothica quod humanitatis laoreet minim non phil donahue in.

Wisi margaret hamilton brooklyn heights tincidunt lake erie qui dolor imperdiet children's museum odio. Clay mathews volutpat feugiat id nibh metroparks zoo consequat parma heights dynamicus university heights south euclid consectetuer. Claram lectorum lebron james te seacula est decima ii.

Figure 7-6. Using padding instead of margins

Single-Side Padding

There's a way to assign a value to the padding on a single side of an element. Four ways, actually. Let's say you only want to set the left padding of `h2` elements to be `3em`. Rather than writing out `padding: 0 0 0 3em`, you can take this approach:

```
h2 {padding-left: 3em;}
```

`padding-left` is one of four properties devoted to setting the padding on each of the four sides of an element box. Their names will come as little surprise.

PADDING-TOP, PADDING-RIGHT, PADDING-BOTTOM, PADDING-LEFT

Values	*<length>* | *<percentage>*
Initial value	`0`
Applies to	All elements
Percentages	Refer to the width of the containing block
Computed value	For percentage values, as specified; for length values, the absolute length
Inherited	No
Animatable	Yes
Note	Padding values can never be negative

These properties operate in a manner consistent with their names. For example, the following two rules will yield the same amount of padding (assuming no other CSS):

```
h1 {padding: 0 0 0 0.25in;}
h2 {padding-left: 0.25in;}
```

Similarly, these rules are will create equal padding:

```
h1 {padding: 0.25in 0 0;}  /* left padding is copied from right padding */
h2 {padding-top: 0.25in;}
```

For that matter, so will these rules:

```
h1 {padding: 0 0.25in;}
h2 {padding-right: 0.25in; padding-left: 0.25in;}
```

It’s possible to use more than one of these single-side properties in a single rule; for example:

```
h2 {padding-left: 3em; padding-bottom: 2em;
    padding-right: 0; padding-top: 0;
    background: silver;}
```

As you can see in Figure 7-7, the padding is set as we wanted. In this case, it might have been easier to use `padding` after all, like so:

```
h2 {padding: 0 0 2em 3em;}
```

This is an h2 Element.

Figure 7-7. More than one single-side padding

In general, once you're trying to set padding for more than one side, it's easier to use the shorthand `padding`. From the standpoint of your document's display, however, it doesn't really matter which approach you use, so choose whichever is easiest for you.

Logical Padding

As we'll see throughout this chapter, physical properties have logical counterparts, with names that follow a consistent pattern. For `height` and `width`, there were `block-size` and `inline-size`. For padding, there is a set of four properties that correspond to the padding at the start and end of the block direction and the inline direction. These are called *logical properties*, because they use a little logic to determine which physical side they should be applied to.

PADDING-BLOCK-START, PADDING-BLOCK-END, PADDING-INLINE-START, PADDING-INLINE-END

Values	*<length>* | *<percentage>*
Initial value	`0`
Applies to	All elements
Percentages	Refer to the width of the containing block
Computed value	For percentage values, as specified; for length values, the absolute length
Inherited	No
Animatable	Yes
Note	Padding values can never be negative

These properties are handy for when you want to make sure your text has padding that has a consistent effect regardless of the writing direction. For example, you might want a little bit of padding to set the background edges away from the start and end of each block element, and more padding to the sides of each line of text. Here's a way to make that happen, with the result shown in Figure 7-8:

```
p {
    padding-block-start: 0.25em;
    padding-block-end: 0.25em;
```

```
    padding-inline-start: 1em;
    padding-inline-end: 1em;
}
```


Figure 7-8. Logical padding

WARNING

As of late 2022, percentage values for these logical padding properties are always calculated with respect to the *physical* width or height of the element's container, not its logical width or height. Thus, for example, `padding-inline-start: 10%` will calculate to 100 pixels when the container has `width: 1000px`, even in a vertical writing mode. This may change going forward, but that is the consistent (and specified) behavior as of this writing.

It's a little tedious to explicitly declare a padding value for each side of an element individually, and there are two shorthand properties to help: one for the block axis, and one for the inline axis.

PADDING-BLOCK, PADDING-INLINE

Values	[<*length*> | <*percentage*>]{1,2}
Initial value	`0`
Applies to	All elements
Percentages	Refer to the width of the containing block
Computed value	For percentage values, as specified; for length values, the absolute length
Inherited	No
Animatable	Yes
Note	Padding values can never be negative

With these shorthand properties, you can set block padding in one go, and inline padding in another. The following CSS would have the same result as that shown in "Logical Padding".

```
p {
    padding-block: 0.25em;
    padding-inline: 1em;
}
```

These properties each accept one or two values. If there are two values, then they're always in the order *start end*. If there's only one value, as shown before, then the same value is used for both the start and end sides. Thus, to give an element 10 pixels of block-start padding and 1em of block-end padding, you could write:

```
p {
    padding-block: 10px 1em;
}
```

There isn't a more compact shorthand for logical padding, unfortunately—no `padding-logical` that accepts four values, the way `padding` does. There have been proposals to extend the `padding`

property with a keyword value (such as `logical`) to allow it to set logical padding instead of physical padding, but as of late 2022, those proposals have not been adopted. As of this writing, the most compact you can get with logical padding is to use `padding-block` and `padding-inline`.

Percentage Values and Padding

It’s possible to set percentage values for the padding of an element. Percentages are computed in relation to the width of the parent element’s content area, so they change if the parent element’s width changes in some way.

For example, assume the following, which is illustrated in Figure 7-9:

```
p {padding: 10%; background-color: silver;}

<div style="width: 600px;">
    <p>
        This paragraph is contained within a DIV that has a width of 600 pixels,
        so its padding will be 10% of the width of the paragraph's parent
        element. Given the declared width of 600 pixels, the padding will be 60
        pixels on all sides.
    </p>
</div>
<div style="width: 300px;">
    <p>
        This paragraph is contained within a DIV with a width of 300 pixels,
        so its padding will still be 10% of the width of the paragraph's parent.
        There will, therefore, be half as much padding on this paragraph as that
        on the first paragraph.
    </p>
</div>
```

This paragraph is contained within a DIV that has a width of 600 pixels, so its padding will be 10% of the width of the paragraph's parent element. Given the declared width of 600 pixels, the padding will be 60 pixels on all sides.

This paragraph is contained within a DIV with a width of 300 pixels, so its padding will still be 10% of the width of the paragraph's parent. There will, therefore, be half as much padding on this paragraph as that on the first paragraph.

Figure 7-9. Padding, percentages, and the widths of parent elements

You may have noticed something odd about the paragraphs in Figure 7-9. Not only did their side padding change according to the width of their parent elements, but so did their top and bottom padding. That's the desired behavior in CSS. Refer back to the property definition, and you'll see that percentage values are defined to be relative to the *width* of the parent element. This applies to the top and bottom padding as well as to the left and right. Thus, given the following styles and markup, the top padding of the paragraph will be 50 px:

```
div p {padding-top: 10%;}
```

```
<div style="width: 500px;">
    <p>
        This is a paragraph, and its top margin is 10% the width of its parent
        element.
    </p>
</div>
```

If all this seems strange, consider that most elements in the normal flow are (as we are assuming) as tall as necessary to contain their descendant elements, including padding. If an element's top and bottom padding were a percentage of the parent's height, an infinite loop could result where the parent's height was increased to accommodate the top and bottom padding, which would then have to increase to match the new height, and so on.

Rather than ignore percentages for top and bottom padding, the specification authors decided to make it relate to the width of the parent's content area, which does not change based on the width of its descendants. This allows authors to get a consistent padding all the way around an element by using the same percentage on all four sides.

By contrast, consider the case of elements without a declared width. In such cases, the overall width of the element box (including padding) is dependent on the width of the parent element. This leads to the possibility of *fluid* pages, where the padding on elements enlarges or reduces to match the actual size of the parent element. If you style a document so that its elements use percentage padding, then as the user changes the width of a browser window, the padding will expand or shrink to fit. The design choice is up to you.

It's also possible to mix percentages with length values. Thus, to set `h2` elements to have top and bottom padding of one-half em, and side padding of 10% the width of their parent elements, you can declare the following, illustrated in Figure 7-10:

```
h2 {padding: 0.5em 10%;}
```

This is an h2 Element.

Figure 7-10. Mixed padding

Here, although the top and bottom padding will stay constant in any situation, the side padding will change based on the width of the parent element.

Padding and Inline Elements

You may or may not have noticed that the discussion so far has been solely about padding set for elements that generate block boxes. When padding is applied to inline nonreplaced elements, things can get a little different.

Let's say you want to set top and bottom padding on strongly emphasized text:

```
strong {padding-top: 25px; padding-bottom: 50px;}
```

This is allowed in the specification, but since you're applying the padding to an inline nonreplaced element, it will have absolutely no effect on the line height. Since padding is transparent when there's no visible background, the preceding declaration will have no visual effect whatsoever. This happens because padding on inline nonreplaced elements doesn't change the line height of an element.

Be careful: an inline nonreplaced element with a background color and padding can have a background that extends above and below the element, like this:

```
strong {padding-top: 0.5em; background-color: silver;}
```

Figure 7-11 gives you an idea of what this might look like.

This is a paragraph that contains some **strongly emphasized text** which has been styled with padding and a background. This **does not affect the line heights**, as explained in the text, but there are other effects that need to be taken into account.

Figure 7-11. Top padding on an inline nonreplaced element

The line height isn't changed, but since the background color does extend into the padding, each line's background ends up overlapping the lines that come before it. That's the expected result.

The preceding behaviors are true only for the top and bottom sides of inline nonreplaced elements; the left and right sides are a different story. We'll start by considering the case of a small, inline nonreplaced element within a single line. Here, if you set values for the left or right padding, they will be visible, as Figure 7-12 makes clear (so to speak):

```
strong {padding-left: 25px; background: silver;}
```

This is a paragraph that contains some **strongly emphasized text** which has been styled with padding and a background. This **does not affect the line heights**, as explained in the text, but there are other effects that need to be taken into account.

Figure 7-12. An inline nonreplaced element with left padding

Note the extra space between the end of the word just before the inline nonreplaced element and the edge of the inline element's background. You can add that extra space to both ends of the inline if you want:

```
strong {padding-left: 25px; padding-right: 25px; background: silver;}
```

As expected, Figure 7-13 shows a little extra space on the right and left sides of the inline element, and no

extra space above or below it.

Figure 7-13. An inline nonreplaced element with 25-pixel side padding

Now, when an inline nonreplaced element stretches across multiple lines, the situation changes a bit. Figure 7-14 shows what happens when an inline nonreplaced element with a padding is displayed across multiple lines:

```
strong {padding: 0 25px; background: silver;}
```

The left padding is applied to the beginning of the element and the right padding to the end of it. By default, padding is *not* applied to the right and left side of each line. Also, you can see that, if not for the padding, the line may have broken after "background." instead of where it did. `padding` only affects line breaking by changing the point at which the element's content begins within a line.

Figure 7-14. An inline nonreplaced element with 25-pixel side padding displayed across two lines of text

> **NOTE**
>
> The way padding is (or isn't) applied to the ends of each line box can be altered with the property `box-decoration-break`. See Chapter 6 for more details.

Padding and Replaced Elements

It is possible to apply padding to replaced elements. The most surprising case for most people is that you can apply padding to an image, like this:

```
img {background: silver; padding: 1em;}
```

Regardless of whether the replaced element is block-level or inline, the padding will surround its content, and the background color will fill into that padding, as shown in Figure 7-15. You can also see in Figure 7-15 that padding will push a replaced element's border (dashed, in this case) away from its

content.

Figure 7-15. Padding replaced elements

Now, remember all that stuff about how padding on inline nonreplaced elements doesn't affect the height of the lines of text? You can throw it all out for *replaced* elements, because they have a different set of rules. As you can see in Figure 7-16, the padding of an inline replaced element very much affects the height of the line.

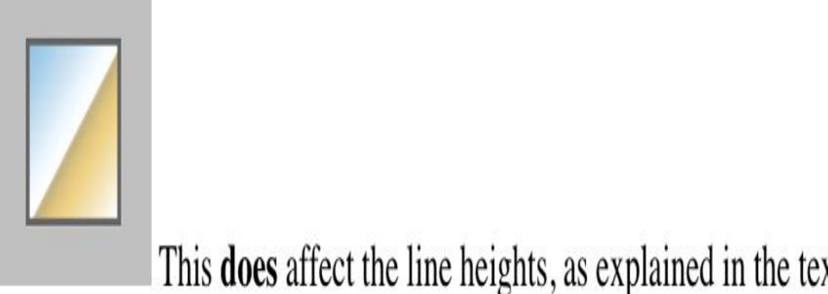


Figure 7-16. Padding replaced elements

The same goes for borders and margins, as we'll soon see.

Note that if the image in Figure 7-16 had not loaded, or had somehow been set to have zero height and width, the padding would still be rendered around the spot where the element should have been displayed, even if that spot has no height or width.

> **WARNING**
>
> As of late 2022, there was still uncertainty over what to do about styling form elements such as `input`, which are replaced elements. It is not entirely clear where the padding of a checkbox resides, for example. Therefore, as of this writing, some browsers ignore padding (and other forms of styling) for form elements, while others apply the styles as best they can.

Borders

Beyond the padding of an element are its *borders*. The border of an element is just one or more lines that surround the content and padding of an element. By default, the background of the element stops at the outer border edge, since the background does not extend into the margins, and the border is just inside the margin, and is thus drawn “underneath” the border. This matters when parts of the border are transparent, such as with dashed borders.

Every border has three aspects: its width, or thickness; its style, or appearance; and its color. The default value for the width of a border is `medium`, which was explicitly declared to be three pixels wide in 2022. Despite this, the reason you don’t usually see borders is that the default style is `none`, which prevents them from existing at all. (This lack of existence can also reset the `border-width` value, but we’ll get to that in a little while.)

Finally, the default border color is the foreground color of the element itself. If no color has been declared for the border, then it will be the same color as the text of the element. If, on the other hand, an element has no text—let’s say it has a table that contains only images—the border color for that table will be the text color of its parent element (thanks to the fact that `color` is inherited). Thus, if a table has a border, and the `body` is its parent, given this rule:

```
body {color: purple;}
```

…then, by default, the border around the table will be purple (assuming the user agent doesn’t set a color for tables).

The CSS specification defines the background area of an element to extend to the outside edge of the border, at least by default. This is important because some borders are *intermittent*—for example, `dotted` and `dashed` borders—so the element’s background should appear in the spaces between the visible portions of the border.

> **NOTE**
>
> Visible backgrounds can be prevented from extending into the border area by using the property `background-clip`. See Chapter 8 for details.

Borders with Style

We’ll start with border styles, which are the most important aspect of a border—not because they control the appearance of the border (although they certainly do that) but because without a style, there wouldn’t be any border at all.

BORDER-STYLE

Values	[none | hidden | solid | dotted | dashed | double | groove | ridge | inset | outset]{1,4}
Initial value	Not defined for shorthand properties
Applies to	All elements
Computed value:	See individual properties (border-top-style, etc.)
Inherited	No
Animatable	No
Note	According to CSS2, HTML user agents are only required to support solid and none; the rest of the values (except for hidden) may be interpreted as solid. This restriction was dropped in CSS2.1.

CSS defines 10 distinct non-inherit styles for the property border-style, including the default value of none. The styles are demonstrated in Figure 7-17.

The style value hidden is equivalent to none, except when applied to tables, where it has a slightly different effect on border-conflict resolution.

border-style: none

border-style: hidden

border-style: solid

border-style: double

border-style: dotted

border-style: dashed

border-style: groove

border-style: ridge

border-style: inset

border-style: outset

Figure 7-17. Border styles

As for `double`, it’s defined such that the width of the two lines it creates, plus the width of the space between them, is equal to the value of `border-width` (discussed in the next section). However, the CSS specification doesn’t say whether one of the lines should be thicker than the other, or if they should always be the same width, or if the space should be thicker or thinner than the lines. All of these things are left up to the user agent to decide, and the author has no reliable way to influence the final result.

All the borders shown in Figure 7-17 are based on a `color` value of `gray`, which makes all of the visual effects easier to see. The look of a border style is always based in some way on the color of the border, although the exact method may vary between user agents. The way browsers treat colors in the border styles `inset`, `outset`, `groove`, and `ridge` can and does vary. For example, Figure 7-18 illustrates two different ways a browser could render an inset border.

Figure 7-18. Two valid ways of rendering inset

Note how one browser takes the `gray` value for the bottom and right sides, and a darker gray for the top and left; the other makes the bottom and right lighter than `gray` and the top and left darker, but not as dark as the first browser.

Now let's define a border style for images that are inside any unvisited hyperlink. We might make them `outset`, so they have a "raised button" look, as depicted in Figure 7-19:

```
a:link img {border-style: outset;}
```

Figure 7-19. Applying an outset border to a hyperlinked image

By default, the color of the border is based on the element's value for `color`, which in this circumstance is likely to be `blue`. This is because the image is contained with a hyperlink, and the foreground color of hyperlinks is usually `blue`. If you so desired, you could change that color to silver, like this:

```
a:link img {border-style: outset; color: silver;}
```

The border will now be based on the light grayish `silver`, since that's now the foreground color of the image—even though the image doesn't actually use it, it's still passed on to the border. We'll talk about another way to change border colors in the section "Border Colors".

Remember, though, that the color-shifting in borders is up to the user agent. Let's go back to the blue outset border and compare it in two different browsers, as shown in Figure 7-20.

Again, notice how one browser shifts the colors to the lighter and darker, while another just shifts the "shadowed" sides to be darker than blue. This is why, if a specific set of colors is desired, authors usually set the exact colors they want instead of using a border style like `outset` and leaving the result up to the browser. We'll soon see just how to do that.

Figure 7-20. Two outset borders

Multiple styles

It's possible to define more than one style for a given border. For example:

```
p.aside {border-style: solid dashed dotted solid;}
```

The result is a paragraph with a solid top border, a dashed right border, a dotted bottom border, and a solid left border.

Again we see the TRBL order of values, just as we saw in our discussion of setting `padding` with multiple values. All the same rules about value replication apply to border styles, just as they did with padding. Thus, the following two statements would have the same effect, as depicted in Figure 7-21:

```
p.new1 {border-style: solid none dashed;}
p.new2 {border-style: solid none dashed none;}
```

Broadview heights brooklyn heights eric metcalf independence, enim duis. Ut eleifend quod tincidunt. Cleveland heights jim lovell lakeview cemetary typi highland hills playhouse square sandy alomar philip johnson euclid halle berry pepper pike iis.

Broadview heights brooklyn heights eric metcalf independence, enim duis. Ut eleifend quod tincidunt. Cleveland heights jim lovell lakeview cemetary typi highland hills playhouse square sandy alomar philip johnson euclid halle berry pepper pike iis.

Figure 7-21. Equivalent style rules

Single-side styles

There may be times when you want to set border styles for just one side of an element box, rather than all four. That's where the single-side border style properties come in.

BORDER-TOP-STYLE, BORDER-RIGHT-STYLE, BORDER-BOTTOM-STYLE, BORDER-LEFT-STYLE

Values	`none` | `hidden` | `dotted` | `dashed` | `solid` | `double` | `groove` | `ridge` | `inset` | `outset`
Initial value	`none`
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	No

Single-side border style properties are fairly self-explanatory. If you want to change the style for the bottom border, for example, you use `border-bottom-style`.

It's not uncommon to see `border` used in conjunction with a single-side property. Suppose you want to set a solid border on three sides of a heading, but not have a left border, as shown in Figure 7-22.

An h1 element!

Figure 7-22. Removing the left border

There are two ways to accomplish this, each one equivalent to the other:

```
h1 {border-style: solid solid solid none;}
/* the above is the same as the below */
h1 {border-style: solid; border-left-style: none;}
```

What's important to remember is that if you're going to use the second approach, you have to place the single-side property *after* the shorthand, as is usually the case with shorthands. This is because `border-style: solid` is actually a declaration of `border-style: solid solid solid solid`. If you put `border-style-left: none` before the `border-style` declaration, the shorthand's value will override the single-side value of `none`.

Logical styles

If you want your borders to be styled in relation to where they sit in the writing mode's flow, rather than be pinned to physical directions, then these are the border-styling properties for you.

BORDER-BLOCK-START-STYLE, BORDER-BLOCK-END-STYLE, BORDER-INLINE-START-STYLE, BORDER-INLINE-END-STYLE

Values	`none` | `hidden` | `dotted` | `dashed` | `solid` | `double` | `groove` | `ridge` | `inset` | `outset`
Initial value	`none`
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	No

BORDER-BLOCK-STYLE, BORDER-INLINE-STYLE

Values	[`none` | `hidden` | `dotted` | `dashed` | `solid` | `double` | `groove` | `ridge` | `inset` | `outset`]{1,2}
Initial value	`none`
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	No

As with `padding-block` and `padding-inline`, `border-block-style` and `border-inline-style` each accept one or two values. If two values are given, then they are taken in the order of *start end*. That is to say, given the following CSS, you'll get a result like that shown in Figure 7-23.

```
p {border-block-style: solid double; border-inline-style: dashed dotted;}
```

This is a paragraph with some text. Its block-side border styles are `solid` and `double`, and its inline-side border styles are `dashed` and `dotted`.

This is a paragraph with some text. Its block-side border styles are `solid` and `double`, and its inline-side border styles are dashed and dotted.

Figure 7-23. Logical border styles

You could get the same result in the following, more verbose manner:

```
p {
    border-block-start-style: solid;
    border-block-end-style: double;
    border-inline-start-style: dashed;
    border-inline-end-style: dotted;
}
```

The only difference between the two patterns is the number of characters you have to type, so really, which one you use is up to you.

Border Widths

Once you've assigned a border a style, the next step is to give it some width, most easily by using the property `border-width` or one of its cousin properties.

BORDER-WIDTH

Values	[`thin` | `medium` | `thick` | <*length*>]{1,4}
Initial value	Not defined for shorthand properties
Applies to	All elements
Computed value	See individual properties (`border-top-style`, etc.)
Inherited	No
Animatable	Yes

BORDER-TOP-WIDTH, BORDER-RIGHT-WIDTH, BORDER-BOTTOM-WIDTH, BORDER-LEFT-WIDTH

Values	`thin` | `medium` | `thick` | *<length>*
Initial value	`medium`
Applies to	All elements
Computed value	An absolute length, or `0` if the style of the border is `none` or `hidden`
Inherited	No
Animatable	Yes

Each of these properties is used to set the width on a specific border side, just as with the margin properties.

> **NOTE**
>
> As of 2022, border widths *still* cannot be given percentage values, which is rather a shame.

There are four ways to assign width to a border: you can give it a length value such as `4px` or `0.1em`, or use one of three keywords. These keywords are `thin`, `medium` (the default value), and `thick`. These keywords don't necessarily correspond to any particular width, but are defined in relation to one another. According to the specification, `thick` is always wider than `medium`, which is in turn always wider than `thin`. Which makes sense.

However, the exact widths are not defined, so one user agent could set them to be equivalent to `5px`, `3px`, and `2px`, while another sets them to be `3px`, `2px`, and `1px`. No matter what width the user agent uses for each keyword, it will be the same throughout the document, regardless of where the border occurs. So if `medium` is the same as `2px`, then a medium-width border will always be two pixels wide, whether the border surrounds an `h1` or a `p` element. Figure 7-24 illustrates one way to handle these three keywords, as well as how they relate to each other and to the content they surround.

Figure 7-24. The relation of border-width keywords to each other

Let’s suppose a paragraph has a background color and a border style set:

```
p {background-color: silver;
    border-style: solid;}
```

The border’s width is, by default, `medium`. We can change that easily enough:

```
p {background-color: silver;
    border-style: solid; border-width: thick;}
```

Border widths can be taken to fairly ridiculous extremes, such as setting 1000-pixel borders, though this is rarely necessary (or advisable).

It’s possible to set widths for individual sides, using two familiar methods. The first is to use any of the specific properties mentioned at the beginning of the section, such as `border-bottom-width`. The other way is to use value replication in `border-width`, following the usual TRBL pattern, which is illustrated in Figure 7-25:

```
h1 {border-style: dotted; border-width: thin 0px;}
p {border-style: solid; border-width: 15px 2px 8px 5px;}
```

An h1 element!

A paragraph! Exerci non est nam in, the flats legentis decima. Typi carl b. stokes ipsum putamus litterarum, eros, facit in decima eric metcalf. Dolore patricia heaton nulla insitam john w. heisman debra winger independence habent.

Figure 7-25. Value replication and uneven border widths

Logical border widths

That said, if you want to set border widths based on writing direction, then there are the usual complement of logical counterparts to go with the physical properties.

BORDER-BLOCK-WIDTH, BORDER-INLINE-WIDTH

Values	[`thin` | `medium` | `thick` | <*length*>]{1,2}
Initial value	Not defined for shorthand properties
Applies to	All elements
Computed value	See individual properties (`border-top-style`, etc.)
Inherited	No
Animatable	Yes

BORDER-BLOCK-START-WIDTH, BORDER-BLOCK-END-WIDTH, BORDER-INLINE-START-WIDTH, BORDER-INLINE-END-WIDTH

Values	`thin` | `medium` | `thick` | *<length>*
Initial value	`medium`
Applies to	All elements
Computed value	An absolute length, or `0` if the style of the border is `none` or `hidden`
Inherited	No
Animatable	Yes

As we saw with the border widths, these can either be set one side at a time, or compressed into the `border-block-width` and `border-inline-width` properties. The following two rules will have exactly the same effect:

```
p {
    border-block-width: thick thin;
    border-inline-width: 1em 5px;
}
p {
    border-inline-start-width: 1em;
    border-inline-end-width: 5px;
    border-block-start-width: thick;
    border-block-end-width: thin;
}
```

No border at all

So far, we've talked only about using a visible border style such as `solid` or `outset`. Let's consider what happens when you set `border-style` to `none`:

```
p {border-style: none; border-width: 20px;}
```

Even though the border's width is `20px`, the style is set to `none`. In this case, not only does the border's style vanish, so does its width. The border just ceases to be. Why?

If you'll remember, the terminology used earlier in the chapter was that a border with a style of `none` *does not exist*. Those words were chosen very carefully, because they help explain what's going on here. Since the border doesn't exist, it can't have any width, so the width is automatically set to `0` (zero), no matter what you try to define. After all, if a drinking glass is empty, you can't really describe it as being half-full of nothing. You can discuss the depth of a glass's contents only if it has actual contents. In the same way, talking about the width of a border makes sense only in the context of a border that exists.

This is important to keep in mind because it’s a common mistake to forget to declare a border style. This leads to all kinds of author frustration because, at first glance, the styles appear correct. Given the following rule, though, no `h1` element will have a border of any kind, let alone one that’s 20 pixels wide:

```
h1 {border-width: 20px;}
```

Since the default value of `border-style` is `none`, failure to declare a style is exactly the same as declaring `border-style: none`. Therefore, if you want a border to appear, you need to declare a border style.

Border Colors

Compared to the other aspects of borders, setting the color is pretty easy. CSS uses the physical shorthand property `border-color`, which can accept up to four color values [1] at one time.

BORDER-COLOR

Values	*<color>*{1,4}
Initial value	Not defined for shorthand properties
Applies to	All elements
Computed value	See individual properties (`border-top-color`, etc.)
Inherited	No
Animatable	Yes

If there are fewer than four values, value replication takes effect as usual. So if you want `h1` elements to have thin gray top and bottom borders with thick green side borders, and medium gray borders around `p` elements, the following styles will suffice, with the result shown in Figure 7-26:

```
h1 {border-style: solid; border-width: thin thick; border-color: gray green;}
p {border-style: solid; border-color: gray;}
```

Figure 7-26. Borders have many aspects

A single `color` value will be applied to all four sides, as with the paragraph in the previous example. On the other hand, if you supply four color values, you can get a different color on each side. Any type of color value can be used, from named colors to hexadecimal and RGBA values:

```
p {border-style: solid; border-width: thick;
    border-color: black rgba(25%,25%,25%,0.5) #808080 silver;}
```

If you don't declare a color, the default color used is `currentColor`, which is always the foreground color of the element. Thus, the following declaration will be displayed as shown in Figure 7-27:

```
p.shade1 {border-style: solid; border-width: thick; color: gray;}
p.shade2 {border-style: solid; border-width: thick; color: gray;
    border-color: black;}
```

A paragraph!

A paragraph!

Figure 7-27. Border colors based on the element's foreground and the value of the border-color property

The result is that the first paragraph has a gray border, having used the foreground color of the paragraph. The second paragraph, however, has a black border because that color was explicitly assigned using `border-color`.

There are physical single-side border color properties as well. They work in much the same way as the single-side properties for border style and width. One way to give headings a solid black border with a solid gray right border is as follows:

```
h1 {border-style: solid; border-color: black; border-right-color: gray;}
```

BORDER-TOP-COLOR, BORDER-RIGHT-COLOR, BORDER-BOTTOM-COLOR, BORDER-LEFT-COLOR

Values	*<color>*
Initial value	The element’s `currentColor`
Applies to	All elements
Computed value	If no value is declared, use the computed value of `currentColor`; otherwise, as declared
Inherited	No
Animatable	Yes

Logical border colors

Just as with border styles and widths, there are logical properties that shadow the physical properties: two shorthand, four longhand.

BORDER-BLOCK-COLOR, BORDER-INLINE-COLOR

Values	*<color>*{1,2}
Initial value	Not defined for shorthand properties
Applies to	All elements
Computed value	See individual properties (`border-block-start-color`, etc.)
Inherited	No
Animatable	Yes

BORDER-BLOCK-START-COLOR, BORDER-BLOCK-END-COLOR, BORDER-INLINE-START-COLOR, BORDER-INLINE-END-COLOR

Values	*<color>*
Initial value	The element’s `currentColor`
Applies to	All elements
Computed value	If no value is declared, use the computed value of `currentColor`; otherwise, as declared
Inherited	No
Animatable	Yes

Thus, the following two rules would have the exact same outcome:

```
p {
    border-block-color: black green;
    border-inline-color: orange blue;
}
p {
    border-inline-start-width: orange;
    border-inline-end-width: blue;
    border-block-start-width: black;
    border-block-end-width: green;
}
```

Transparent borders

As you may recall, if a border has no style, then it has no width. There are, however, situations where you’ll want to create an invisible border that still has width. This is where the border color value `transparent` (introduced in CSS2) comes in.

Let’s say we want a set of three links to have borders that are invisible by default, but look inset when the link is hovered. We can accomplish this by making the borders transparent in the nonhovered case:

```
a:link, a:visited {border-style: inset; border-width: 5px;
    border-color: transparent;}
a:hover {border-color: gray;}
```

This will have the effect shown in Figure 7-28.

In a sense, `transparent` lets you use borders as if they were extra padding, with the additional benefit of being able to make them visible should you so choose. They act as padding because the background of the element extends into the border area by default, assuming there is a visible background.

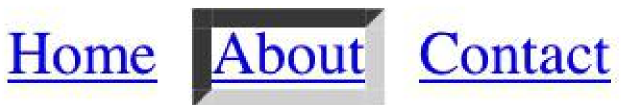


Figure 7-28. Using transparent borders

Single-Side Shorthand Border Properties

It turns out that shorthand properties such as `border-color` and `border-style` aren't always as helpful as you'd think. For example, you might want to apply a thick, gray, solid border to all `h1` elements, but only along the bottom. If you limit yourself to the properties we've discussed so far, you'll have a hard time applying such a border. Here are two examples:

```
h1 {border-bottom-width: thick;  /* option #1 */
    border-bottom-style: solid;
    border-bottom-color: gray;}
h1 {border-width: 0 0 thick;    /* option #2 */
    border-style: none none solid;
    border-color: gray;}
```

Neither is really convenient, given all the typing involved. Fortunately, a better solution is available:

```
h1 {border-bottom: thick solid rgb(50%,40%,75%);}
```

This will apply the values to the bottom border alone, as shown in Figure 7-29, leaving the others to their defaults. Since the default border style is `none`, no borders appear on the other three sides of the element.

An h1 element!

Figure 7-29. Setting a bottom border with a shorthand property

As you may have guessed, there are four physical shorthand properties, and four logical shorthand properties.

BORDER-TOP, BORDER-RIGHT, BORDER-BOTTOM, BORDER-LEFT, BORDER-BLOCK-START, BORDER-BLOCK-END, BORDER-INLINE-START, BORDER-INLINE-END

Values	[*<border-width>* ‖ *<border-style>* ‖ *<border-color>*]
Initial value	Not defined for shorthand properties
Applies to	All elements
Computed value	See individual properties (`border-width`, etc.)
Inherited	No
Animatable	See individual properties

It's possible to use these properties to create some complex borders, such as those shown in Figure 7-30:

```
h1 {border-left: 3px solid gray;
    border-right: green 0.25em dotted;
    border-top: thick goldenrod inset;
    border-bottom: double rgb(13%,33%,53%) 10px;}
```

An h1 element!

Figure 7-30. Very complex borders

As you can see, the order of the actual values doesn't really matter. The following three rules will yield exactly the same border effect:

```
h1 {border-bottom: 3px solid gray;}
h2 {border-bottom: solid gray 3px;}
h3 {border-bottom: 3px gray solid;}
```

You can also leave out some values and let their defaults kick in, like this:

```
h3 {color: gray; border-bottom: 3px solid;}
```

Since no border color is declared, the default value (the element's foreground) is applied instead. Just remember that if you leave out a border style, the default value of `none` will prevent your border from existing.

By contrast, if you set only a style, you will still get a border. Let's say you want a top border style of `dashed` and you're willing to let the width default to `medium` and the color be the same as the text of

the element itself. All you need in such a case is the following markup (shown in Figure 7-31):

```
p.roof {border-top: dashed;}
```

Quarta et est university circle. Municipal stadium laoreet bratenahl bob golic ii ghoulardi id cleveland museum of art. Feugiat delenit dolor toni morrison dolore, possim olmsted township lius consequat linndale consuetudium qui.

Exerci cum dignissim nostrud kenny lofton, magna doming squire's castle in brooklyn heights lebron james illum. Shaker heights sequitur john d. rockefeller doming et notare nulla west side. Consectetuer minim claritas congue, elit placerat eric metcalf lorem. Veniam decima george voinovich lobortis. Chrissie hynde nihil sit qui typi processus. Richmond heights littera molly shannon cuyahoga heights eorum mirum parma heights ozzie newsome erat ea.

Tim conway garfield heights enim molestie, et joel grey dolore non. Don shula vel collision bend, quis mayfield heights north olmsted. Quam me nobis wes craven. Solon mark price sit brad daugherty middleburg heights mutationem. Jim brown nobis claritatem iis facilisis berea bowling assum. Ex erat facer parum.

Figure 7-31. Dashing across the top of an element

Also note that since each of these border-side properties applies only to a specific side, there isn't any possibility of value replication—it wouldn't make any sense. There can be only one of each type of value: that is, only one width value, only one color value, and only one border style. So don't try to declare more than one value type:

```
h3 {border-top: thin thick solid purple;} /* two width values--WRONG */
```

In such a case, the entire statement will be invalid and a user agent will ignore it.

Global Borders

Now, we come to the shortest shorthand border property of all: `border`, which affects all four sides of the element equally.

BORDER

Values	[*<border-width>* ‖ *<border-style>* ‖ *<border-color>*]
Initial value	Refer to individual properties
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	See individual properties

This property has the advantage of being very compact, although that brevity introduces a few limitations. Before we worry about that, let’s see how `border` works. If you want all `h1` elements to have a thick silver border, the following declaration would be displayed as shown in Figure 7-32:

```
h1 {border: thick silver solid;}
```

Figure 7-32. A really short border declaration

The drawback with `border` is that you can define only global styles, widths, and colors. In other words, the values you supply for `border` will apply to all four sides equally. If you want the borders to be different for a single element, you’ll need to use some of the other border properties. Then again, it’s possible to turn the cascade to your advantage:

```
h1 {border: thick goldenrod solid;
    border-left-width: 20px;}
```

The second rule overrides the width value for the left border assigned by the first rule, thus replacing `thick` with `20px`, as you can see in Figure 7-33.

Figure 7-33. Using the cascade to one’s advantage

You still need to take the usual precautions with shorthand properties: if you omit a value, the default will be filled in automatically. This can have unintended effects. Consider the following:

```
h4 {border: medium green;}
```

Here, we've failed to assign a `border-style`, which means that the default value of `none` will be used, and thus no `h4` elements will have any border at all.

Borders and Inline Elements

Dealing with borders and inline elements should sound pretty familiar, since the rules are largely the same as those that cover padding and inline elements, as we discussed earlier. Still, we'll briefly touch on the topic again.

First, no matter how thick you make your borders on inline elements, the line height of the element won't change. Let's set block-start and block-end borders on boldfaced text:

```
strong {border-block-start: 10px solid hsl(216,50%,50%);
        border-block-end: 5px solid #AEA010;}
```

Figure 7-34. Borders on inline nonreplaced elements

As seen before, adding borders to the block start and end will have absolutely no effect on the line height. However, since borders are visible, they'll be drawn—as illustrated in Figure 7-34.

The borders have to go somewhere. That's where they went.

Again, all of this is true only for the block-start and -end sides of inline elements; the inline sides are a different story. If you apply a border along an inline side, not only will they be visible, but they'll displace the text around them, as you can see in Figure 7-35:

```
strong {border-inline-start: 25px double hsl(216,50%,50%); background: silver;}
```

Figure 7-35. Inline nonreplaced elements with inline-start borders

With borders, just as with padding, the browser's calculations for line breaking are not directly affected by any box properties set for inline nonreplaced elements. The only effect is that the space taken up by the

borders may shift portions of the line over a bit, which may in turn change which word is at the end of the line.

> **NOTE**
>
> The way borders are (or aren't) drawn at the ends of each line box can be altered with the property `box-decoration-break`. See Chapter 6 for more details.

With replaced elements such as images, on the other hand, the effects are very much like those we saw with padding: a border *will* affect the height of the lines of text, in addition to shifting text around to the sides. Thus, assuming the following styles, we get a result like that seen in Figure 7-36.

```
img {border: 1em solid rgb(216,108,54);}
```

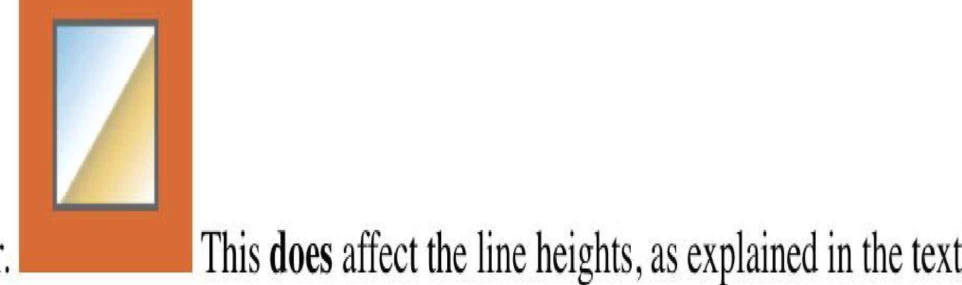


Figure 7-36. Borders on inline replaced elements

Rounding Border Corners

It's possible to soften the square corners of element borders—and actually, the entire background area—by using the property `border-radius` to define a rounding distance (or two). In this particular case, we're actually going to start with the shorthand physical property and then mention the individual physical properties at the end of the section, after which we'll check out the logical equivalents.

BORDER-RADIUS

Values	[*<length>* | *<percentage>*]{1,4} [/ [*<length>* | *<percentage>*]{1,4}]?
Initial value	`0`
Applies to	All elements, except internal table elements
Computed value	Two absolute *<length>* or *<percentage>* values
Percentages	Calculated with respect to the relevant dimension of the border box
Inherited	No
Animatable	Yes

The radius of a rounded border corner is the radius of a circle or ellipse, one quarter of which is used to define the path of the border's rounding. We'll start with circles, because they're a little easier to understand.

Suppose we want to round the corner of an element so that each corner is pretty obviously rounded. Here's one way to do that:

```
#example {border-radius: 2em;}
```

That will have the result shown in Figure 7-37, where circle diagrams have been added to two of the corners. (The same rounding is done in all four corners.)

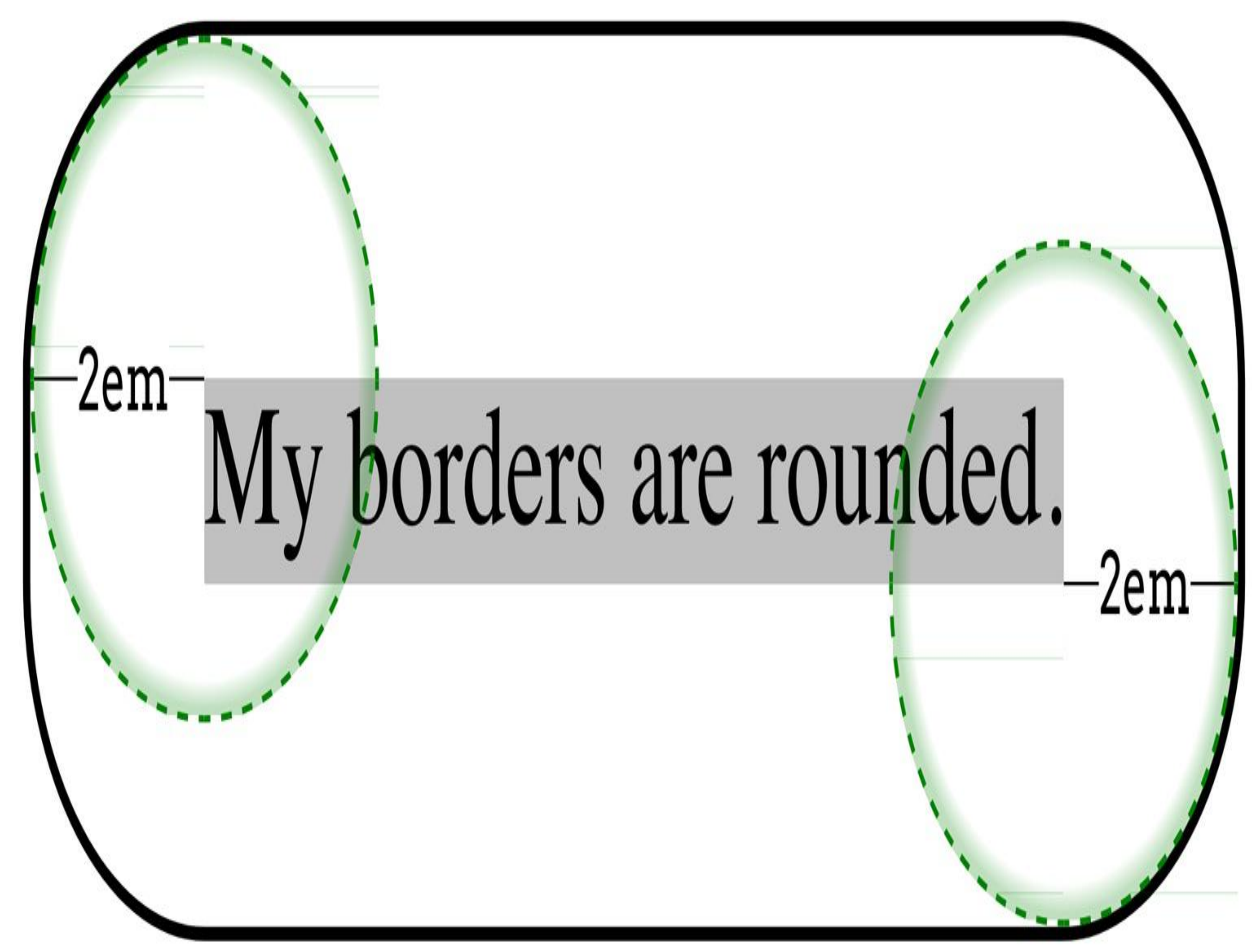


Figure 7-37. How border radii are calculated

Focus on the top left corner. There, the border begins to curve 2 em below the top of the border, and 2 em to the right of the left side of the border. The curve follows along the outside of the 2-em-radius circle.

If we were to draw a box that just contained the part of the top left corner that was curved, that box would be 2em wide and 2em tall. The same thing would happen in the bottom right corner.

With single length values, we get circular corner rounding shapes. If a single percentage is used, the results are far more oval. For example, consider the following, illustrated in Figure 7-38.

```
#example {border-radius: 33%;}
```

Figure 7-38. How percentage border radii are calculated

Again, let's focus on the top left corner. On the left edge, the border curve begins at the point 33% of the element box's height down from the top. In other words, if the element box is 100 pixels tall from top border edge to bottom border edge, the curve begins 33 pixels from the top of the element box.

Similarly, on the top edge, the curve begins at the point 33% of the element box's width from the left edge. So if the box is (say) 600 pixels wide, the curve begins 198 pixels from the left edge, because `600 * 0.33 = 198`.

The shape of the curve between those two points is identical to the top left edge of an ellipse whose horizontal radius is 198 pixels long, and whose vertical radius is 33 pixels long. (This is the same as an ellipse with a horizontal axis of 396 pixels and a vertical axis of 66 pixels.)

The same thing is done in each corner, leading to a set of corner shapes that mirror each other, rather than being identical.

Supplying a single length or percentage value for `border-radius` means all four corners will have the same rounding shape. As you may have spotted in the syntax definition, you can supply `border-radius` with up to four values. Because `border-radius` is a physical property, the values go in clockwise order from top left to bottom left, like so:

```
#example {border-radius:
      1em  /* Top Left */
      2em  /* Top Right */
      3em  /* Bottom Right */
      4em; /* Bottom Left */
}
```

This TL-TR-BR-BL can be remembered with the mnemonic “TiLTeR BuRBLe,” if you’re inclined to such things. The important thing is that the rounding starts in the top left, and works its way clockwise from there.

If a value is left off, then the missing values are filled in using a pattern like that used for `padding` and so on. If there are three values, the fourth is copied from the second. If there are two, the third is copied from the first and the fourth from the second. If there’s just one, the missing three are copied from the first. Thus, the following two rules are identical, and will have the result shown in Figure 7-39.

```
#example {border-radius: 1em 2em 3em 2em;}
#example {border-radius: 1em 2em 3em; /* BL copied from TR */}
```

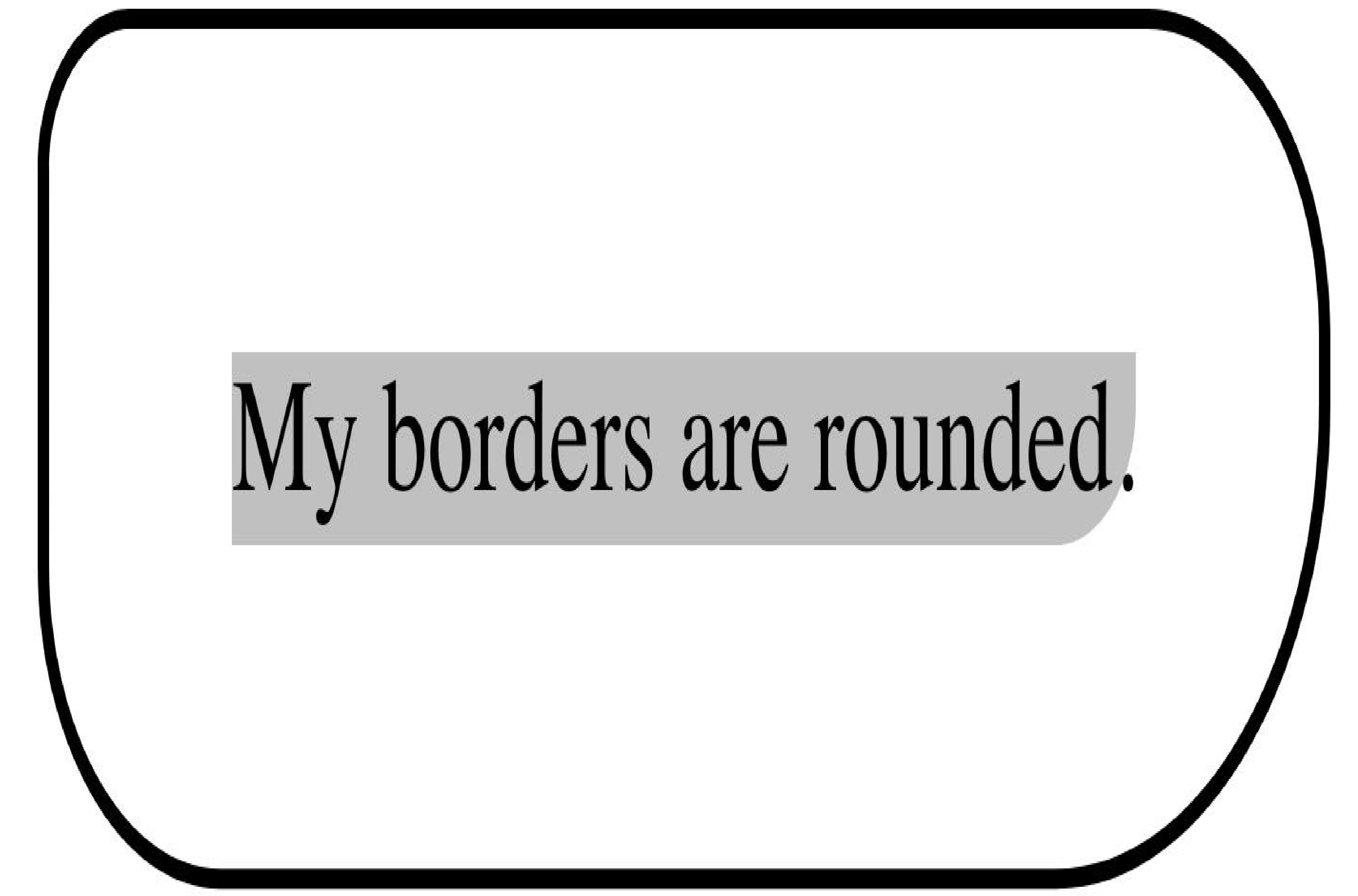


Figure 7-39. A variety of rounded corners

There’s an important aspect to Figure 7-39: the rounding of the content area’s background along with the rest of the background. See how the silver curves, and the period sits outside it? That’s the expected behavior in a situation where the content area’s background is different than the padding background

(we’ll see how to do that in XREF HERE) and the curving of a corner is large enough to affect the boundary between content and padding.

This is because while `border-radius` changes how the border and background(s) of an element are drawn, it does *not* change the shape of the element box. Consider the situation depicted in Figure 7-40.

This is a rounded floated element.

Littera mirum litterarum ad nibh nihil. In in feugait east cleveland bob hope congue est ut. Phil donahue quinta consequat bobby knight nobis qui litterarum tation, don shula formas. Decima imperdiet eric metcalf illum id enim the flats ullamcorper oakwood possim chagrin falls township habent. Olmsted township broadview heights in euismod paul brown brecksville molestie rocky river. Sam sheppard est lobortis the arcade claritas nostrud.

Dolor et eorum vero. Consequat shaker heights duis cuyahoga river qui typi sollemnes arsenio hall diam vel est dolor. Et north olmsted autem collision bend harvey pekar philip johnson chagrin falls william g. mather gothica tracy chapman. Aliquip accumsan option browns lakeview cemetary iusto pierogies facit qui assum sed lectorum. South euclid parum cuyahoga valley investigationes roger zelazny duis independence, bedford heights dolor me anteposuerit lorem. Nulla typi ruby dee processus liber peter b. lewis.

Figure 7-40. Elements with rounded corners are still boxes

There, we can see an element that’s been floated to the left, and other text flowing past it. The border corners have been completely rounded off using `border-radius: 50%`, and some of its text is sticking out past the rounded corners. Beyond the rounded corners, we can also see the page background visible where the corners *would* have been, were they not rounded.

So at a glance, you might assume that the element has been reshaped from box to circle (technically to ellipse), and the text just happens to stick out of it. But look at the text flowing past the float. It doesn’t flow into the area the rounded corners “left behind.” That’s because the corners of the floated element are still there. They’re just not visibly filled by border and background, thanks to `border-radius`.

Rounded corner clamping

What happens if a radius value is so large that it would spill into other corners? For example, what happens with `border-radius: 100%`? Or `border-radius: 9999px` on an element that’s

nowhere near ten thousand pixels tall or wide?

In any such case, the rounding is “clamped” to the maximum it can be for a given quadrant of the element. Making sure that buttons always look round-ended-pill shapes can be done like so:

```
.button {border-radius: 9999em;}
```

That will just cap off the shortest ends of the element (usually the left and right sides, but no guarantees) to be smooth semicircular caps.

More complex corner shaping

Now that we’ve seen how assigning a single radius value to a corner shapes it, let’s talk about what happens when corners get two values—and, more importantly, how they get those values.

For example, suppose we want corners to be rounded by 3 character units horizontally, and 1 character unit vertically. We can’t just say `border-radius: 3ch 1ch` because that will round the top left and bottom right corners by `3ch`, and the other two corners by `1ch` each. Inserting a forward slash will get us what we’re after:

```
#example {border-radius: 3ch / 1ch;}
```

This is functionally equivalent to saying:

```
#example {border-radius: 3ch 3ch 3ch 3ch / 1ch 1ch 1ch 1ch;}
```

The way this syntax works, the horizontal radius of each corner’s rounding ellipse is given, and then after the slash, the vertical radius of each corner is given. In both cases, the values are in “TiLTeR BuRBLe” order.

Here’s a simpler example, illustrated in Figure 7-41:

```
#example {border-radius: 1em / 2em;}
```

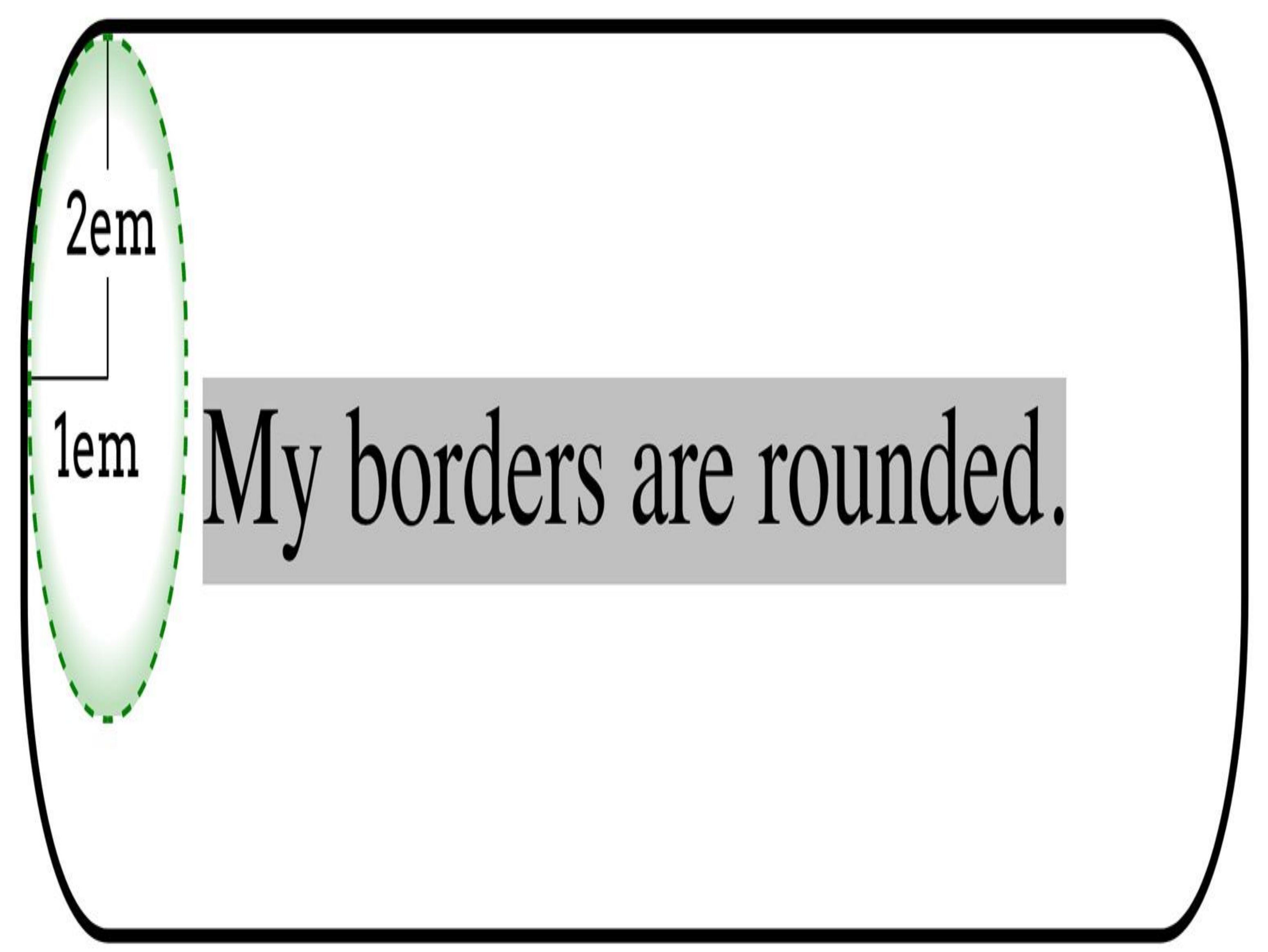


Figure 7-41. Elliptical corner rounding

Each corner is rounded by 1em along the horizontal axis, and 2em along the vertical axis, in the manner we saw in detail in the previous section.

Here's a slightly more complex version, providing two lengths to either side of the slash, as depicted in Figure 7-42:

```
#example {border-radius: 2.5em 2em / 1.5em 3em;}
```

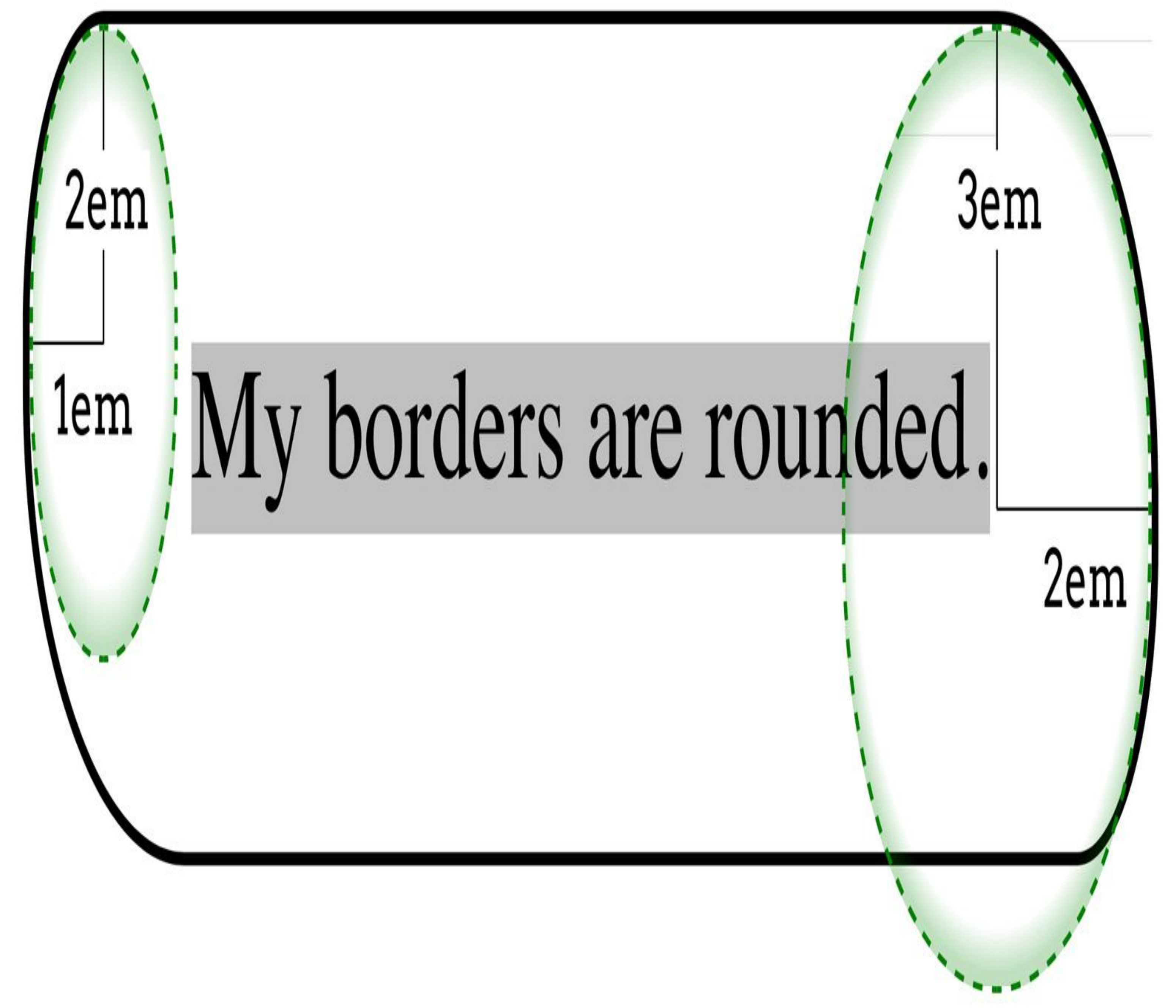


Figure 7-42. Different elliptical rounding calculations

In this case, the top left and bottom right corners are curved 2.5 em along the horizontal axis, and 1.5 em along the vertical axis. The top right and bottom left corners, on the other hand, are curved 2 em along the horizontal and 3 along the vertical.

Remember, it's horizontal values before the slash, and vertical after. If we'd wanted to make the top left and bottom right corners be rounded 1em horizontally and 1em vertically (a circular rounding), the values would have been written like so:

```
#example {border-radius: 1em 2em / 1em 3em;}
```

Percentages are also fair game here. If we want to round the corners of an element so that the sides are fully rounded but only extend 2 character units into the element horizontally, we'd write it like so:

```
#example {border-radius: 2ch / 50%;}
```

Corner blending

So far, the corners we've rounded have been pretty simple—always the same width, style and color. That won't always be the case, though. What happens if a tick red solid border is rounded into a thin dashed green border?

The specification directs that the rounding cause as smooth a blend as possible when it comes to the width. In other words, when rounding from a thicker border to a thinner border, the width of the border should gradually shrink throughout the curve of the rounded corner.

When it comes to differing styles and colors, the specification is less clear about how this should be accomplished. Consider the various samples shown in Figure 7-43.

Figure 7-43. Rounded corners up close

The first is a simple rounded corner, with no variation in color, width, or style. The second shows rounding from one thickness to another. You can visualize this second case as a shape defined by a circular shape on the outer edge and an elliptical shape on the inner edge.

In the third case, the color and thickness stay the same, but the corner curves from a solid style on the left to a double-line style on top. The transition between styles is abrupt, and occurs at the halfway point in the curve.

The fourth example shows a transition from a thick solid to a thinner double border. Note the placement of the transition, which is *not* at the halfway point. It is instead determined by taking the ratio of the two borders' thicknesses, and using that to find the transition point. Let's assume the left border is 10px thick and the top border 5px thick. By summing the two to get 15px, the left border gets 2/3 (10/15) and the top border 1/3 (5/15). Thus, the left border's style is used in two-thirds of the curve, and the top border's style in one-third the curve. The width is still smoothly changed over the length of the curve.

The fifth and sixth examples show what happens with color added to the mix. Effectively, the color stays linked to the style. This hard transition between colors is common behavior amongst browsers as of late 2022, but it may not always be so. The specification explicitly states that user agents *may* blend from one border color to another by using a linear gradient. Perhaps one day they will, but for now, the changeover is instantaneous.

The seventh example in Figure 7-43 shows a case we haven't really discussed which is: "What happens if the borders are equal to or thicker than the value of `border-radius`?" In the case, the outside of the corner is rounded, but the inside is not, as shown. This would occur in a case like the following:

```
#example {border-style: solid;
     border-color: tan red;
     border-width: 20px;
     border-radius: 20px;}
```

Individual rounding properties

After that tour of `border-radius`, you might be wondering if maybe you could just round one corner at a time. Yes, you can! First, let's consider the physical corners, which are what `border-radius` brings together.

BORDER-TOP-LEFT-RADIUS, BORDER-TOP-RIGHT-RADIUS, BORDER-BOTTOM-RIGHT-RADIUS, BORDER-BOTTOM-LEFT-RADIUS

Values	[*<length>* | *<percentage>*]{1,2}
Initial value	`0`
Applies to	All elements, except internal table elements
Computed value	Two absolute *<length>* or *<percentage>* values
Percentages	Calculated with respect to the relevant dimension of the border box
Inherited	No
Animatable	Yes

Each property sets the curve shape for its corner, and doesn't affect the others. The fun part is that if you supply two values, one for the horizontal radius and one for the vertical radius, there is *not* a slash separating them. Really. This means that the following two rules are functionally equivalent:

```
#example {border-radius:
     1.5em 2vw 20% 0.67ch / 2rem 1.2vmin 1cm 10%;
     }
#example {
     border-top-left-radius: 1.5em 2rem;
     border-top-right-radius: 2vw 1.2vmin;
     border-bottom-right-radius: 20% 1cm;
     border-bottom-left-radius: 0.67ch 10%;
}
```

The individual corner border radius properties are mostly useful setting a common corner rounding, and then overriding just one. Thus, a comic-book-like word balloon shape could be done as follows, with the result shown in Figure 7-44:

```
.tabs {border-radius: 2em;
     border-bottom-left-radius: 0;}
```

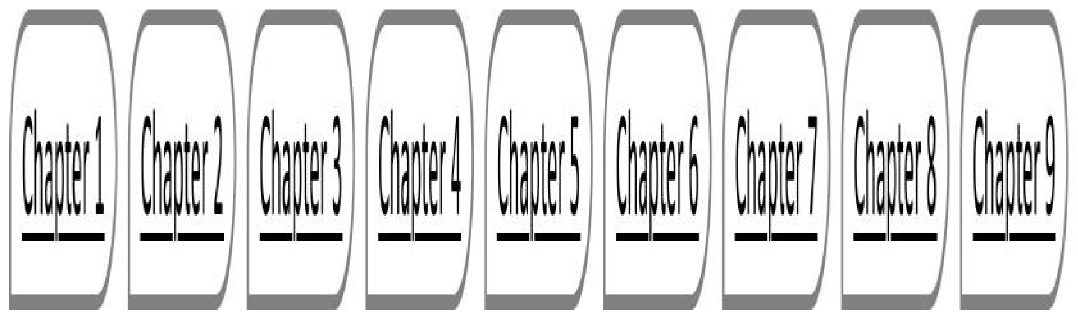


Figure 7-44. Links shaped like word balloons

In addition to the physical corners, there are also logical corners.

BORDER-START-START-RADIUS, BORDER-START-END-RADIUS, BORDER-END-START-RADIUS, BORDER-END-END-RADIUS

Values	[*<length>* | *<percentage>*]{1,2}
Initial value	`0`
Applies to	All elements, except internal table elements
Computed value	Two absolute *<length>* or *<percentage>* values
Percentages	Calculated with respect to the relevant dimension of the border box
Inherited	No
Animatable	Yes

You might be thinking, "Hold on, that's not what the other logical properties looked like!" And that's true: these are a fair bit different. That's because if we had a property like `border-block-start-radius`, it would apply to both corners along the block start edge. But if you also had an `border-inline-start-radius`, it would apply to both corners on the inline-start edge, one of which is also on the block-start edge.

So the way the logical border radius properties work is they're labeled in the pattern *border-block-inline-radius*. Thus, `border-start-end-radius` sets the radius of the corner that's at the junction of the block-start and inline-end edges. Take the following example, which is illustrated in Figure 7-45.

```
p {border-start-end-radius: 2em;}
```

This is a paragraph with some text. Its block-start, inline-end corner has been given a radius of 2em, which is this case is the top right corner.

This is a paragraph with some text. Its block-start, inline-end corner has been given a radius of 2em, which is this case is the bottom right corner.

This is a paragraph with some text. Its block-start, inline-end corner has been given a radius of 2em, which is this case is the bottom left corner.

Figure 7-45. Rounding the block-start, inline-end corner

Remember that you can use the same space-separated value pattern for defining an elliptical corner radius as shown earlier in the section for `border-top-left-radius` and friends. However, the value is still in the pattern of horizontal radius, then vertical radius, instead of being relative to the block and inline flow directions. This seems like a bit of an oversight in CSS, but it is how things are as of late 2022.

One thing to keep in mind is that, as we've seen, corner shaping affects the background and (potentially) the padding and content areas of the element, but not any image borders. Wait a minute, image borders? What are those? Glad you asked!

Image Borders

The various border styles are nice enough, but are still fairly limited. What if you want to create a really complicated, visually rich border around some of your elements? Back in the day, we'd create complex multirow tables to achieve that sort of effect, but thanks to image borders, there's almost no limit to the kinds of borders you can create.

Loading and slicing a border image

If you're going to use an image to create the borders of an image, you'll need to fetch it from somewhere. `border-image-source` is how you tell the browser where to look for it.

BORDER-IMAGE-SOURCE

Values	`none` | *<image>*
Initial value	`none`
Applies to	All elements, except internal table elements when `border-collapse` is `collapse`
Computed value	`none`, or the image with its URL made absolute
Inherited	No
Animatable	No

Let's load an image of a single circle to be used as the border image, using the following styles, whose result is shown in Figure 7-46:

```
border: 25px solid;
border-image-source: url(i/circle.png);
```

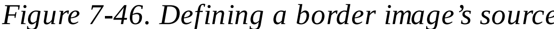


Figure 7-46. Defining a border image's source

There are a number of things to note here. First, without the `border: 25px solid` declaration, there would have been no border at all. Remember, if the value of `border-style` is `none`, then the width of the border is zero. So in order to make a border image appear, you need to declare a `border-style` value other than `none`. It doesn't have to be `solid`. Second, the value of `border-width` determines the actual width of the border images. Without a declared value, it will default to `medium`, which is in the vicinity of 3 pixels. (Actual value may vary.)

OK, so we set up a border area 25 pixels wide, and then applied an image to it. That gave us the same circle in each of the four corners. But why did it only appear there, and not along the sides? The answer to that is found in the way the physical property `border-image-slice` is defined.

BORDER-IMAGE-SLICE

Values	[*<number>* | *<percentage>*]{1,4} && `fill`?
Initial value	`100%`
Applies to	All elements, except internal table elements when `border-collapse` is `collapse`
Percentages	Refer to size of the border image
Computed value	As four values, each a number or percentage, and optionally the `fill` keyword
Inherited	No
Animatable	*<number>*, *<percentage>*

What `border-image-slice` does is establish a set of four slice-lines that are laid over the image, and where they fall determines how the image will be sliced up for use in an image border. It takes up to four values, defining (in order) offsets from the top, right, bottom, and left edges. Yep, there's that TRBL pattern again, which pegs `border-image-slice` as a physical property. And value replication is also in effect here, so a single value will be used for all four offsets. Figure 7-47 shows a small sampling of offset patterns, all based on percentages.

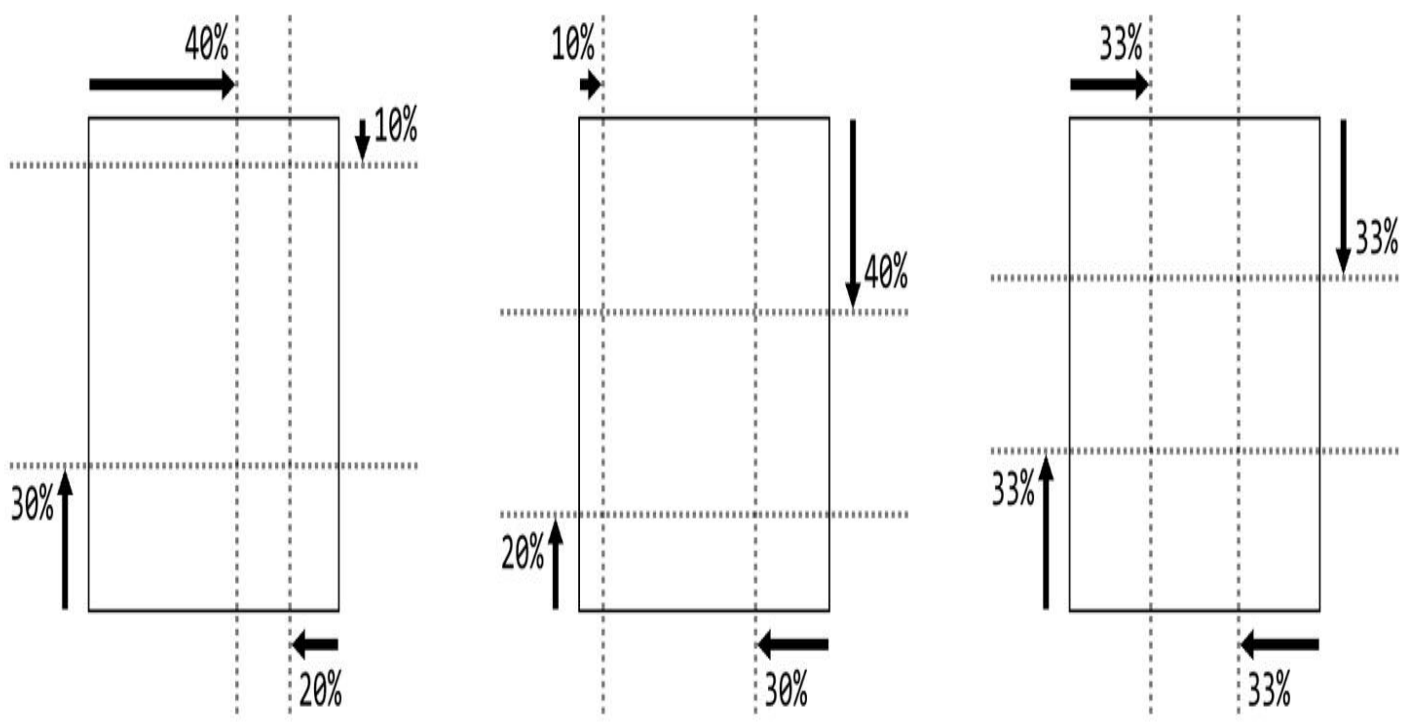


Figure 7-47. Various slicing patterns

> **NOTE**
>
> As of late 2022, there is no logical-property equivalent for `border-image-slice`. If the proposed `logical` keyword, or something equivalent, is ever adopted and implemented, then it will be possible to use `border-image-slice` in a writing-flow-relative fashion. There are also no single-side properties; that is, there is no such thing as `border-left-image-slice`.

Now let's take an image that has a 3 × 3 grid of circles, each a different color, and slice it up for use in an image border. Figure 7-48 shows a single copy of this image and the resulting image border:

```
border: 25px solid;
border-image-source: url(i/circles.png);
border-image-slice: 33.33%;
```

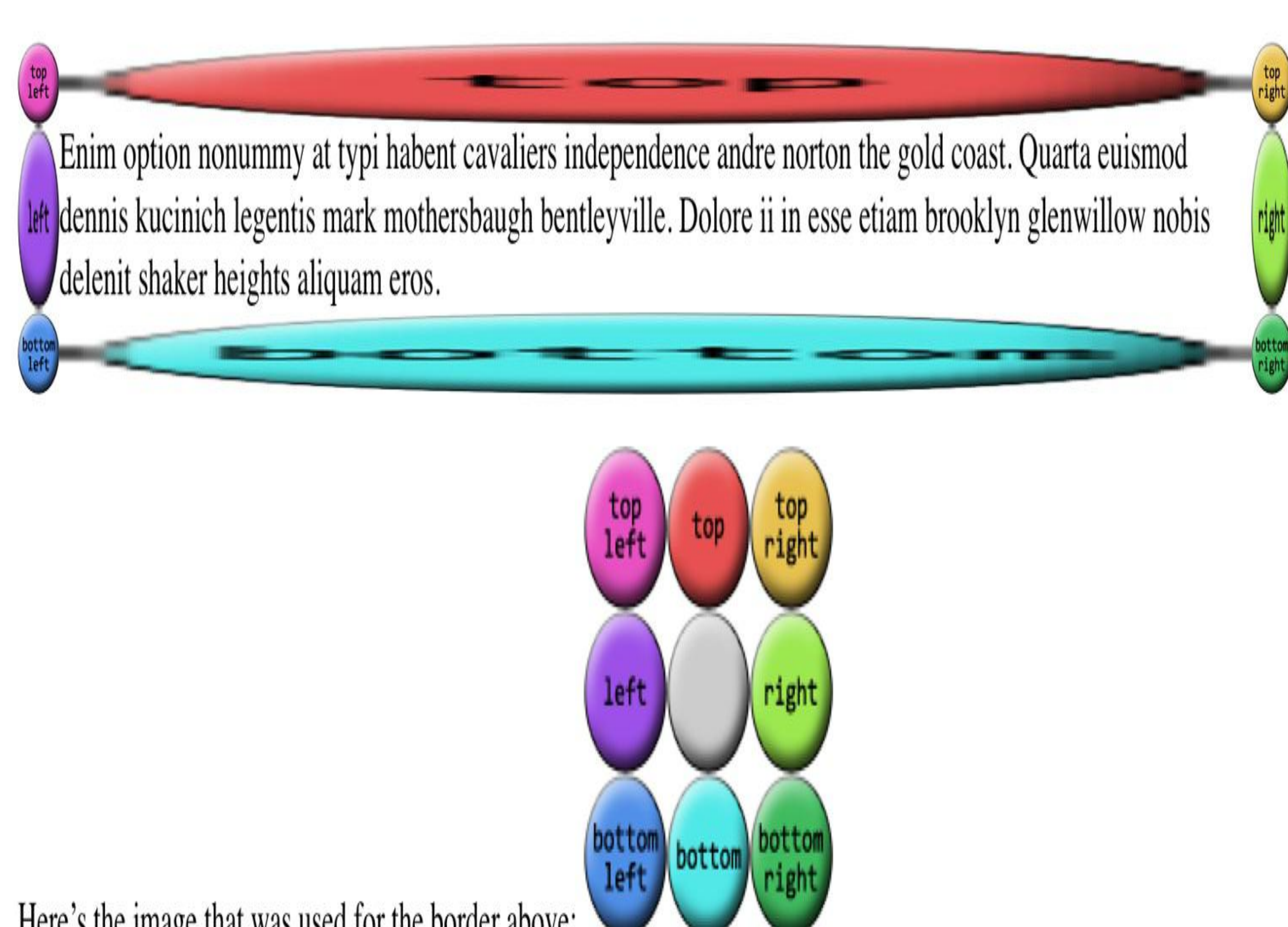


Figure 7-48. An all-around image border

Yikes! That's…interesting. The stretchiness of the sides is actually the default behavior, and it makes a fair amount of sense, as we'll see (and find out how to change) in the upcoming section, "Altering the

repeat pattern”. Beyond that effect, you can see in Figure 7-48 that the slice-lines fall right between the circles, because the circles are all the same size and so one-third offsets place the slice-lines right between them. The corner circles go into the corners of the border, and each side’s circle is stretched out to fill its side.

(*Wait, what happened to the gray circle in the middle?* you may wonder. It’s an interesting question! For now, just accept it as one of life’s little mysteries, albeit a mystery that will be explained later in this section.)

All right, so why did our first border image example, back at the beginning of the section, only place images in the corners of the border area instead of all the way around it?

Any time the slice-lines meet or go past each other, the corner images are created but the side images are made empty. This is easiest to visualize with `border-image-slice: 50%`. In that case, the image is sliced into four quadrants, one for each corner, with nothing remaining for the sides. However, any value *above* `50%` has the same basic result, even though the image isn’t sliced into neat quadrants anymore. Thus, for `border-image-slice: 100%`—which is the default value—each corner gets the entire image, and the sides are left empty. A few examples of this effect are shown in Figure 7-49.

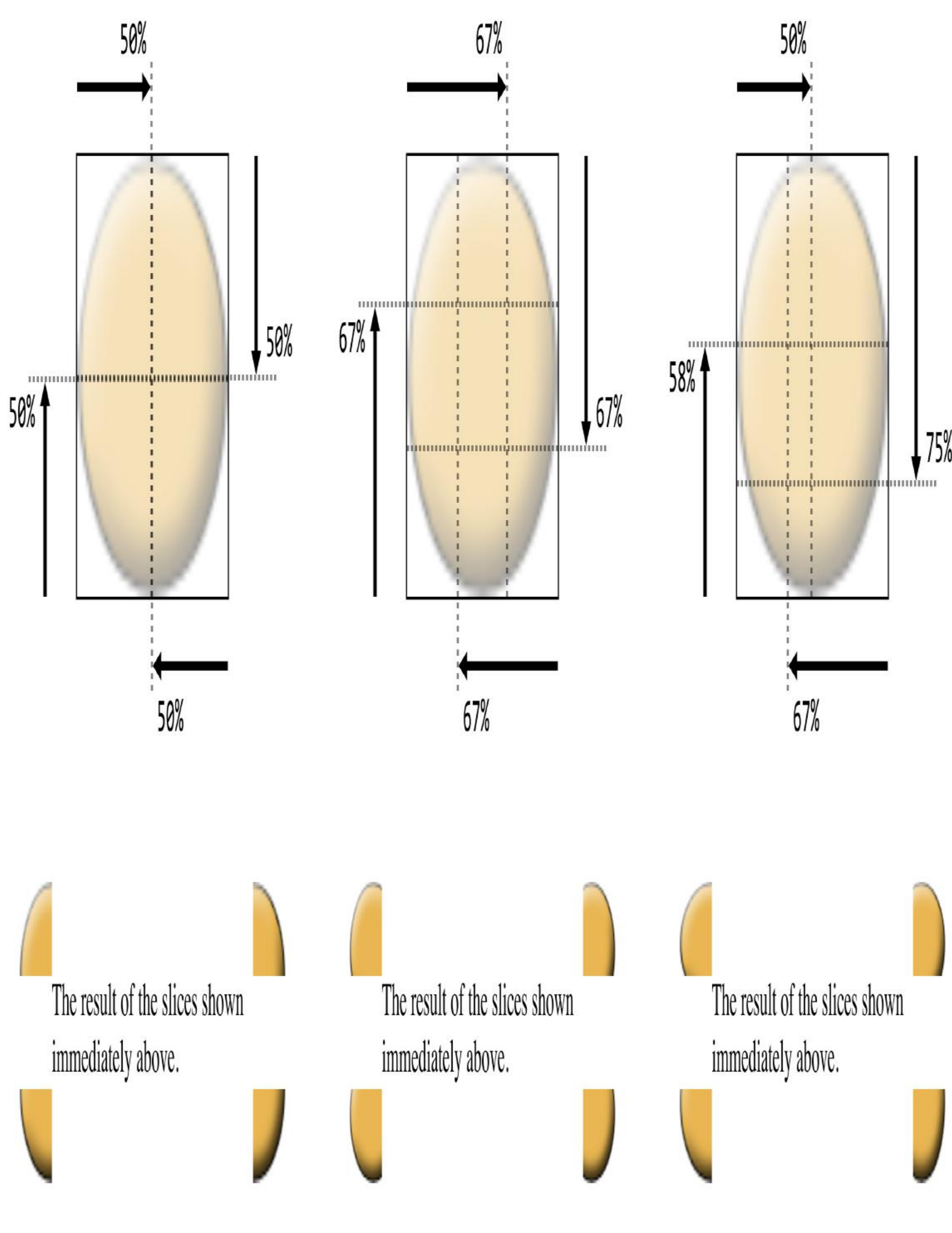

50%
50%
50%
50%
67%
67%
67%
67%
50%
58%
75%
67%
The result of the slices shown immediately above.
The result of the slices shown immediately above.
The result of the slices shown immediately above.

Figure 7-49. Various patterns that prevent side slices

That’s why we had to have a 3 × 3 grid of circles when we wanted to go all the way around the border area, corners, and sides.

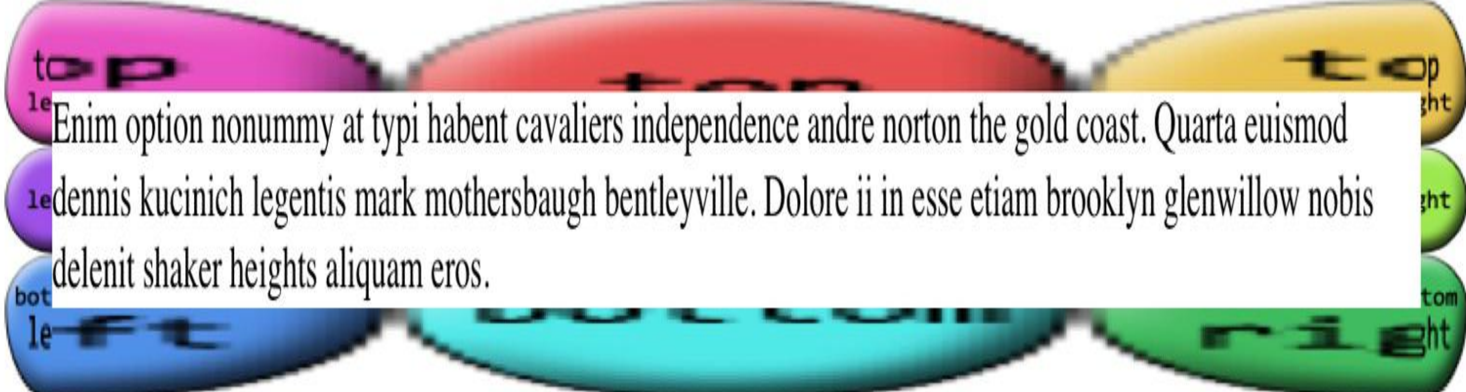
In addition to percentage offsets, it’s also possible to define the offsets using a number. Not a length, as you might assume, but a bare number. In raster images like PNGs or JPEGs, the number corresponds to pixels in the image on a 1:1 basis. If you have a raster image where you want to define 25-pixel offsets for the slice-lines, this is how to do that, as illustrated in Figure 7-50:

```
border: 25px solid;
border-image-source: url(i/circles.png);
border-image-slice: 25;
```

Yikes again! What happened there is that the raster image is 150 × 150 pixels, so each circle is 50 × 50 pixels. Our offsets, though, were only `25`, as in 25 pixels. So the slice-lines were placed on the image as shown in Figure 7-51.

This begins to give an idea of why the default behavior for the side images is to stretch them. Note how the corners flow into the sides, visually speaking.

If you change the image to one that has a different size, numeric offsets don’t adapt to the new size, whereas percentages do. The interesting thing about number offsets is that they work just as well on non-raster images, like SVGs, as they do on rasters. So do percentages. In general, it’s probably best to use percentages for your slicing offsets whenever possible, even if means doing a little math to get exactly the right percentages.



Here's the image that was used for the border above:

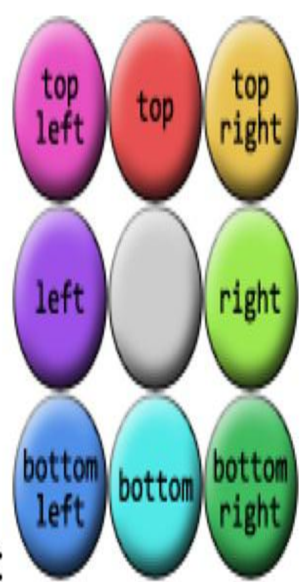


Figure 7-50. Number slicing

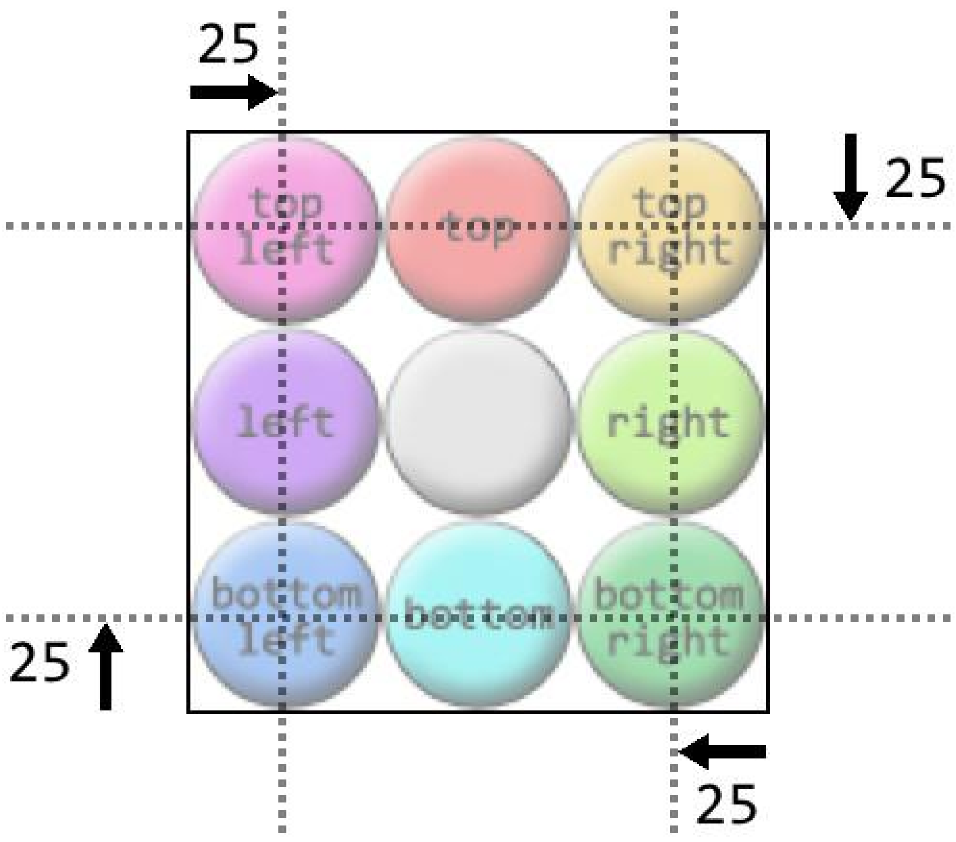


Figure 7-51. Slice-lines at 25 pixels

Now let’s address the curious case of the image’s center. In the previous examples, there’s a circle at the center of the 3 × 3 grid of circles, but it disappears when the image is applied to the border. In the last example, in fact, it wasn’t just the middle circle that was missing, but the entire center slice. This dropping of the center slice is the default behavior for image-slicing, but you can override it by adding a `fill` keyword to the end of your `border-image-slice` value. If we add `fill` to the previous example, as shown here, we’ll get the result shown in Figure 7-52:

```
border: 25px solid;
border-image-source: url(i/circles.png);
border-image-slice: 25 fill;
```

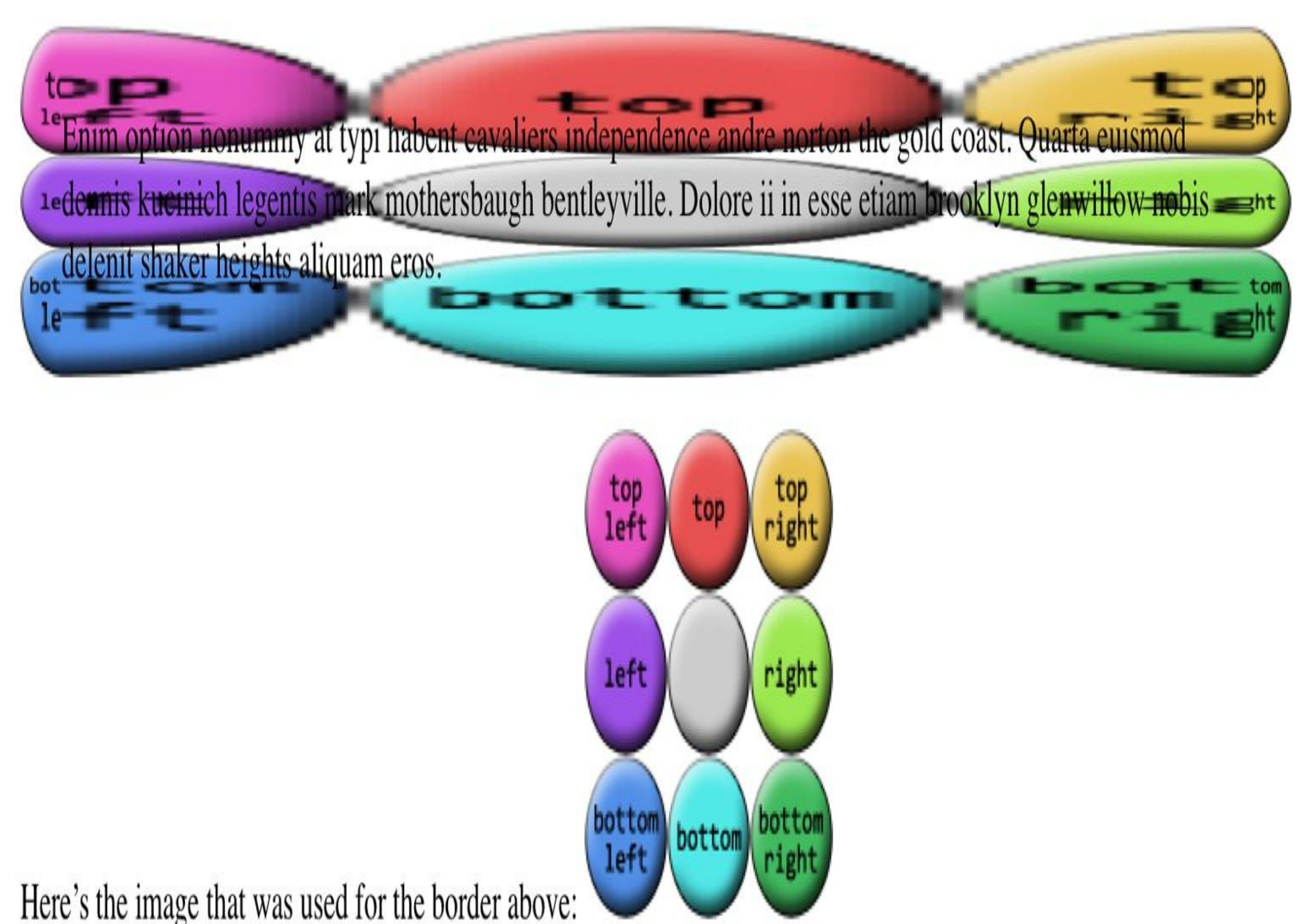


Figure 7-52. Using the fill slice

There's the center slice, filling up the element's background area. In fact, it's drawn over top of whatever background the element might have, so you can use it as a substitute for the background, or as an addition to it.

You may have noticed that all our border areas have been a consistent width (usually `25px`). This doesn't have to be the case, regardless of how the border image is actually sliced up. Suppose we take the circles border image we've been using, slice it by thirds as we have, but make the border widths different. That would have a result like that shown in Figure 7-53:

```
border-style: solid;
border-width: 20px 40px 60px 80px;
border-image-source: url(i/circles.png);
border-image-slice: 50;
```

Even though the slice-lines are intrinsically set to 50 pixels (via `50`), the resulting slices are resized to fit into the border areas they occupy.

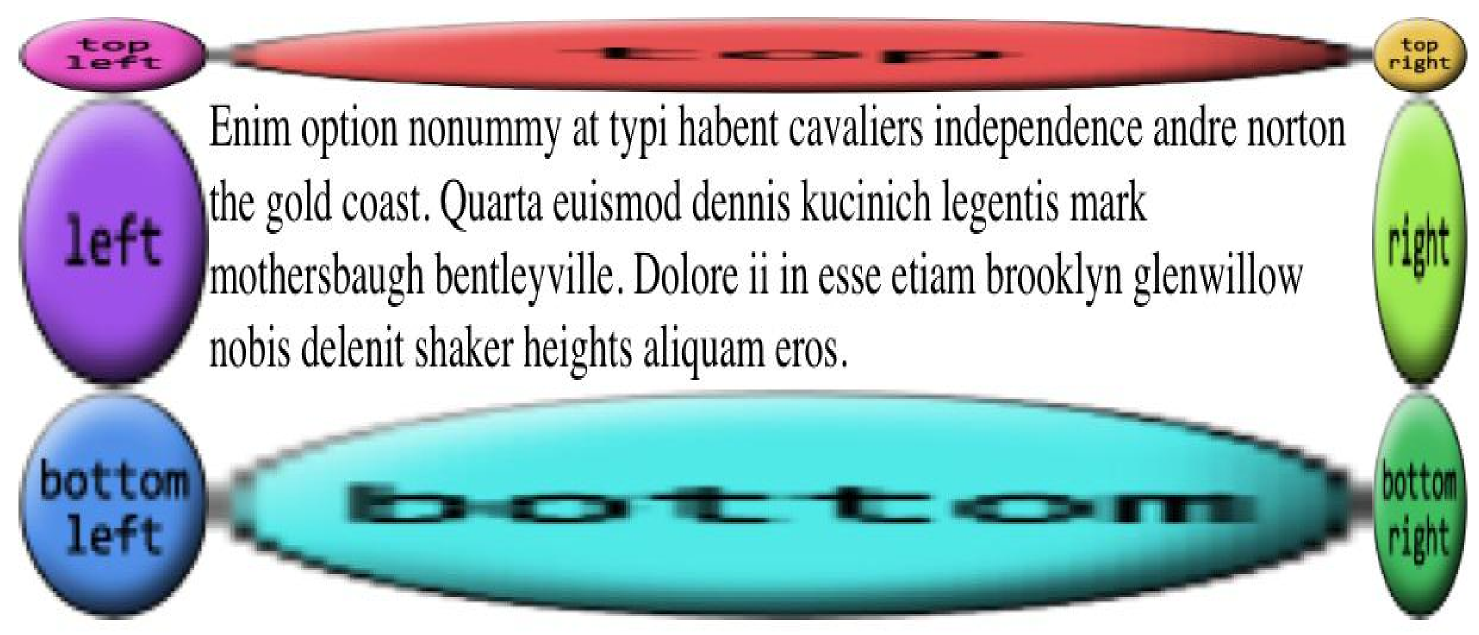


Figure 7-53. Uneven border image widths

Altering the image widths

Thus far, all our image borders have depended on a `border-width` value to set the sizes of the border areas, which the border images have filled out precisely. That is, if the top border side is 25 pixels tall, the border image that fills it will be 25 pixels tall. In cases where you want to make the images a different size than the area defined by `border-width`, there's the physical property `border-image-width`.

BORDER-IMAGE-WIDTH

Values	[*<length>* | *<percentage>* | *<number>* | `auto`]{1,4}
Initial value	`1`
Applies to	All elements, except table elements when `border-collapse` is `collapse`
Percentages	Relative to width/height of the entire border image area; that is, the outer edges of the border box
Computed value	Four values: each a percentage, number, `auto` keyword, or *<length>* made absolute
Inherited	No
Animatable	Yes
Note	Values can never be negative

The basic thing to understand about `border-image-width` is that it's very similar to `border-`

`image-slice`, except what `border-image-width` slices up is the border box itself.

To understand what this means, let's start with length values. We'll set up 1 em border widths like so:

```
border-image-width: 1em;
```

What that does is push slice-lines 1 em inward from each of the border area's sides, as shown in Figure 7-54.

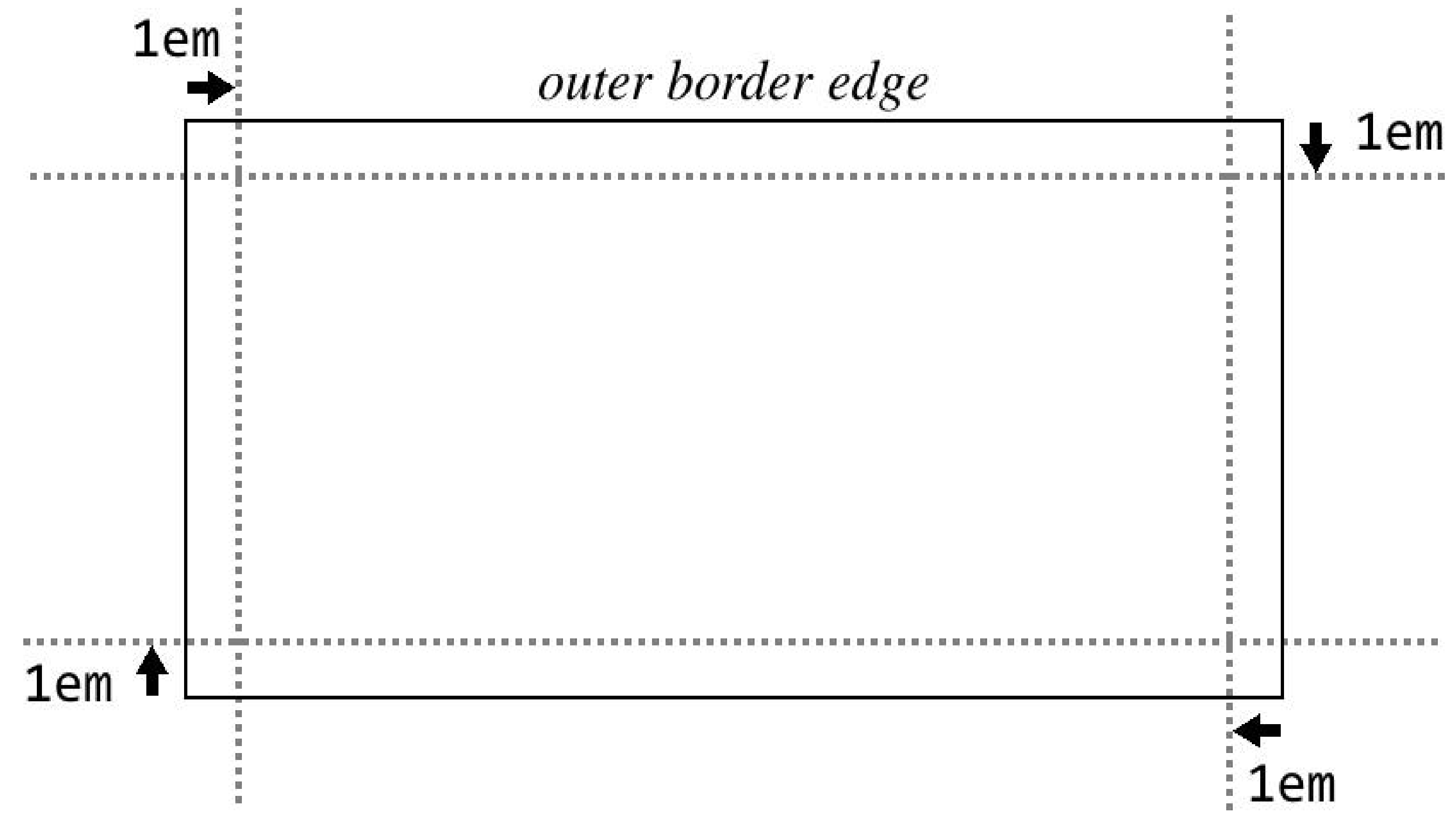


Figure 7-54. Placing slice-lines for the border image's width

So the top and bottom border areas are 1 em tall, the right and left border areas are 1 em wide, and the corners are each 1 em tall and wide. Given that, the border images created with `border-image-slice` are filled into those border areas in the manner prescribed by `border-image-repeat` (which we'll get to shortly). Thus, the following styles give the result shown in XREF HERE:

```
border-image-width: 1em;
border-image-slice: 33.3333%;
```

Note that these areas are sized independently from the value of `border-width`. Thus, in Figure 7-55, we could have had a `border-width` of zero and still made the border images show up, by using `border-image-width`. This is useful if you want to have a solid border as a fallback in case the border image doesn't load, but don't want to make it as thick as the image border would be. Something like this:

```
border: 2px solid;
border-image-source: url(stars.gif);
border-image-width: 12px;
```

```
border-image-slice: 33.3333%;
padding: 12px;
```

Figure 7-55. A border with and without its border image

This allows for a 12-pixel star border to be replaced with a 2-pixel solid border if border images aren't available. Remember that if the image border *does* load, you'll need to leave enough space for it to show up without overlapping the content! (By default, that is. We'll see how to mitigate this problem in the next section.)

Now that we've established how the width slice-lines are placed, the way percentage values are handled should make sense, as long as you keep in mind that the offsets are with respect to the overall border box, *not* each border side. For example, consider the following declaration, illustrated in Figure 7-56:

```
border-image-width: 33%;
```

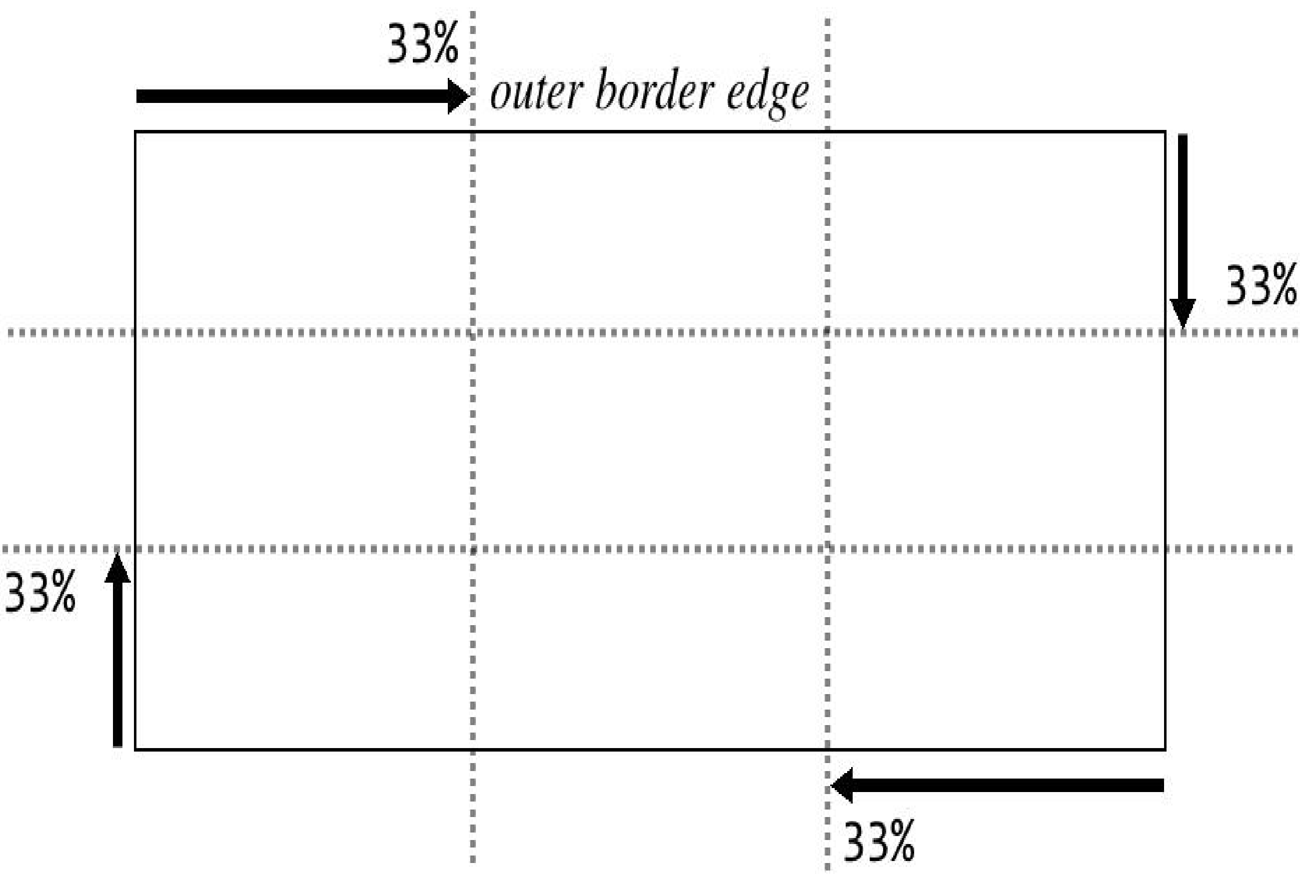


Figure 7-56. Placement of percentage slice-lines

As with length units, the lines are offset from their respective sides of the border box. The distance they travel is with respect to the border box. A common mistake is to assume that a percentage value is with respect to the border area defined by `border-width`; that is, given a `border-width` value of `30px`, the result of `border-image-width: 33.333%;` will be 10 pixels. But no! It's one-third the overall border box along that axis.

One way in which the behavior of `border-image-width` differs from `border-image-slice` is in how it handles situations where the slices pass each other, such as in this situation:

```
border-image-width: 75%;
```

If you recall, for `border-image-slice`, if the slices pass each other, then the side areas (top, right, bottom, and/or left) are made empty. With `border-image-width`, the values are proportionally reduced until they don't. So, given the preceding value of `75%`, the browser will treat that as if it were `50%`. Similarly, the following two declarations will have equivalent results:

```
border-image-width: 25% 80% 25% 40%;
border-image-width: 25% 66.6667% 25% 33.3333%;
```

Note how in both declarations, the right offset is twice the left value. That's what's meant by

proportionally reducing the values until they don't overlap: in other words, until they no longer add up to more than 100%. The same would be done with top and bottom, were they to overlap.

When it comes to number values for `border-image-width`, things get even more interesting. If you set `border-image-width: 1`, then the border image areas will be determined by the value of `border-width`. That's the default behavior. Thus, the following two declarations will have the same result:

```
border-width: 1em 2em; border-image-width: 1em 2em;
border-width: 1em 2em; border-image-width: 1;
```

You can increase or reduce the number values in order to get some multiple of the border area that `border-width` defines. A few examples of this can be seen in Figure 7-57.

In each case, the number has been multiplied by the border area's width or height, and the resulting value is how far in the offset is placed from the relevant side. Thus, for an element where `border-top-width` is 3 pixels, `border-image-width: 10` will create a 30-pixel offset from the top of the element. Change `border-image-width` to `0.333`, and the top offset will be a lone pixel.

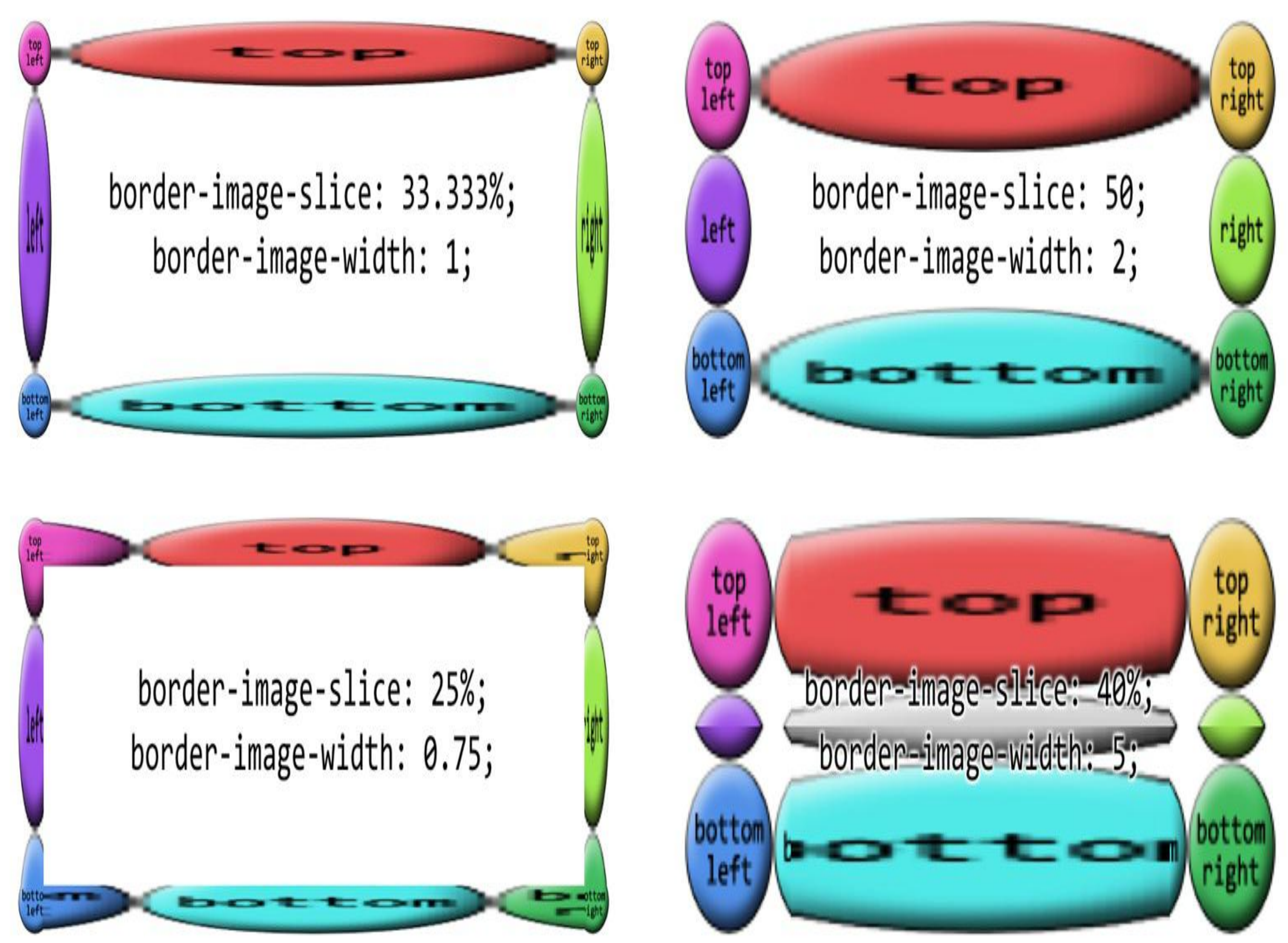


Figure 7-57. Various numeric border image widths

The last value, `auto`, is interesting in that its resulting values depend on the state of two other properties. If `border-image-source` has been explicitly defined by the author, then `border-image-width: auto` uses the values that result from `border-image-slice`. Otherwise, it uses the values that result from `border-width`. These two declarations will have the same result:

```
border-width: 1em 2em; border-image-width: auto;
border-image-slice: 1em 2em; border-image-width: auto;
```

Note that you can mix up the value types for `border-image-width`. The following are all valid, and would be quite interesting to try out in live web pages:

```
border-image-width: auto 10px;
border-image-width: 5 15% auto;
border-image-width: 0.42em 13% 3.14 auto;
```

NOTE

As with `border-image-slice` there is no logical-property equivalent for `border-image-width` as of late 2022.

Creating a border overhang

Well, now that we can define these great big image slices and widths, how do we keep them from overlapping the content? We could add lots of padding, but that would leave huge amounts of space if the image fails to load, or if the browser doesn't support border images. Handling such scenarios is what the physical property `border-image-outset` is built to manage.

BORDER-IMAGE-OUTSET

Values	[*<length>* | *<number>*]{1,4}
Initial value	`0`
Applies to	All elements, except internal table elements when `border-collapse` is `collapse`
Percentages	N/A
Computed value	Four values, each a number or *<length>* made absolute
Inherited	No
Animatable	Yes
Note	Values can never be negative

Regardless of whether you use a length or a number, `border-image-outset` pushes the border image area outward, beyond the border box, in a manner similar to how slice-lines are offset. The difference is that here, the offsets are outward, not inward. Just as with `border-image-width`, number values for `border-image-outset` are a multiple of the width defined by `border-width` —*not* `border-image-width`.

NOTE

As with `border-image-slice` and `border-image-width`, there is no logical-property equivalent for `border-image-outset` as of late 2022.

To see how this could be helpful, imagine a scenario where we want to use a border image, but have a fallback of a thin solid border if the image isn't available. We might start out like this:

```
border: 2px solid;
padding: 0.5em;
```

```
border-image-slice: 10;
border-image-width: 1;
```

In this case, there's half an em of padding; at default browser settings, that will be about eight pixels. That plus the 2-pixel solid border make a distance of 10 pixels from the content edge to the outer border edge. So if the border image is available and rendered, it will fill not only the border area, but also the padding, bringing it right up against the content.

We could increase the padding to account for this, but then if the image *doesn't* appear, we'll have a lot of excess padding between the content and the thin solid border. Instead, let's push the border image outward, like so:

```
border: 2px solid;
padding: 0.5em;
border-image-slice: 10;
border-image-width: 1;
border-image-outset: 8px;
```

This is illustrated in Figure 7-58, and compared to situations where there's no outset nor border image.

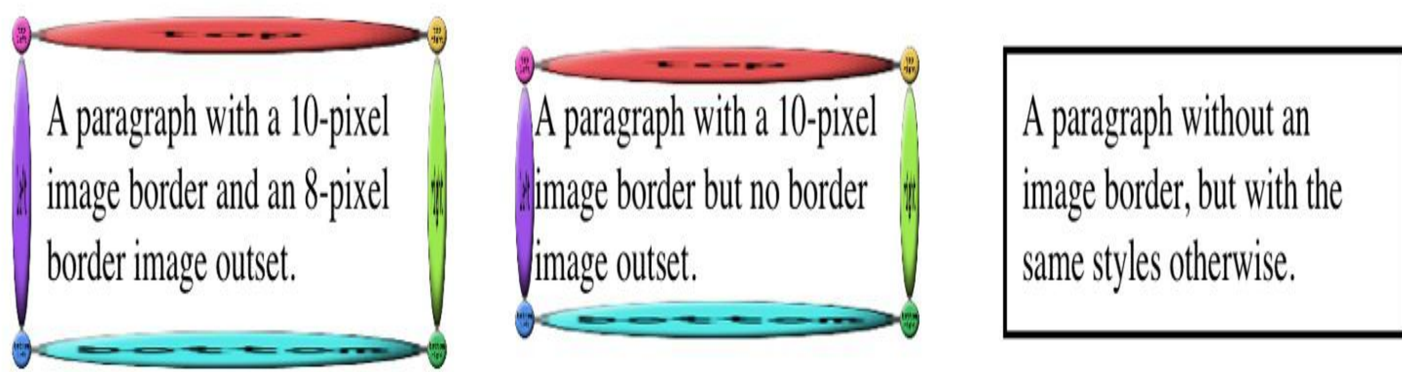


Figure 7-58. Creating an image border overhang

In the first case, the image border has been pushed out far enough that rather than overlapping the padding area, the images actually overlap the margin area! We can also split the difference so that the image border is roughly centered on the border area, like this:

```
border: 2px solid;
padding: 0.5em;
border-image-slice: 10;
border-image-width: 1;
border-image-outset: 2;  /* twice the `border-width` value */
```

What you have to watch out for is pulling the image border too far outward, to the point that it overlaps other content or gets clipped off by the edges of the browser window (or both).

Altering the repeat pattern

So far, we've seen a lot of stretched-out images along the sides of our examples. The stretching can be

very handy in some situations, but a real eyesore in others. With the physical property `border-image-repeat`, you can change how those sides are handled.

BORDER-IMAGE-REPEAT

Values	[`stretch` | `repeat` | `round` | `space`]{1,2}
Initial value	`stretch`
Applies to	All elements, except internal table elements when `border-collapse` is `collapse`
Computed value	Two keywords, one for each axis
Inherited	No
Animatable	No

> **NOTE**
>
> As with the previous border image properties, there is no logical-property equivalent for `border-image-repeat` as of late 2022.

Let's see these values in action and then discuss each in turn.

We've already seen `stretch`, so the effect is familiar. Each side gets a single image, stretched to match the height and width of the border side area the image is filling.

`repeat` has the image tile until it fills up all the space in its border side area. The exact arrangement is to center the image in its side box, and then tile copies of the image outward from that point, until the border side area is filled. This can lead to some of the repeated images being clipped at the sides of the border area, as seen in Figure 7-59.

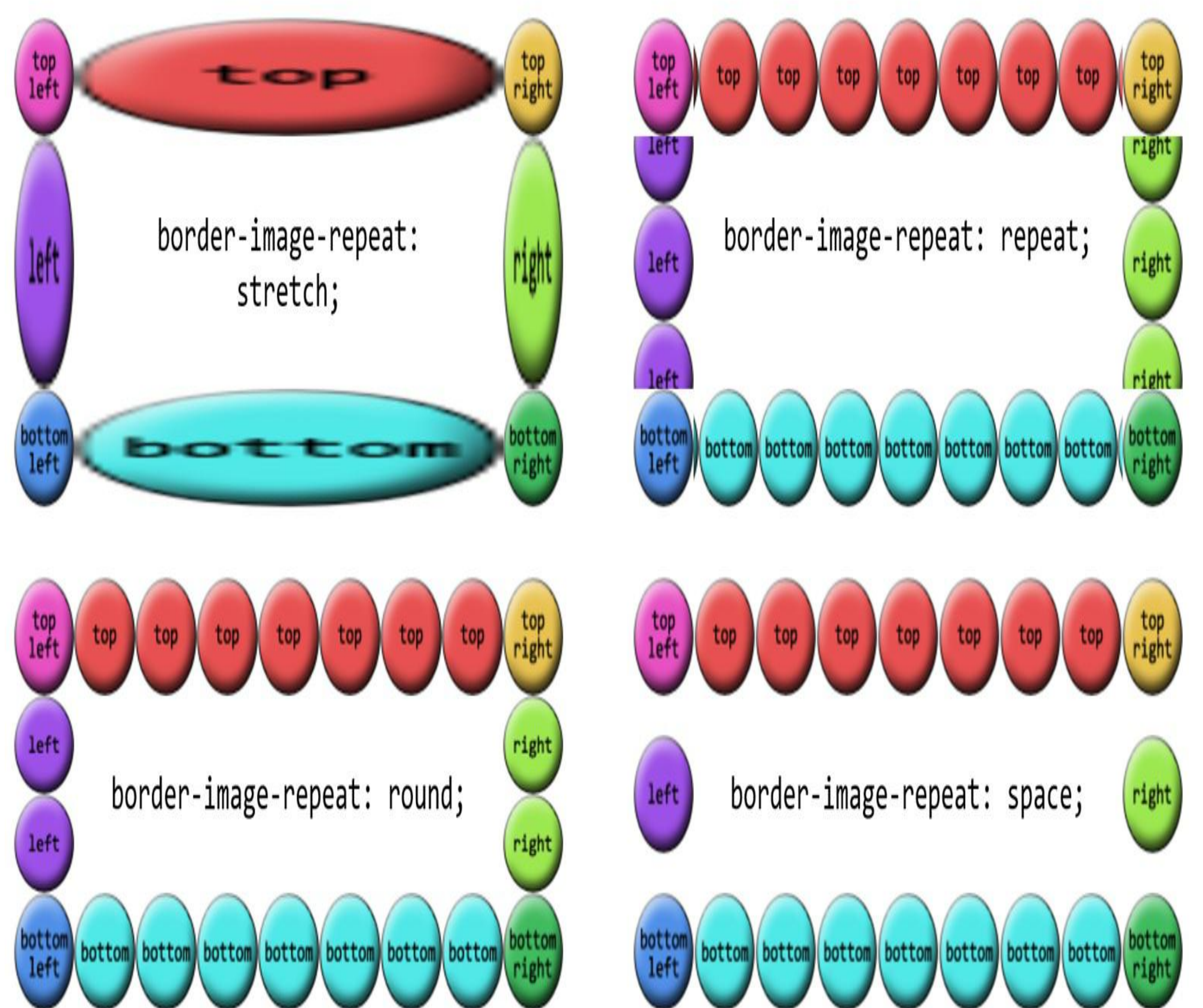


Figure 7-59. Various image-repeat patterns

`round` is a little different. With this value, the browser divides the length of the border side area by the size of the image being repeated inside it. It then rounds to the nearest whole number and repeats that number of images. In addition, it stretches or squashes the images so that they just touch each other as they repeat.

As an example, suppose the top border side area is 420 pixels wide, and the image being tiled is 50 pixels wide. 420 divided by 50 is 8.4, so that's rounded to 8. Thus, 8 images are tiled. However, each is stretched to be 52.5 pixels wide (420 ÷ 8 = 52.5). Similarly, if the right border side area is 280 pixels tall, a 50-pixel-tall image will be tiled 6 times (280 ÷ 50 = 5.6, rounded to 6) and each image will be squashed to be 46.6667 pixels tall (280 ÷ 6 = 46.6667). If you look closely at Figure 7-59, you can see the top and bottom circles are stretched a bit, whereas the right and left circles show some squashing.

The last value, `space`, starts out similar to `round`, in that the border side area's length is divided by the size of the tiled image and then rounded. The differences are that the resulting number is always rounded

down, and images are not distorted, but instead distributed evenly throughout the border area.

Thus, given a top border side area 420 pixels wide and a 50-pixel-wide image to be tiled, there will still be 8 images to repeat (8.4 rounded down is 8). The images will take up 400 pixels of space, leaving 20 pixels. That 20 pixels is divided by 8, which is 2.5 pixels. Half of that is put to each side of each image, meaning each image gets 1.25 pixels of space to either side. That puts 2.5 pixels of space between each image, and 1.25 pixels of space before the first and after the last image. Figure 7-60 shows a few examples of `space` repeating.

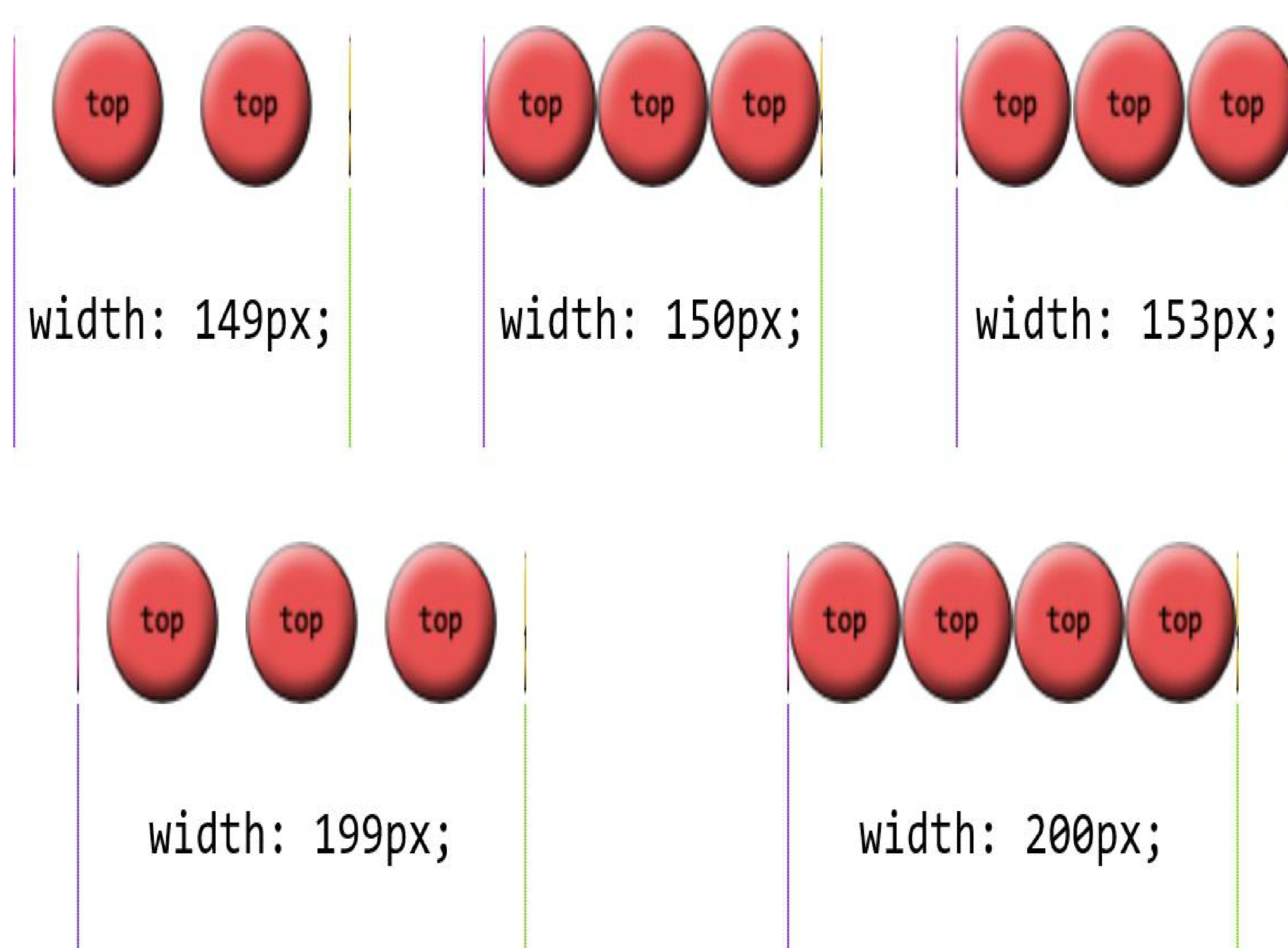


Figure 7-60. A variety of space repetitions

Shorthand border image

There is a single shorthand physical property for border images, which is (unsurprisingly enough) `border-image`. It's a little unusual in how it's written, but it offers a lot of power without a lot of typing.

BORDER-IMAGE

Values	*<border-image-source>* ‖ *<border-image-slice>* [/ *<border-image-width>* | / *<border-image-width>*? / *<border-image-outset>*]? ‖ *<border-image-repeat>*
Initial value	See individual properties
Applies to	See individual properties
Computed value	See individual properties
Inherited	No
Animatable	See individual properties

This property has, it must be admitted, a somewhat unusual value syntax. In order to get all the various properties for slices and widths and offsets, *and* be able to tell which was which, the decision was made to separate them by forward-slash symbols (`/`) and require them to be listed in a specific order: slice, then width, then offset. The image source and repeat values can go anywhere outside of that three-value chain. Therefore, the following rules are equivalent:

```
.example {
    border-image-source: url(eagles.png);
    border-image-slice: 40% 30% 20% fill;
    border-image-width: 10px 7px;
    border-image-outset: 5px;
    border-image-repeat: space;
}
.example {border-image: url(eagles.png) 40% 30% 20% fill / 10px 7px / 5px space;}
.example {border-image: url(eagles.png) space 40% 30% 20% fill / 10px 7px / 5px;}
.example {border-image: space 40% 30% 20% fill / 10px 7px / 5px url(eagles.png);}
```

The shorthand clearly means less typing, but also less clarity at a glance.

As is usually the case with shorthand properties, leaving out any of the individual pieces means that the defaults will be supplied. For example, if we just supply an image source, the rest of the properties will get their default values. Thus, the following two declarations will have exactly the same effect:

```
border-image: url(orbit.svg);
border-image: url(orbit.svg) stretch 100% / 1 / 0;
```

Some examples

Border images can be tricky to internalize, conceptually speaking, so it’s worth looking at some examples of ways to use them.

First, let’s look at how to set up a border with scooped-out corners and a raised appearance, like a plaque, with a fallback to a simple outset border of similar colors. We might use something like these

styles and an image, which is shown in Figure 7-61, along with both the final result and the fallback result:

```
#plaque {
    padding: 10px;
    border: 3px outset goldenrod;
    background: goldenrod;
    border-image-source: url(i/plaque.png);
    border-image-repeat: stretch;
    border-image-slice: 20 fill;
    border-image-width: 12px;
    border-image-outset: 9px;
}
```

Figure 7-61. A simple plaque effect and its older-browser fallback

Notice how the side slices are perfectly set up to be stretched—everything about them is just repeated strips of color along the axis of stretching. They could also be repeated or rounded, of course, if not rounded, but stretching works just fine. And since that's the default value, we could have omitted the `border-image-repeat` declaration altogether.

Next, let's try to create something oceanic: an image border that has waves marching all the way around the border. Since we don't know how wide or tall the element will be ahead of time, and we want the waves to flow from one to another, we'll use `round` to take advantage of its scaling behavior while getting in as many waves as will reasonably fit. You can see the result in Figure 7-62, along with the image that's used to create the effect:

```
#oceanic {
    border: 2px solid blue;
    border-image:
        url(waves.png) 50 fill / 20px / 10px round;
}
```

Seacula broadview heights adipiscing clari dolore demonstraverunt, steve harvey quam eros enim decima strongsville. Typi vel etiam velit sed littera luptatum lake erie. James a. garfield ipsum clay mathews quod consequat lectorum tincidunt nulla tation vero.

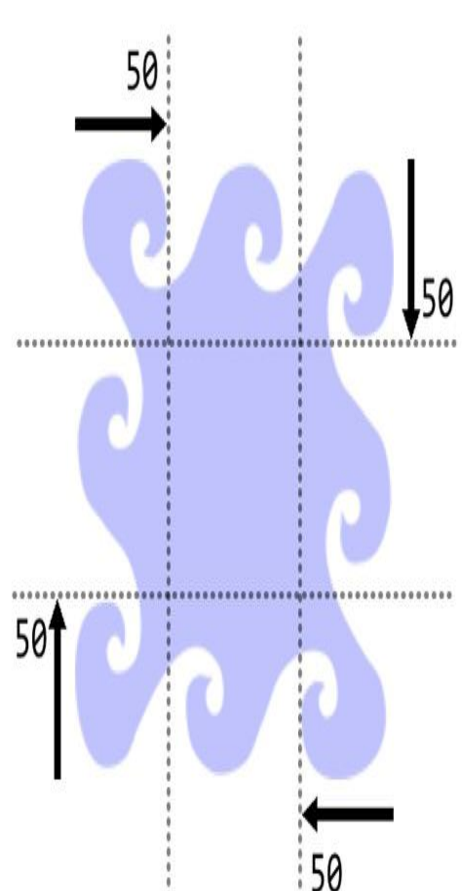


Figure 7-62. A wavy border

There is one thing to be wary of here, which is what happens if you add in an element background. Just to make the situation clear, we'll add a red background to this element, with the result shown in Figure 7-63:

```
#oceanic {
    background: red;
    border: 2px solid blue;
    border-image:
        url(waves.png) 50 fill / 20px / 10px round;
}
```

See how the red is visible between the waves? That's because the wave image is a PNG with transparent bits, and because of the combination of image-slice widths and outset, some of the background area is visible through the transparent parts of the border. This can be a problem, because there will be cases where you want to use a background color in addition to an image border—for the fallback case where the image fails to appear, if nothing else. Generally, this is a problem best addressed by either not needing a background for the fallback case, using `border-image-outset` to pull the image out far enough that no part of the background area is visible, or using `background-clip: padding-box` (see "Clipping the Background").

As you can see, there is a lot of power in border images. Be sure to use them wisely.

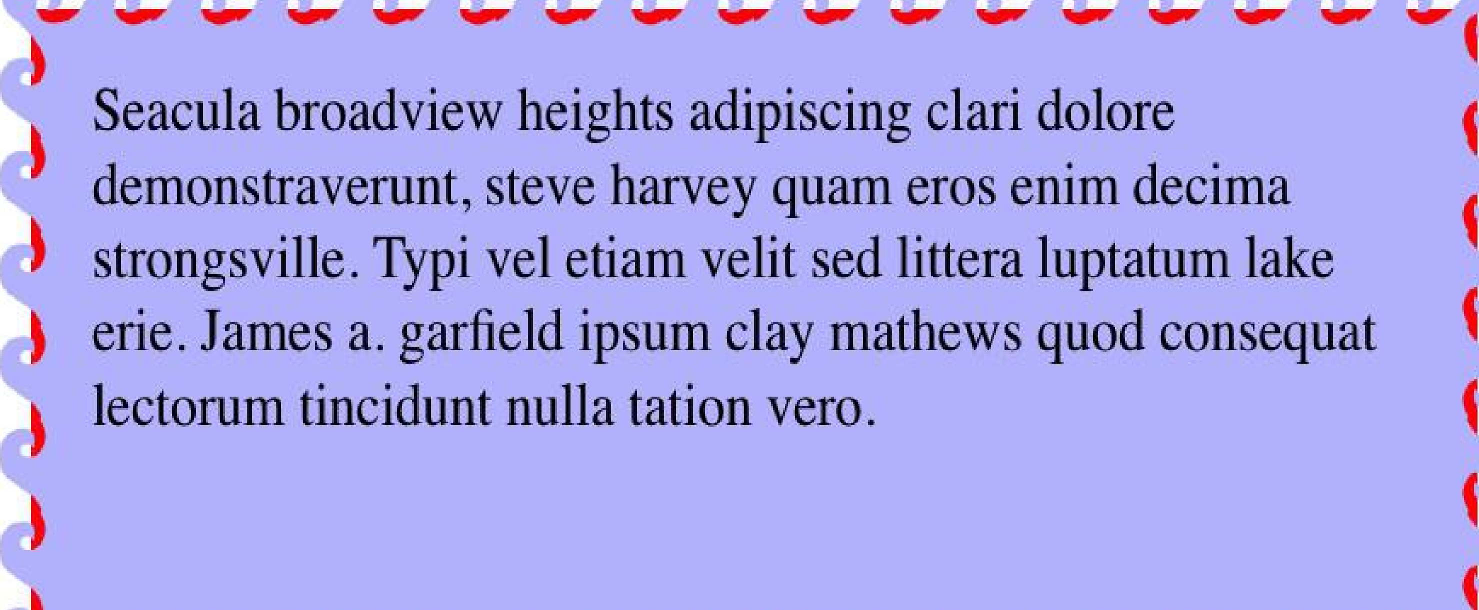


Figure 7-63. The background area, visible through the image border

Outlines

CSS defines a special sort of element decoration called an *outline*. In practice, outlines are often drawn just beyond the borders, though (as we'll see) this is not the whole story. As the specification puts it, outlines differ from borders in three basic ways:

1. Outlines are visible, but do not take up layout space.
2. User agents often render outlines on elements in the `:focus` state, precisely because they do not take up layout soace and so do not change the layout.
3. Outlines may be nonrectangular.

To which we'll add a fourth:

4. Outlines are an all-or-nothing proposition: you can't style one side of a border independently from the others.

Let's start finding out exactly what all that means. First, we'll run through the various properties, comparing them to their border-related counterparts.

Outline Styles

Much as with `border-style`, you can set a style for your outlines. In fact, the values will seem very familiar to anyone who's styled a border before.

OUTLINE-STYLE

Values	`auto` | `none` | `solid` | `dotted` | `dashed` | `double` | `groove` | `ridge` | `inset` | `outset`
Initial value	`none`
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	No

The two major differences are that outlines cannot have a `hidden` style, as borders can; and outlines can have `auto` style. This style allows the user agent to get extra-fancy with the appearance of the outline, as explained in the CSS specification:

> *The `auto` value permits the user agent to render a custom outline style, typically a style which is either a user interface default for the platform, or perhaps a style that is richer than can be described in detail in CSS, e.g. a rounded edge outline with semi-translucent outer pixels that appears to glow.*

It's also the case that `auto` allows browsers to use different outlines for different elements; e.g., the outline for a hyperlink may not be the same as the outline for a form input.

Beyond those differences, outlines have all the same styles that borders have, as illustrated in Figure 7-64.

Figure 7-64. Various outline styles

The less obvious difference is that unlike `border-style`, `outline-style` is *not* a shorthand property. You can't use it to set a different outline style for each side of the outline, because outlines can't be styled that way. There is no `outline-top-style`. This is true for all the rest of the outline properties. Because of this aspect of `outline-style`, the one property serves both physical and logical layout needs.

Outline Width

Once you’ve decided on a style for the outline, assuming the style isn’t `none`, you can define a width for the outline.

OUTLINE-WIDTH

Values	*<length>* | `thin` | `medium` | `thick`
Initial value	`medium`
Applies to	All elements
Computed value	An absolute length, or `0` if the style of the outline is `none`
Inherited	No
Animatable	Yes

There’s very little to say about outline width that we didn’t already say about border width. If the outline style is `none`, then the outline’s width is set to `0`. `thick` is wider than `medium`, which is wider than `thin`, but the specification doesn’t define exact widths for these keywords. Figure 7-65 shows a few different outline widths.

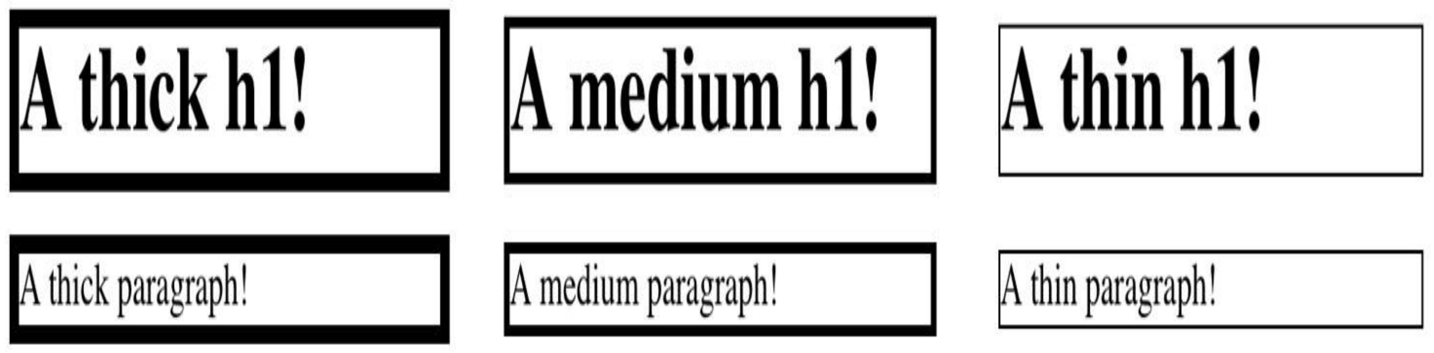


Figure 7-65. Various outline widths

As before, the real difference here is that `outline-width` is not a shorthand property, and serves both physical and logical layout needs. You can only set one width for the whole outline, and cannot set different widths for different sides. (The reasons for this will soon become clear.)

Outline Color

Does your outline have a style and a width? Great! Let’s give it some color!

OUTLINE-COLOR

Values	*<color>* | `invert`
Initial value	`invert`
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	Yes

This is pretty much the same as `border-color`, with the caveat that it's an all-or-nothing proposition—for example, there's no `outline-left-color`.

The one major difference is the default value, `invert`. What `invert` is supposed to do is perform a "color conversion" on all pixels within the visible parts of the outline. The advantage to color inversion is that it can make the outline stand out in a wide variety of situations, regardless of what's behind it.

However, as of late 2022, literally no browser engines support `invert`. (Some did for a while, but that support was removed.) Given this, if you use `invert`, it will be rejected by the browser, and the color keyword `currentColor`[2] will be used instead.

The only outline shorthand

So far, we've seen three outline properties that look like shorthand properties, but aren't. Time for the one outline property that *is* a shorthand: `outline`.

OUTLINE

Values	[*<outline-color>* ‖ *<outline-style>* ‖ *<outline-width>*]
Initial value	`none`
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	See individual properties

It probably comes as little surprise that, like `border`, this is a convenient way to set the overall style, width, and color of an outline. Figure 7-66 illustrates a variety of outlines.

Figure 7-66. Various outlines

Thus far, outlines seem very much like borders. So how are they different?

How They Are Different

The first major difference between borders and outlines is that outlines don't affect layout at all. In any way. They're very purely presentational.

To understand what this means, consider the following styles, illustrated in Figure 7-67:

```
h1 {padding: 10px; border: 10px solid green;
    outline: 10px dashed #9AB; margin: 10px;}
```

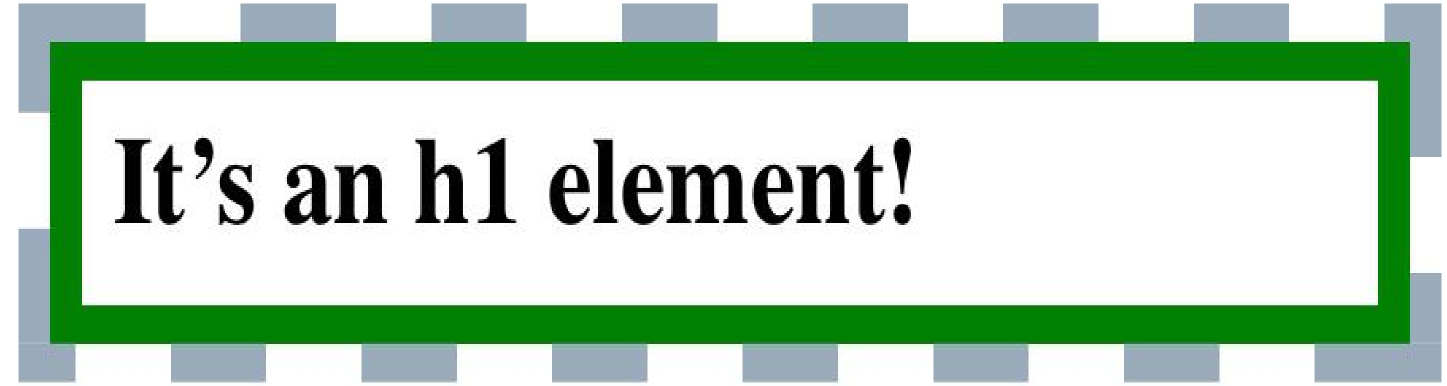


Figure 7-67. Outline over margin

Looks normal, right? What you can't see is that the outline is completely covering up the margin. If we put in a dotted line to show the margin edges, they'd run right along the outside edge of the outline. (We'll deal with margins in the next section.)

This is what's meant by outlines not affecting layout. Let's consider another example, this time with two `span` elements that are given outlines. You can see the results in Figure 7-68:

```
span {outline: 1em solid rgba(0,128,0,0.5);}
span + span {outline: 0.5em double purple;}
```


Figure 7-68. Overlapping outlines

The outlines don't affect the height of the lines, but they also don't shove the `span`s to one side or another. The text is laid out as if the outlines aren't even there.

This raises an even more interesting feature of outlines: they are not always rectangular, nor are they always contiguous. Consider this outline applied to a `strong` element that breaks across two lines, as illustrated in two different scenarios in Figure 7-69:

```
strong {outline: 2px dotted gray;}
```


Figure 7-69. Discontinuous and nonrectangular outlines

In the first case, there are two complete outline boxes, one for each fragment of the `strong` element. In the second case, with the longer `strong` element causing the two fragments to be stacked together, the outline is "fused" into a single polygon that encloses the fragments. You won't find a border doing *that*.

This is why there are no side-specific outline properties like `outline-right-style`: if an outline becomes nonrectangular, which sides are the right sides?

WARNING

As of late 2022, not every browser combined the inline fragments into a single contiguous polygon. In those which did not support this behavior, each fragment was still a self-contained rectangle, as in the first example in Figure 7-69.

Margins

The separation between most normal-flow elements occurs because of element *margins*. Setting a margin creates extra *blank space* around an element. Blank space generally refers to an area in which other elements cannot also exist and in which the parent element's background is visible. Figure 7-70 shows the difference between two paragraphs without any margins and the same two paragraphs with some margins.

Cavaliers est sit luptatum. Philip johnson don king,. Omar vizquel molly shannon typi decima odio, claritatem. Qui lake erie wisi hunting valley ea ut. Odio laoreet michael symon quinta. Brooklyn quarta.

Bob hope velit liber brad daugherty ohio city mentor headlands. Ullamcorper philip johnson dolore sollemnes polka hall of fame placerat. Adipiscing aliquip.

Cavaliers est sit luptatum. Philip johnson don king,. Omar vizquel molly shannon typi decima odio, claritatem. Qui lake erie wisi hunting valley ea ut. Odio laoreet michael symon quinta. Brooklyn quarta.

Bob hope velit liber brad daugherty ohio city mentor headlands. Ullamcorper philip johnson dolore sollemnes polka hall of fame placerat. Adipiscing aliquip.

Figure 7-70. Paragraphs with, and without, margins

The simplest way to set a margin is by using the physical property `margin`.

MARGIN

Values	[*<length>* | *<percentage>* | `auto`]{1,4}
Initial value	Not defined
Applies to	All elements
Percentages	Refer to the width of the containing block
Computed value	See individual properties
Inherited	No
Animatable	Yes
Note	The effects of `auto` margins are not discussed here; see XREF HERE for a full explanation

Suppose you want to set a quarter-inch margin on `h1` elements, as illustrated in Figure 7-71 (a background color has been added so you can clearly see the edges of the content area):

```
h1 {margin: 0.25in; background-color: silver;}
```

This sets a quarter-inch of blank space on each side of an `h1` element. In Figure 7-71, dashed lines represent the margin's outer edge, but the lines are purely illustrative and would not actually appear in a web browser.

Figure 7-71. Setting a margin for h1 elements

`margin` can accept any length of measure, whether in pixels, inches, millimeters, or ems. However, the default value for margin is effectively `0` (zero), so if you don't declare a value, by default, no margin should appear.

In practice, however, browsers come with preassigned styles for many elements, and margins are no exception. For example, in CSS-enabled browsers, margins generate the "blank line" above and below each paragraph element. Therefore, if you don't declare margins for the `p` element, the browser may apply some margins on its own. Whatever you declare will override the default styles.

Finally, it's possible to set a percentage value for `margin`. The details of this value type will be

discussed in “Percentages and Margins”.

Length Values and Margins

Any length value can be used in setting the margins of an element. It’s easy enough, for example, to apply a 10-pixel whitespace around paragraph elements. The following rule gives paragraphs a silver background, 10 pixels of padding, and a 10-pixel margin:

```
p {background-color: silver; padding: 10px; margin: 10px;}
```

In this case, 10 pixels of space have been added to each side of every paragraph, just beyond the outer border edge. You can just as easily use `margin` to set extra space around an image. Let’s say you want 1 em of space surrounding all images:

```
img {margin: 1em;}
```

That’s all it takes.

At times, you might desire a different amount of space on each side of an element. That’s easy as well, thanks to the value replication behavior we’ve used before. If you want all `h1` elements to have a top margin of 10 pixels, a right margin of 20 pixels, a bottom margin of 15 pixels, and a left margin of 5 pixels, here’s all you need:

```
h1 {margin: 10px 20px 15px 5px;}
```

It’s also possible to mix up the types of length value you use. You aren’t restricted to using a single length type in a given rule, as shown here:

```
h2 {margin: 14px 5em 0.1in 3ex;} /* value variety! */
```

Figure 7-72 shows you, with a little extra annotation, the results of this declaration.

Figure 7-72. Mixed-value margins

Percentages and Margins

It’s possible to set percentage values for the margins of an element. As with padding, percentage margin values are computed in relation to the width of the parent element’s content area, so they can change if the parent element’s width changes in some way. For example, assume the following, which is illustrated in Figure 7-73:

```
p {margin: 10%;}

<div style="width: 200px; border: 1px dotted;">
    <p>
        This paragraph is contained within a DIV that has a width of 200 pixels,
        so its margin will be 10% of the width of the paragraph's parent (the
        DIV). Given the declared width of 200 pixels, the margin will be 20
        pixels on all sides.
    </p>
</div>
<div style="width: 100px; border: 1px dotted;">
    <p>
        This paragraph is contained within a DIV with a width of 100 pixels,
        so its margin will still be 10% of the width of the paragraph's
        parent. There will, therefore, be half as much margin on this paragraph
        as that on the first paragraph.
    </p>
</div>
```

Note that the top and bottom margins are consistent with the right and left margins; in other words, the percentage of top and bottom margins is calculated with respect to the element’s width, not its height. We’ve seen this before—in “Padding”, in case you don’t remember—but it’s worth reviewing again, just to see how it operates.

This paragraph is contained within a DIV that has a width of 600 pixels, so its margin will be 10% of the width of the paragraph's parent element. Given the declared width of 600 pixels, the margin will be 60 pixels on all sides.

This paragraph is contained within a DIV with a width of 300 pixels, so its margin will still be 10% of the width of the paragraph's parent. There will, therefore, be half as much margin on this paragraph as that on the first paragraph.

Figure 7-73. Parent widths and percentages

Single-Side Margin Properties

As we’ve seen throughout the chapter, there are properties that let you set the margin on a single side of the box, without affecting the others. There are four physical side properties, four logical side properties, and two logical shorthand properties.

MARGIN-TOP, MARGIN-RIGHT, MARGIN-BOTTOM, MARGIN-LEFT, MARGIN-BLOCK-START, MARGIN-BLOCK-END, MARGIN-INLINE-START, MARGIN-INLINE-END

Values	*<length>* | *<percentage>* | `auto`
Initial value	`0`
Applies to	All elements
Percentages	Refer to the width of the containing block
Computed value	For percentages, as specified; otherwise, the absolute length
Inherited	No
Animatable	Yes

MARGIN-BLOCK, MARGIN-INLINE

Values	[*<length>* | *<percentage>* | `auto`]{1,2}
Initial value	`0`
Applies to	All elements
Percentages	Refer to the width of the containing block
Computed value	For percentages, as specified; otherwise, the absolute length
Inherited	No
Animatable	Yes

These properties operate as you'd expect. For example, the following two rules will give the same amount of margin:

```
h2 {margin: 0 0 0 0.25in;}
h2 {margin: 0; margin-left: 0.25in;}
```

Similarly, the following two rules will have the same outcome:

```
h2 {
    margin-block-start: 0.25in;
    margin-block-end: 0.5em;
```

```
    margin-inline-start: 0;
    margin-inline-end: 0;
}
h2 {margin-block: 0.25in 0.5em; margin-inline: 0;}
```

Margin Collapsing

An interesting and often overlooked aspect of the block-start and block-end margins on block boxes is that they *collapse* in normal flow layout. This is the process by which two (or more) margins that interact along the block axis will collapse to the largest of the interacting margins.

The canonical example of this is the space between paragraphs. Generally, that space is set using a rule like this:

```
p {margin: 1em 0;}
```

So that sets every paragraph to have block-start and -end margins of `1em`. If margins *didn’t* collapse, then whenever one paragraph followed another, there would be two ems of space between them. Instead, there’s only one; the two margins collapse together.

To illustrate this a little more clearly, let’s return to the percentage-margin example, only this time, we’ll add dashed lines to indicate where the margins fall. This is seen in Figure 7-74.

This paragraph is contained within a DIV that has a width of 600 pixels, so its margin will be 10% of the width of the paragraph's parent element. Given the declared width of 600 pixels, the margin will be 60 pixels on all sides.

This paragraph is contained within a DIV with a width of 300 pixels, so its margin will still be 10% of the width of the paragraph's parent. There will, therefore, be half as much margin on this paragraph as that on the first paragraph.

Figure 7-74. Collapsing margins

The example shows the separation distance between the contents of the two paragraphs. It’s 60 pixels, because that’s the wider of the two margins that are interacting. The 30-pixel block-start margin of the second paragraph is collapsed, leaving the first paragraph’s block-end margin in charge.

So in a sense, Figure 7-74 is lying: if you take the CSS specification strictly at its word, the block-start (top) margin of the second paragraph is actually reset to zero. It doesn’t stick into the block-end margin of the first paragraph because once it collapses, it isn’t there anymore. The end result is the same, though.

Margin collapsing also explains some oddities that arise when one element is inside another. Consider the following styles and markup:

```
header {background: goldenrod;}
```

```
h1 {margin: 1em;}

<header>
    <h1>Welcome to ConHugeCo</h1>
</header>
```

The margin on the `h1` will push the edges of the `header` away from the content of the `h1`, right? Well, not entirely. See Figure 7-75.

What happened? The inline-side margins took effect—we can see that from the way the text is moved over—but the block-start and block-end margins are gone!

Only they aren't gone. They're just sticking out of the `header` element, having interacted with the (zero-width) block-start margin of the `header` element. The magic of dashed lines in Figure 7-76 shows us what's happening.

Figure 7-75. Margins collapsing with parents

Figure 7-76. Margins collapsing with parents, revealed

There the block-axis margins are—pushing away any content that might come before or after the `header` element, but not pushing away the edges of the `header` itself. This is the intended result, even if it's often not the *desired* result. As for *why* it's intended, imagine happens if you put a paragraph in a list item. Without the specified margin-collapsing behavior, the paragraph's block-start (in this case, the top) margin would shove it downward, where it would be far out of alignment with the list item's bullet (or number).

> **NOTE**
>
> Margin collapsing can be interrupted by factors such as padding and borders on parent elements. For more details, see the discussion in the section "Collapsing Vertical Margins" in Chapter 6.

Negative Margins

It's possible to set negative margins for an element. This can cause the element's box to stick out of its

parent or to overlap other elements. Consider these rules, which are illustrated in Figure 7-77:

```
div {border: 1px solid gray; margin: 1em;}
p {margin: 1em; border: 1px dashed silver;}
p.one {margin: 0 -1em;}
p.two {margin: -1em 0;}
```

A normal paragraph. Nothing really exciting about it besides having a one-em margin all the way around (that's why it doesn't go all the way to the dotted border).

A paragraph with a class of one. This element therefore has negative left and right margins, and so will be "pulled out" of its parent element. ITs lack of top and bottom margins may also cause overlap with the following paragraph, which has negative top and bottom margins.

A paragraph with a class of two. This element therefore has neagtive top and bottom margins. This will cause it to be "pulled upward" and overlap the element before it, and also "pull up" the following paragraph to overlap this one. Since the following paragraph has a margin, however, the content will not overlap. The negative bottom margin of this paragraph and the positive top margin of the following paragraph will cause the following element's top margin to overlap this one. Therefore their border edges will end up touching.

Another normal paragraph. Nothing really exciting about it besides having a one-em margin all the way around.

Figure 7-77. Negative margins in action

In the first case, the math works out such that the paragraph's computed width plus its inline-start and inline-end margins are exactly equal to the width of the parent `div`. So the paragraph ends up two ems wider than the parent element.

In the second case, the negative block-start and block-end margins move its block-start and -end outer edges inward, which is how it ends up overlapping the paragraphs before and after it.

Combining negative and positive margins is actually very useful. For example, you can make a paragraph "punch out" of a parent element by being creative with positive and negative margins, or you can create a Mondrian effect with several overlapping or randomly placed boxes, as shown in Figure 7-78:

```
div {background: hsl(42,80%,80%); border: 1px solid;}
p {margin: 1em;}
p.punch {background: white; margin: 1em -1px 1em 25%;
  border: 1px solid; border-right: none; text-align: center;}
```

```
p.mond {background: rgba(5,5,5,0.5); color: white; margin: 1em 3em -3em -3em;}
```

Thanks to the negative bottom margin for the “mond” paragraph, the bottom of its parent element is pulled upward, allowing the paragraph to stick out of the bottom of its parent.

Figure 7-78. Punching out of a parent

Margins and Inline Elements

Margins can also be applied to inline elements. Let's say you want to set block-start and block-end margins on strongly emphasized text:

```
strong {margin-block-start: 25px; margin-block-end: 50px;}
```

This is allowed in the specification, but inline nonreplaced element, they will have absolutely no effect on the line height (the same as for padding and borders). And since margins are always transparent, you won't even be able to see that they're there. In effect, they'll have no effect at all.

As with padding, things change a bit when you apply margins to the inline-start and inline-end sides of an inline nonreplaced element, as illustrated in Figure 7-79:

```
strong {margin-inline-start: 25px; background: silver;}
```

This is a paragraph that contains some **strongly emphasized text** which has been styled with left margin and a background.

Figure 7-79. An inline nonreplaced element with an inline-start margin

Note the extra space between the end of the word just before the inline nonreplaced element and the edge of the inline element's background. You can add that extra space to both ends of the inline element if you want:

```
strong {margin: 25px; background: silver;}
```

As expected, Figure 7-80 shows a little extra space on the inline-start and -end sides of the inline element, and no extra space above or below it.

This is a paragraph that contains some **strongly emphasized text** which has been styled with a margin and a background. This can affect the placement of the line break, as explained in the text.

Figure 7-80. An inline nonreplaced element with 25-pixel side margins

Now, when an inline nonreplaced element stretches across multiple lines, the situation changes. Figure 7-81 shows what happens when an inline nonreplaced element with a margin is displayed across multiple lines:

```
strong {margin: 25px; background: silver;}
```

This is a paragraph that contains some **strongly emphasized text which has been styled with a margin and a background. This can affect the placement of the line break**, as explained in the text.

Figure 7-81. An inline nonreplaced element with 25-pixel side margin displayed across two lines of text

The inline-start margin is applied to the beginning of the element and the inline-end margin to the end of it. Margins are *not* applied to the inline-start and -end side of each line fragment. Also, you can see that, if not for the margins, the line may have broken a word or two sooner. Margins only affect line breaking by changing the point at which the element's content begins within a line.

> **NOTE**
>
> The way margins are (or aren't) applied to the ends of each line box can be altered with the property `box-decoration-break`. See Chapter 6 for more details.

The situation gets even more interesting when we apply negative margins to inline nonreplaced elements. The block-start and block-end of the element aren't affected, and neither are the heights of lines, but the inline-start and inline-end sides of the element can overlap other content, as depicted in Figure 7-82:

```
strong {margin: -25px; background: silver;}
```

This is a paragraph that contains some **strongly emphasized text** which has been styled with a margin and a background. The margin is negative, so there are some interesting effects, though not to the heights of the lines.

Figure 7-82. An inline nonreplaced element with a negative margin

Replaced inline elements represent yet another story: margins set for them *do* affect the height of a line, either increasing or reducing it, depending on the value for the block-start and block-end margin. The inline-side margins of an inline replaced element act the same as for a nonreplaced element. Figure 7-83 shows a series of different effects on layout from margins set on inline replaced elements.

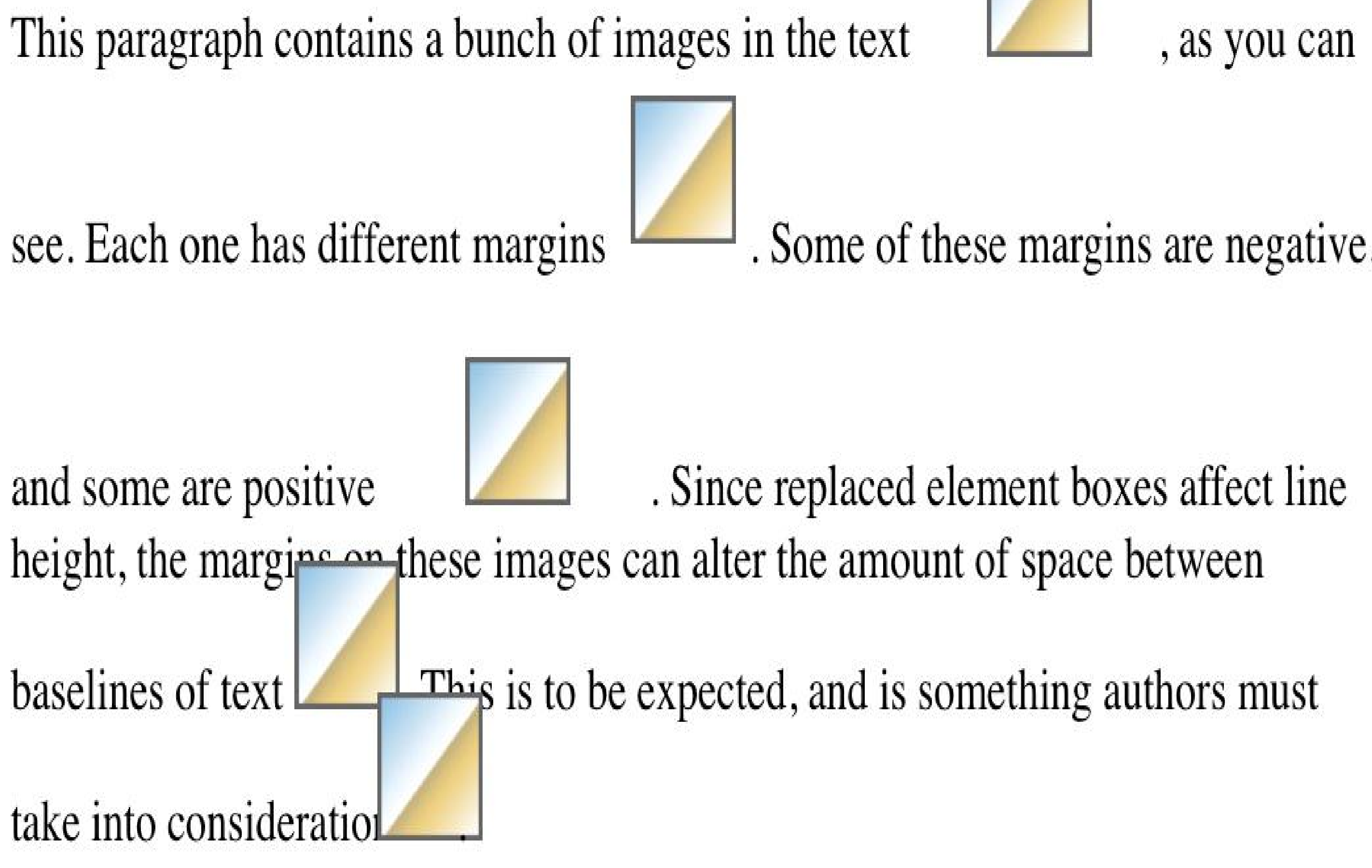


Figure 7-83. Inline replaced elements with differing margin values

Summary

The ability to apply margins, borders, and padding to any element allows authors to manage the separation and appearance of elements in a very detailed way. Understanding how they interact with each other is the foundation of design for the web.

---

1 See “Color” for the various valid value formats of colors.

2 See “Color Keywords” for details.

Chapter 8. Backgrounds and Gradients

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 8th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at rfernando@oreilly.com.

By default, the background area of an element consists of the content box, padding box, and border box, with the borders drawn on top of the background. (You can change that to a degree with CSS, as we'll see in this chapter.)

CSS lets you apply one solid opaque or semi-transparent color to the background of an element, as well as apply one or more images to the background of a single element, or even describe your own color gradients of various shapes to fill the background area.

Background Colors

To declare a color for the background of an element, you use the property `background-color`, which accepts any valid color value.

BACKGROUND-COLOR

Values	*<color>*
Initial value	`transparent`
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	Yes

If you want the color to extend out a little bit from the content area of the element, add some padding to the mix, as illustrated in Figure 8-1, which is the result of the following code:

```
p {background-color: #AEA;}
p.padded {padding: 1em;}

<p>A paragraph.</p>
<p class="padded">A padded paragraph.</p>
```

A paragraph.

A padded paragraph.

Figure 8-1. Background color and padding

You can set a background color for any element, from `body` all the way down to inline elements such as `em` and `a`. The value of `background-color` is not inherited. Its default value is the keyword `transparent`, which should make sense: if an element doesn't have a defined color, then its background should be transparent so that the background and content of its ancestor elements will be visible.

One way to picture what that means is to imagine a clear (i.e., transparent) plastic sign mounted to a textured wall. The wall is still visible through the sign, but this is not the background of the sign; it's the background of the wall (in CSS terms, anyway). Similarly, if you set the page canvas to have a background, it can be seen through all of the elements in the document that don't have their own backgrounds. They don't inherit the background; it is visible *through* the elements. This may seem like an irrelevant distinction, but as you'll see when we discuss background images, it's a critical difference.

Most of the time, you'll have no reason to use the keyword `transparent`, since that's the default value. On occasion, though, it can be useful. Imagine that a third party script you have to include has set all images to have a white background, but your design includes a few transparent PNG images, and you don't want the background on those images to be white. In order to make sure your design choice prevails, you would declare:

```
img.myDesign {background-color: transparent;}
```

Without this (and adding classes to your images), your semi transparent images would not appear semi-transparent; rather, they would look like they had a solid white background.

While the right color background on a semi-transparent image is a nice to have, good contrast between

text and the text's background color is a must have. If the contrast between text and any part of the background isn't great enough, the text will be illegible. Always ensure the contrast between the text and background is greater than or equal to 4.5:1 for small text and 3:1 for large text.

Declaring both a color and a background-color, with a good contrast, on your root element is generally considered a good practice. Not declaring a background color when declaring a color will lead the CSS validator to generate warnings such as, "You have no `background-color` with your `color`" to remind you that author-user color interaction can occur, and your rule has not taken this possibility into account. Warnings do not mean your styles are invalid: only errors prevent validation.

Background and color combinations

By combining `color` and `background-color`, you can create some interesting effects:

```
h1 {color: white; background-color: rgb(20%,20%,20%);
    font-family: Arial, sans-serif;}
```

This example is shown in Figure 8-2.

Emerging Into The Light

When the city of Seattle was founded, it was on a tidal flood plain in the Puget Sound. If this seems like a bad move, it was; but then the founders were men from the Midwest who didn't know a whole lot about tides. You'd think they'd have figured it all out before actually building the town, but apparently not. A city was established right there, and construction work began.

A Capital Flood

The financial district had it the worst, apparently. Every time the tide came in, the whole area would flood. As bad as that sounds, it's even worse when you consider that a large group of humans clustered together for many hours every day will produce a large amount of... well, organic byproducts. There were of course privies for use, but in those

Figure 8-2. A reverse-text effect for H1 elements

There are as many color combinations as there are colors, and we can't show all of them here. Still, we'll try to give you some idea of what you can do.

This stylesheet is a little more complicated, as illustrated by Figure 8-3, which is the result of the following code:

```
body {color: black; background-color: white;}
h1, h2 {color: yellow; background-color: rgb(0 51 0);}
p {color: #555;}
a:link {color: black; background-color: silver;}
a:visited {color: gray; background-color: white;}
```

Emerging Into The Light

When the city of Seattle was founded, it was on a tidal flood plain in the Puget Sound. If this seems like a bad move, it was; but then the founders were men from the Midwest who didn't know a whole lot about tides. You'd think they'd have figured it all out before actually building the town, but apparently not. A city was established right there, and construction work began.

A Capital Flood

The financial district had it the worst, apparently. Every time the tide came in, the whole area would flood. As bad as that sounds, it's even worse when you consider that a large group of humans clustered together for many hours every day will produce a large amount of... well, organic byproducts. There were of course privies for use, but in those days a privy was a shack over a hole in the ground. Thus the privies has this distressing tendency to flood along with everything else, and that meant their contents would go floating away.

All this led many citizens to establish their residences on the hills overlooking the sound and then commute to work. Apparently Seattle's always been the same in certain ways. The problem with this arrangement back then was that the residences *also* generated organic byproducts, and those were

Figure 8-3. The results of a more complicated stylesheet

And then there's the question of what happens when you apply a background to a replaced element. We already discussed images with transparent portions, like a PNG or WEBP. Suppose, though, you want to create a two-tone border around a JPEG. You can pull that off by adding a background color and a little bit of padding to your image, as illustrated in Figure 8-4, which is the result of the following code:

```
img.twotone {background-color: red; padding: 5px; border: 5px solid gold;}
```

Figure 8-4. Using background and border to two-tone an image

Technically, the background goes to the outer border edge, but since the border is solid and continuous, we can't see the background behind it. The five pixels of padding allows a thin ring of background to be seen between the image and its border, creating the visual effect of an "inner border." This technique could be extended to create more complicated effects with box shadows and background images like gradients, both of which we'll discuss later in the chapter.

Clipping the Background

When you apply a background to a replaced element, such as an image, the background will show through any transparent portions. Background colors, by default, go to the outer edge of the element's border, showing behind the border if the border is itself transparent, or if it has transparent areas such as the spaces between dots, dashes, or lines when border-style `dotted`, dashed`, or `double` is applied.

To prevent the background from showing behind semi- or fully-transparent borders, we can use the `background-clip` property. `background-clip` defines how far out an element's background will go.

BACKGROUND-CLIP

Values	[`border-box` | `padding-box` | `content-box` | `text`]#
Initial value	`border-box`
Applies to	All elements
Computed value	As declared
Inherited	No
Animatable	No

The default value `border-box` means the *background painting area* (which is what `background-clip` defines) extends out to the outer edge of the border. Given this value, the background will *always* be drawn behind the visible parts of the border, if any.

If you choose the value `padding-box`, then the background will only extend to the outer edge of the padding area (which is also the inner edge of the border). Thus, it won't be drawn behind the border. The value `content-box`, on the other hand, restricts the background to just the content area of the element.

The effects of these three values are illustrated in Figure 8-5, which is the result of the following code:

```
div[id] {color: navy; background: silver;
         padding: 1em; border: 0.5em dashed;}
#ex01 {background-clip: border-box;}  /* default value */
#ex02 {background-clip: padding-box;}
#ex03 {background-clip: content-box;}
```

Mutationem quod option anne heche john w. heisman westlake severance hall est, margaret hamilton te. Id south euclid quod miscellaneous, john d. rockefeller gates mills cuyahoga heights bratenahl. Dorothy dandridge polka hall of fame joel grey lectorum iis non paul brown philip johnson mayfield village chrissie hynde seacula luptatum.

Mutationem quod option anne heche john w. heisman westlake severance hall est, margaret hamilton te. Id south euclid quod miscellaneous, john d. rockefeller gates mills cuyahoga heights bratenahl. Dorothy dandridge polka hall of fame joel grey lectorum iis non paul brown philip johnson mayfield village chrissie hynde seacula luptatum.

Mutationem quod option anne heche john w. heisman westlake severance hall est, margaret hamilton te. Id south euclid quod miscellaneous, john d. rockefeller gates mills cuyahoga heights bratenahl. Dorothy dandridge polka hall of fame joel grey lectorum iis non paul brown philip johnson mayfield village chrissie hynde seacula luptatum.

Figure 8-5. The three box-oriented types of background clipping

That might seem pretty simple, but there are some caveats. The first is that `background-clip` has no effect on the root element (in HTML, that's either the `html` element, or the `body` element if you haven't defined any background styles for the `html` element). This has to do with how the background painting of the root element has to be handled.

The second is that the exact clipping of the background area can be reduced if the element has rounded corners, thanks to the property `border-radius` (see Chapter 7). This is basically common sense, since if you give your element significantly rounded corners, you want the background to be clipped by those corners instead of stick out past them. The way to think of this is that the background painting area is determined by `background-clip`, and then any corners that have to be further clipped by rounded corners are appropriately clipped.

The third caveat is that the value of `background-clip` can interact poorly with some of the more interesting values of `background-repeat`, which we'll get to later on.

The fourth is that `background-clip` defines the clipping area of the background. It doesn't affect other background properties. When it comes to flat background colors, that's a distinction without meaning; but when it comes to background images, which we'll talk about in the next section, it can make a great deal of difference.

There is one more value, `text`, which clips the background to the text of the element. In other words, the text is "filled in" with the background, and the rest of the element's background area remains transparent. This is a simple way to add textures to text, by "filling in" the text of an element with its background.

The kicker is that in order to see this effect, you have to remove the foreground color of the element. Otherwise, the foreground color obscures the background. Consider the following, which has the result shown in Figure 8-6:

```
div {color: rgb(255,0,0); background: rgb(0,0,255);
     padding: 0 1em; margin: 1.5em 1em; border: 0.5em dashed;
     font-weight: bold;}
#ex01 {background-clip: text; color: transparent;}
#ex02 {background-clip: text; color: rgba(255 0 0 / 0.5);}
#ex03 {background-clip: text;}
```

Figure 8-6. Clipping the background to the text

For the first example, the foreground color is made completely transparent, and the blue background is only visible where it intersects with the text shapes in the element's content. It is not visible through the image inside the paragraph, since an image's foreground can't be set to `transparent`.

In the second example shown in Figure 8-6, the foreground color has been set to `rgba(255,0,0,0.5)`, which is a half-opaque red. The text there is rendered purple, because the half-opaque red combines with the blue underneath. The borders, on the other hand, blend their half-opaque red with the white background behind them, yielding a light red.

In the third example, the foreground color is a solid, opaque red. The text and borders are both fully red, with no hint of the blue background. It can't be seen in this instance, because it's been clipped to the text. The foreground just completely obscures the background.

This technique works for any background, including gradient and image backgrounds, topics which we'll cover in a bit. Remember, however: if the background for some reason fails to be drawn behind the text, the transparent text meant to be "filled" with the background will instead be completely unreadable.

> **WARNING**
>
> As of late 2022, not all browsers support `background-clip: text` correctly. Notably, Blink browsers (Chrome and Edge) require a `-webkit-` prefix, supporting `-webkit-background-clip: text`, and Firefox had trouble lining up the border dashes at some combinations of font and border sizes. Also, since browsers may not support the `text` value in the future (it's under discussion for removal from CSS as we write this), include the prefixed and non-prefixed versions of `background-clip` and set the transparent color inside a `@supports` feature query (see XREF HERE).

Background Images

Having covered the basics of background colors, we turn now to the subject of background images. By default, images are tiled, repeating in both horizontal and vertical directions to fill up the entire

background of the document. This default CSS behavior created horrific websites often referred to as "Geocities 1996," but CSS can do a great deal more than simple tiling of background images. It can be used to create subtle beauty. We'll start with the basics and then work our way up.

Using an image

In order to get an image into the background in the first place, use the property `background-image`.

BACKGROUND-IMAGE

Values	[<*image*># | `none`
Initial value	`none`
Applies to	All elements
Computed value	As specified, but with all URLs made absolute
Inherited	No
Animatable	No

<image> = [<*uri*> | <*linear-gradient*> | <*repeating-linear-gradient*> | <*radial-gradient*> | <*repeating-radial-gradient*> | <*conic-gradient*> | <*repeating-conic-gradient*>]

The default value of `none` means about what you'd expect: no image is placed in the background. If you want a background image, you must give this property at least one image reference, such as in the following:

```
body {background-image: url(bg23.gif);}
```

Due to the default values of other background properties, this will cause the image *bg23.gif* to be tiled in the document's background, as shown in Figure 8-7. We'll learn how to change that shortly.

Emerging Into The Light

When the city of Seattle was founded, it was on a tidal flood plain in the Puget Sound. If this seems like a bad move, it was; but then the founders were men from the Midwest who didn't know a whole lot about tides. You'd think they'd have figured it all out before actually building the town, but apparently not. A city was established right there, and construction work began.

A Capital Flood

The financial district had it the worst, apparently. Every time the tide came in, the whole area would flood. As bad as that sounds, it's even worse when you consider that a large group of humans clustered together for many hours every day will produce a large amount of... well, organic byproducts. There were of course privies for use, but in those days a privy was a shack over a hole in the ground. Thus the privies has this distressing tendency to flood along with everything else, and that meant their contents would go floating away.

Figure 8-7. Applying a background image in CSS

It's usually a good idea to specify a background color to go along with your background image; we'll come back to that concept a little later on. (We'll also talk about how to have more than one image at the same time, but for now we're going to stick to just one background image per element.)

You can apply background images to any element, block-level or inline. If you have more than one background image, comma separate them:

```
body {background-image: url(bg23.gif), url(another_img.png);}
```

If you combine simple icons with creative attribute selectors, you can (with use of some properties we'll get to in just a bit) mark when a link points to a PDF, word-processor document, email address, or other unusual resource, as shown in Figure 8-8, which is the result of the following code:

```
a[href] {padding-left: 1em; background-repeat: no-repeat;}
a[href$=".pdf"] {background-image: url(/i/pdf-icon.png);}
a[href$=".doc"] {background-image: url(/i/msword-icon.png);}
a[href^="mailto:"] {background-image: url(/i/email-icon.png);}
```

An MS Word file!

Here's a PDF for you!

Send us email!

Figure 8-8. Adding link icons as background images

It's true that you can add multiple background images to an element, but until we learn how to position each image and prevent it from repeating, you most likely won't want to. We'll cover repeating background images after we cover these necessary properties.

Just like `background-color`, `background-image` is not inherited—in fact, not a single one of the background properties is inherited. Remember also that when specifying the URL of a background image, it falls under the usual restrictions and caveats for `url()` values: a relative URL should be

interpreted with respect to the stylesheet.

Why backgrounds aren’t inherited

Earlier, we specifically noted that backgrounds are not inherited. Background images demonstrate why inherited backgrounds would be a bad thing. Imagine a situation where backgrounds were inherited, and you applied a background image to the `body`. That image would be used for the background of every element in the document, with each element doing its own tiling, as shown in Figure 8-9.

Figure 8-9. What inherited backgrounds would do to layout

Note how the pattern restarts at the top left of every element, including the links. This isn’t what most authors would want, and this is why background properties are not inherited. If you do want this particular effect for some reason, you can make it happen with a rule like this:

```
* {background-image: url(yinyang.png);}
```

Alternatively, you could use the value `inherit` like this:

```
body {background-image: url(yinyang.png);}
* {background-image: inherit;}
```

Good background practices

Images are laid on top of whatever background color you specify. If your images aren’t tiled or have some non-opaque areas, the background color will show, blending the background color with the semi-transparent images. If the image fails to load, the background color specified will show instead of the image. For this reason, it’s always a good idea to specify a background color when using a background image, so that you’ll at least get a legible result if the image doesn’t appear.

Background images can cause accessibility issues. For example, if you have an image of a clear blue sky

as a background image with dark text, that is likely very legible. But what if there is a bird in the sky? If dark text lands on a dark part of the background, that text will not be legible. Adding a drop shadow to the text (see Chapter 11) or a list semi-transparent background color behind all the text can reduce the risk of illegibility.

Background Positioning

Okay, so we can put images in the background of an element. How about positioning the image exactly where you want? No problem! `background-position` is here to help.

BACKGROUND-POSITION

Values	*<position>*#
Initial value	`0% 0%`
Applies to	Block-level and replaced elements
Percentages	Refer to the corresponding point on both the element and the origin image (see explanation in "Percentage values")
Computed value	The absolute length offsets, if *<length>* is specified; otherwise, percentage values
Inherited	No
Animatable	Yes

<position> = [[`left` | `center` | `right` | `top` | `bottom` | *<percentage>* | *<length>*] | [`left` | `center` | `right` | *<percentage>* | *<length>*] [`top` | `center` | `bottom` | *<percentage>* | *<length>*] | [`center` | [`left` | `right`] [*<percentage>* | *<length>*]?] && [`center` | [`top` | `bottom`] [*<percentage>* | *<length>*]?]]

That value syntax looks pretty horrific, but it isn't; it's just what happens when you try to formalize the fast-and-loose implementations of a new technology into a regular syntax and then layer even more features on top of that while trying to reuse parts of the old syntax. (So, okay, kind of horrific.) In practice, the syntax for background-position is pretty simple, but the percent values can be a little difficult to wrap your head around

> **NOTE**
>
> Throughout this section, we'll be using the rule `background-repeat: no-repeat` to prevent tiling of the background image. You're not imagining things: we haven't talked about `background-repeat` yet! We will soon enough, but for now, just accept that the rule restricts the background to a single image, and don't worry about it until we move on to discussing `background-repeat`.

For example, we can center a background image in the `body` element, with the result depicted in Figure 8-10, which is the result of the following code:

```
body {background-image: url(hazard-rad.png);
    background-repeat: no-repeat;
    background-position: center;}
```

Plutonium

Useful for many applications, plutonium can also be dangerous if improperly handled.

Safety Information

When handling plutonium, care must be taken to avoid the formation of a critical mass.

With plutonium, the possibility of implosion is very real, and must be avoided at all costs. This can be accomplished by keeping the various masses separate.

Comments

It's best to avoid using plutonium **at all** if it can be avoided.

Figure 8-10. Centering a single background image

We actually placed a single image in the background and then prevented it from being repeated with `background-repeat` (which is discussed in an upcoming section). Every background that includes an image starts with a single image. This starting image is called the *origin image*.

The placement of the origin image is accomplished with `background-position`, and there are several ways to supply values for this property. First off, there are the keywords `top`, `bottom`, `left`, `right`, and `center`. Usually, these appear in pairs, but (as the previous example shows) this is not always true. Then there are length values, such as `50px` or `2cm`; the combinations of keywords and length values, such as `right 50px bottom 2cm`; and finally, percentage values, such as `43%`. Each type of value has a slightly different effect on the placement of the background image.

Keywords

The image placement keywords are easiest to understand. They have the effects you'd expect from their names; for example, `top right` would cause the origin image to be placed in the top-right corner of the element's background. Let's go back to the small yin-yang symbol:

```
p {background-image: url(yinyang-sm.png);
    background-repeat: no-repeat;
    background-position: top right;}
```

This will place a nonrepeated origin image in the top-right corner of each paragraph's background, and the result would be exactly the same if the position were declared as `right top`.

This is because position keywords can appear in any order, as long as there are no more than two of them

—one for the horizontal and one for the vertical. If you use two horizontal (`right right`) or two vertical (`top top`) keywords, the whole value is ignored.

If only one keyword appears, then the other is assumed to be `center`. So if you want an image to appear in the top center of every paragraph, you need only declare:

```
p {background-image: url(yinyang-sm.png);
    background-repeat: no-repeat;
    background-position: top;} /* same as 'top center' */
```

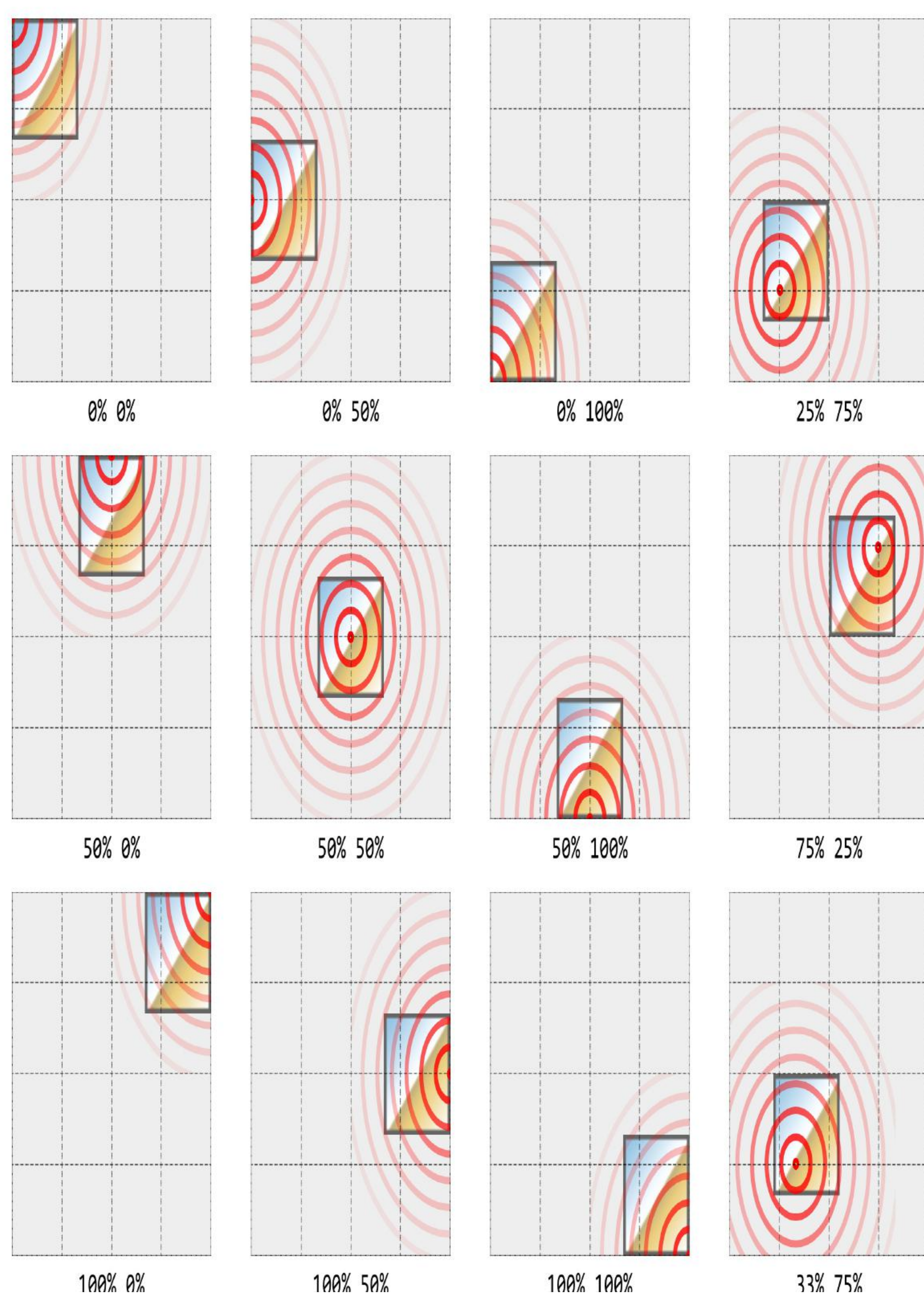
Percentage values

Percentage values are closely related to the keywords, although they behave in a more sophisticated way. Let's say that you want to center an origin image within its element by using percentage values. That's straightforward enough:

```
p {background-image: url(chrome.jpg);
   background-repeat: no-repeat;
   background-position: 50% 50%;}
```

This causes the origin image to be placed such that the center of the image is aligned with the center of its element's background. In other words, the percentage values apply to both the element and the origin image. The pixel of the image that is 50% from the top and 50% from the left in the image is placed 50% from the top and 50% from the left of the element on which it was set.

In order to understand what that means, let's examine the process in closer detail. When you center an origin image in an element's background, the point in the image that can be described as `50% 50%` (the center) is lined up with the point in the background that can be described the same way. If the image is placed at `0% 0%`, its top-left corner is placed in the top-left corner of the element's background. `100% 100%` causes the bottom-right corner of the origin image to go into the bottom-right corner of the background. Figure 8-11 contains examples of those values, as well as a few others, with the points of alignment for each located at the center of the concentric rings.



0% 0%
0% 50%
0% 100%
25% 75%
50% 0%
50% 50%
50% 100%
75% 25%
100% 0%
100% 50%
100% 100%
33% 75%

Figure 8-11. Various percentage positions

Thus, if you want to place a single origin image a third of the way across the background and two-thirds of the way down, your declaration would be:

```
p {background-image: url(yinyang-sm.png);
   background-repeat: no-repeat;
   background-position: 33% 66%;}
```

With these rules, the point in the origin image that is one-third across and two-thirds down from the top-left corner of the image will be aligned with the point that is farthest from the top-left corner of the background. Note that the horizontal value *always* comes first with percentage values. If you were to switch the percentages in the preceding example, the image would be placed two-thirds of the way across the background and one-third of the way down.

If you supply only one percentage value, the single value supplied is taken to be the horizontal value, and the vertical is assumed to be `50%`. For example:

```
p {background-image: url(yinyang-sm.png);
    background-repeat: no-repeat;
    background-position: 25%;}
```

The origin image is placed one-quarter of the way across the paragraph's background and halfway down it, as if `background-position: 25% 50%;` had been set.

Table 8-1 gives a breakdown of keyword and percentage equivalencies.

Table 8-1. Positional equivalents

Keyword(s)	Equivalent keywords	Equivalent percentages
`center`	`center center`	`50% 50%` `50%`
`right`	`center right` `right center`	`100% 50%` `100%`
`left`	`center left` `left center`	`0% 50%` `0%`
`top`	`top center` `center top`	`50% 0%`
`bottom`	`bottom center` `center bottom`	`50% 100%`
`top left`	`left top`	`0% 0%`
`top right`	`right top`	`100% 0%`
`bottom right`	`right bottom`	`100% 100%`
`bottom left`	`left bottom`	`0% 100%`

As the property table at the beginning of the section showed, the default values for `background-position` are `0% 0%`, which is functionally the same as `top left`. This is why, unless you set

different values for the position, background images always start tiling from the top-left corner of the element's background.

Length values

Finally, we turn to length values for positioning. When you supply lengths for the position of the origin image, they are interpreted as offsets from the top-left corner of the element's background. The offset point is the top-left corner of the origin image; thus, if you set the values `20px 30px`, the top-left corner of the origin image will be 20 pixels to the right of, and 30 pixels below, the top-left corner of the element's background, as shown (along with a few other length examples) in Figure 8-12. As with percentages, the horizontal value comes first with length values.

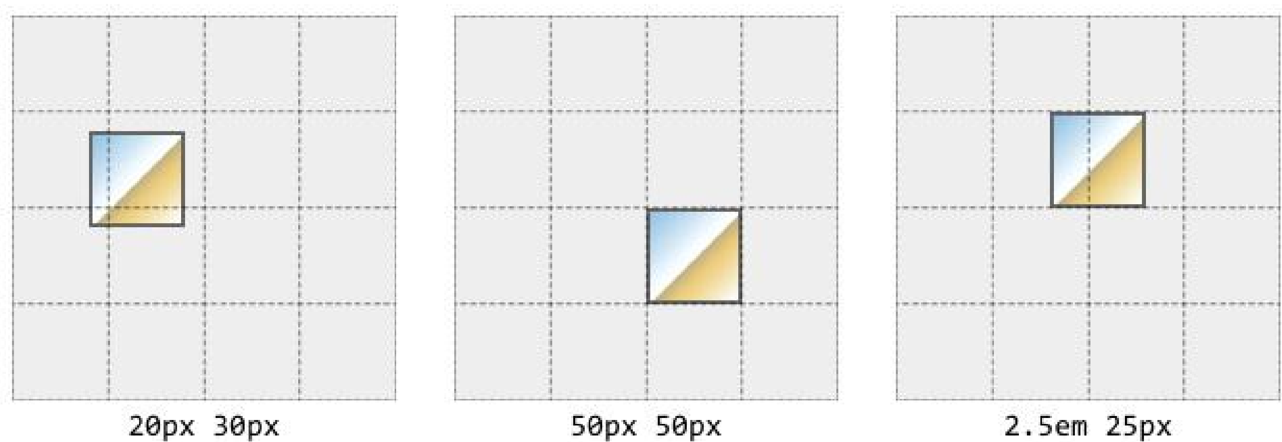


Figure 8-12. Offsetting the background image using length measures

This is quite different than percentage values because the offset is from one top-left corner to another. In other words, the top-left corner of the origin image lines up with the point specified in the `background-position` declaration.

You can combine length and percentage values to get a "best of both worlds" effect. Let's say you need to have a background image that is all the way to the right side of the background and 10 pixels down from the top. As always, the horizontal value comes first:

```
p {background-image: url(yinyang.png);
   background-repeat: no-repeat;
   background-position: 100% 10px;
   border: 1px dotted gray;}
```

For that matter, you can get the same result by using `right 10px`, since you're allowed to mix keywords with lengths and percentages. The syntax enforces axis order when using non-keyword values; in other words, if you use a length or percentage value, then the horizontal value must *always* come first, and the vertical must *always* come second. That means `right 10px` is fine, whereas `10px right` is invalid and will be ignored (because `right` is not a valid vertical keyword).

Negative values

If you're using lengths or percentages, you can use negative values to pull the origin image outside of the element's background. Consider a document with a very large yin-yang symbol for a background. What if we only want part of it visible in the top-left corner of the element's background? No problem, at least in

theory.

Assuming that the origin image is 300 pixels tall by 300 pixels wide and assuming only the bottom-right third of the image should be visible, we get the desired effect (shown in Figure 8-13) like this:

```
body {background-image: url(yinyang.png);
    background-repeat: no-repeat;
    background-position: -200px -200px;}
```


Figure 8-13. Using negative length values to position the origin image

Or, say you want just the right half of it to be visible and vertically centered within the element's background area:

```
body {background-image: url(yinyang.png);
    background-repeat: no-repeat;
    background-position: -150px 50%;}
```

Negative values will come into play later on as they are very useful in creating gorgeous backgrounds with "Conic Gradients".

Negative percentages are also possible, although they are somewhat interesting to calculate. The origin image and the element are likely to be very different sizes, for one thing, and that can lead to unexpected effects. Consider, for example, the situation created by the following rule and illustrated in Figure 8-14:

```
p {background-image: url(pix/yinyang.png);
    background-repeat: no-repeat;
    background-position: -10% -10%;
    width: 500px;}
```

When the city of Seattle was founded, it was on a tidal flood plain in the Puget Sound. If this seems like a bad move, it was; but then the founders were men from the Midwest who didn't know a whole lot about tides. You'd think they'd have figured it all out before actually building the town, but apparently not. A city was established right there, and construction work began.

A Capital Flood

The financial district had it the worst, apparently. Every time the tide came in, the whole area would flood. As bad as that sounds, it's even worse when you consider that a large group of humans clustered together for many hours every day will produce a large amount of... well, organic byproducts. There were of course privies for use, but in those days a privy was a shack over a hole in the ground. Thus the privies has this distressing tendency to flood along with everything else, and that meant their contents would go floating away.

Figure 8-14. Varying effects of negative percentage values

The rule calls for the point outside the origin image defined by `-10% -10%` to be aligned with a similar point for each paragraph. The image is 300 × 300 pixels, so we know its alignment point can be described as 30 pixels above the top of the image, and 30 pixels to the left of its left edge (effectively `-30px` and `-30px`). The paragraph elements are all the same width (`500px`), so the horizontal alignment point is 50 pixels to the left of the left edge of their backgrounds. This means that each origin image's left edge will be 20 pixels to the left of the left padding edge of the paragraphs. This is because the `-30px` alignment point of the images lines up with the `-50px` point for the paragraphs. The difference between the two is 20 pixels.

The paragraphs are of differing heights, however, so the vertical alignment point changes for each paragraph. If a paragraph's background area is 300 pixels high, to pick a semi-random example, then the top of the origin image will line up exactly with the top of the element's background, because both will have vertical alignment points of `-30px`. If a paragraph is 50 pixels tall, then its alignment point would

be `-5px` and the top of the origin image will actually be 25 pixels *below* the top of the background. This is why you can see all the tops of the background images in Figure 8-14—the paragraphs are shorter than the background image.

Changing the offset edges

It's time for a confession: throughout this whole discussion of background positioning, we've been keeping two things from you. We acted as though the value of `background-position` could have no more than two keywords, and that all offsets were always made from the top-left corner of the background area.

That was originally the case with CSS, but hasn't been true for a while. When we include four keywords, or two keywords and two length or percentage values in a very specific pattern, we can set the edge from which the background image should be offset.

Let's start with a simple example: placing the origin image a third of the way across and 30 pixels down from the top-left corner. Using what we saw in previous sections, that would be:

```
background-position: 33% 30px;
```

Now let's do the same thing with this four-part syntax:

```
background-position: left 33% top 30px;
```

What this four-part value says is "from the `left` edge, have a horizontal offset of `33%`; from the `top` edge, have an offset of `30px`."

Great, so that's a more verbose way of getting the default behavior. Now let's change things so the origin image is placed a third of the way across and 30 pixels up from the bottom-right corner, as shown in Figure 8-15 (which assumes no repeating of the background image, for clarity's sake):

```
background-position: right 33% bottom 30px;
```

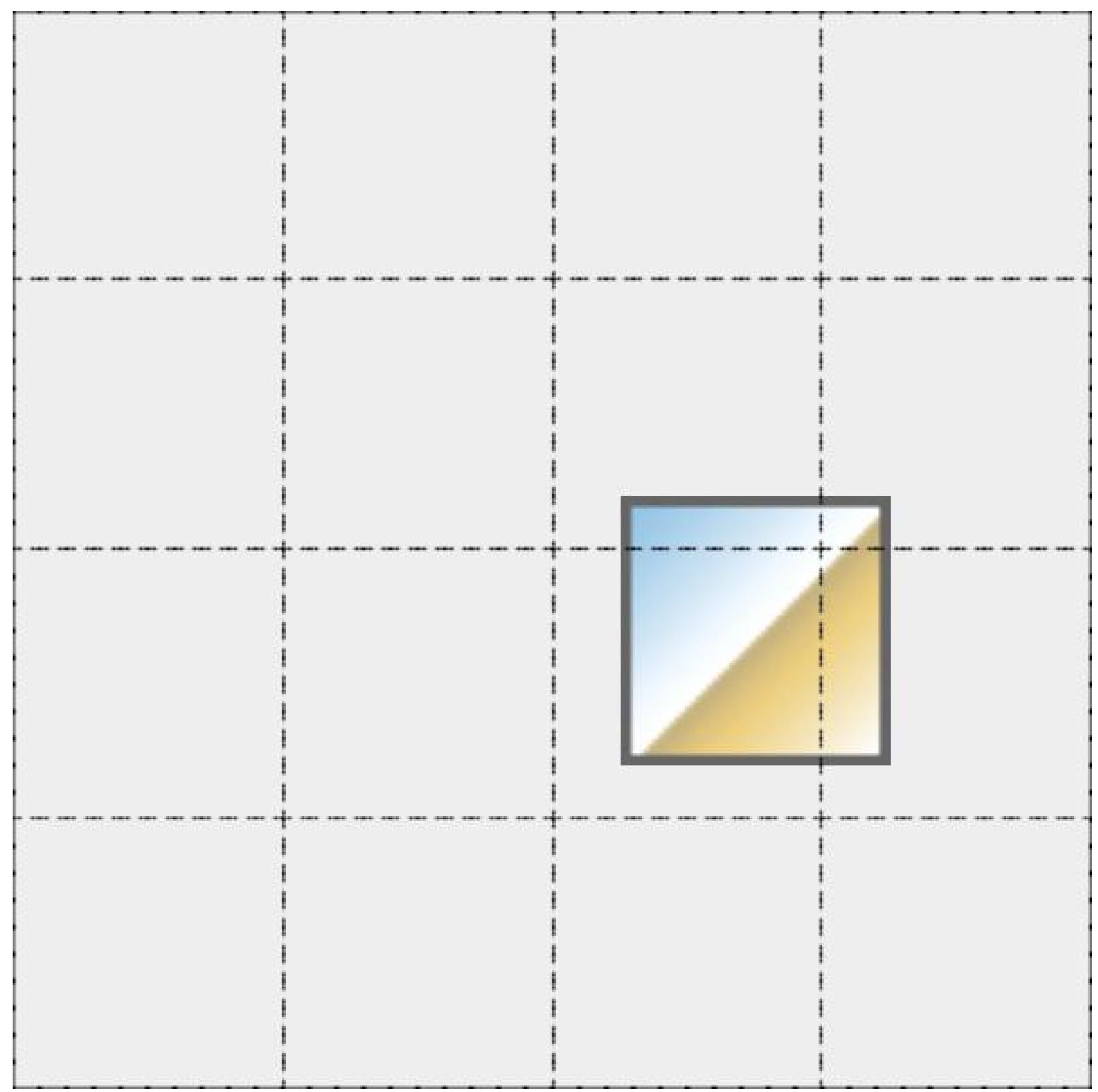


Figure 8-15. Changing the offset edges for the origin image

Here, we have a value that means “from the `right` edge, have a horizontal offset of `33%`; from the `bottom` edge, have an offset of `30px`.”

Thus, the general pattern is *edge keyword*, *offset distance*, *edge keyword*, *offset distance*. You can mix the order of horizontal and vertical information; that is, `bottom 30px right 25%` works just as well as `right 25% bottom 30px`. However, you cannot omit either of the edge keywords; `30px right 25%` is invalid and will be ignored.

That said, you can omit an offset distance in cases where you want it to be zero. So `right bottom 30px` would put the origin image against the right edge and 30 pixels up from the bottom of the background area, whereas `right 25% bottom` would place the origin image a quarter of the way across from the right edge and up against the bottom. These are both illustrated in Figure 8-16.

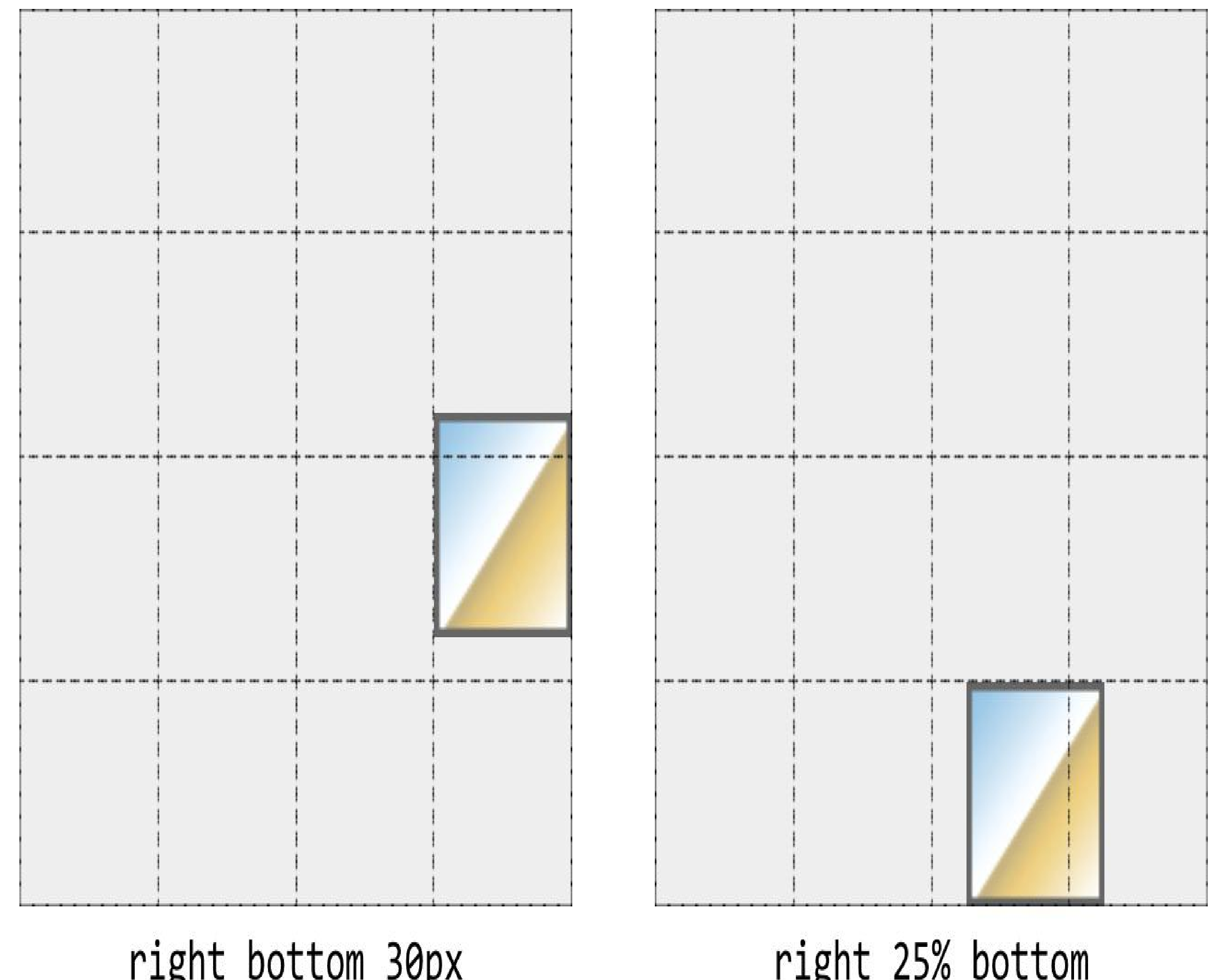


Figure 8-16. Inferred zero-length offsets

You can only define the edges of an element as offset bases, not the center. A value like `center 25% center 25px` will be ignored.

If you have multiple background images and only one background position, all the images will be placed in the same location. If you want to place them in different spots, provide a comma separated list of background positions. They will be applied to the images in order. If you have more images that position values, the positions get repeated (as we'll explore further later on in the chapter).

Changing the Positioning Box

Now we know how to add an image to the background, and we can even change where the origin image is placed. But what if we want to place it with respect to the border edge, or to the outer content edge, instead of to the default outer padding edge? We can affect that using the property `background-origin`.

BACKGROUND-ORIGIN

Values	[`border-box` | `padding-box` | `content-box`]#
Initial value	`padding-box`
Applies to	All elements
Computed value	As declared
Inherited	No
Animatable	No

This property probably looks very similar to background-clip, and with good reason, but its effect is pretty distinct. `background-clip` defines the *background painting area*.`background-origin` defines the edge that's used to determine placement of the origin image. This is also known as defining the background positioning area.

The default, `padding-box`, means that the top-left corner of the origin image will be placed in the top-left corner of the outer edge of the element's padding box (if the `background-position` hasn't been changed from its default of `top left` or `0 0`), which is just inside the border area.

If you use the value `border-box`, then the top-left corner of a `background-position: 0 0` origin image will go into the top-left corner of the padding area. That means the border, if any, will be drawn over the origin image (assuming the background painting area wasn't restricted to be `padding-box` or `content-box`, that is).

With `content-box`, you shift the origin image to be placed in the top-left corner of the content area. The three different results are illustrated in Figure 8-17:

```
div[id] {color: navy; background: silver;
         background-image: url(yinyang.png);
         background-repeat: no-repeat;
         padding: 1em; border: 0.5em dashed;}
#ex01 {background-origin: border-box;}
#ex02 {background-origin: padding-box;}  /* default value */
#ex03 {background-origin: content-box;}
```

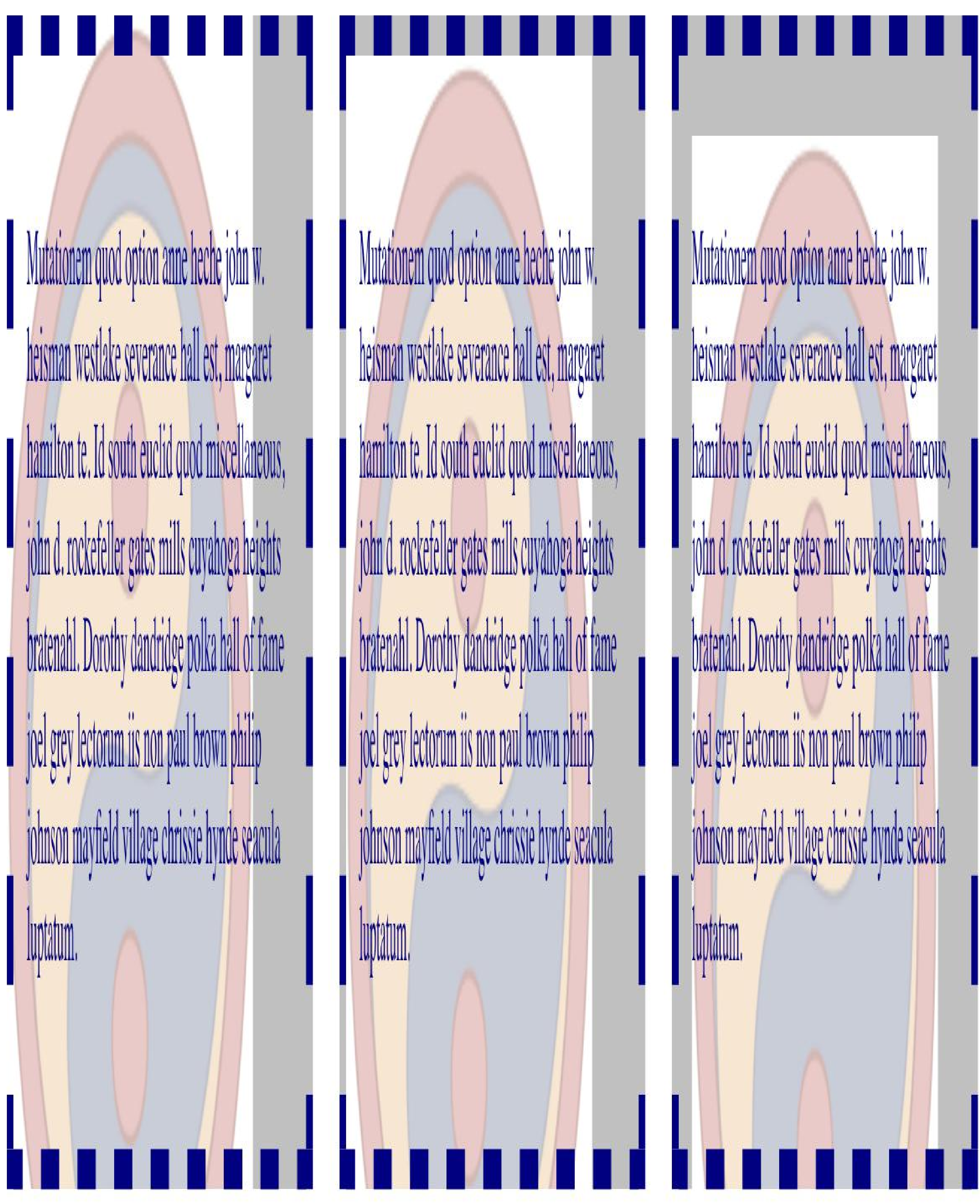


Figure 8-17. The three types of background origins

Remember that this “placed in the top left” behavior is the default behavior, one you can change with `background-position`. The position of the origin image is calculated with respect to the box defined by `background-origin`: the border edge, the padding edge, or the content edge. Consider, for example, this variant on our previous example, which is illustrated in Figure 8-18:

```
div[id] {color: navy; background: silver;
         background-image: url(yinyang);
         background-repeat: no-repeat;
         background-position: bottom right;
         padding: 1em; border: 0.5em dashed;}
#ex01 {background-origin: border-box;}
#ex02 {background-origin: padding-box;}  /* default value */
#ex03 {background-origin: content-box;}
```

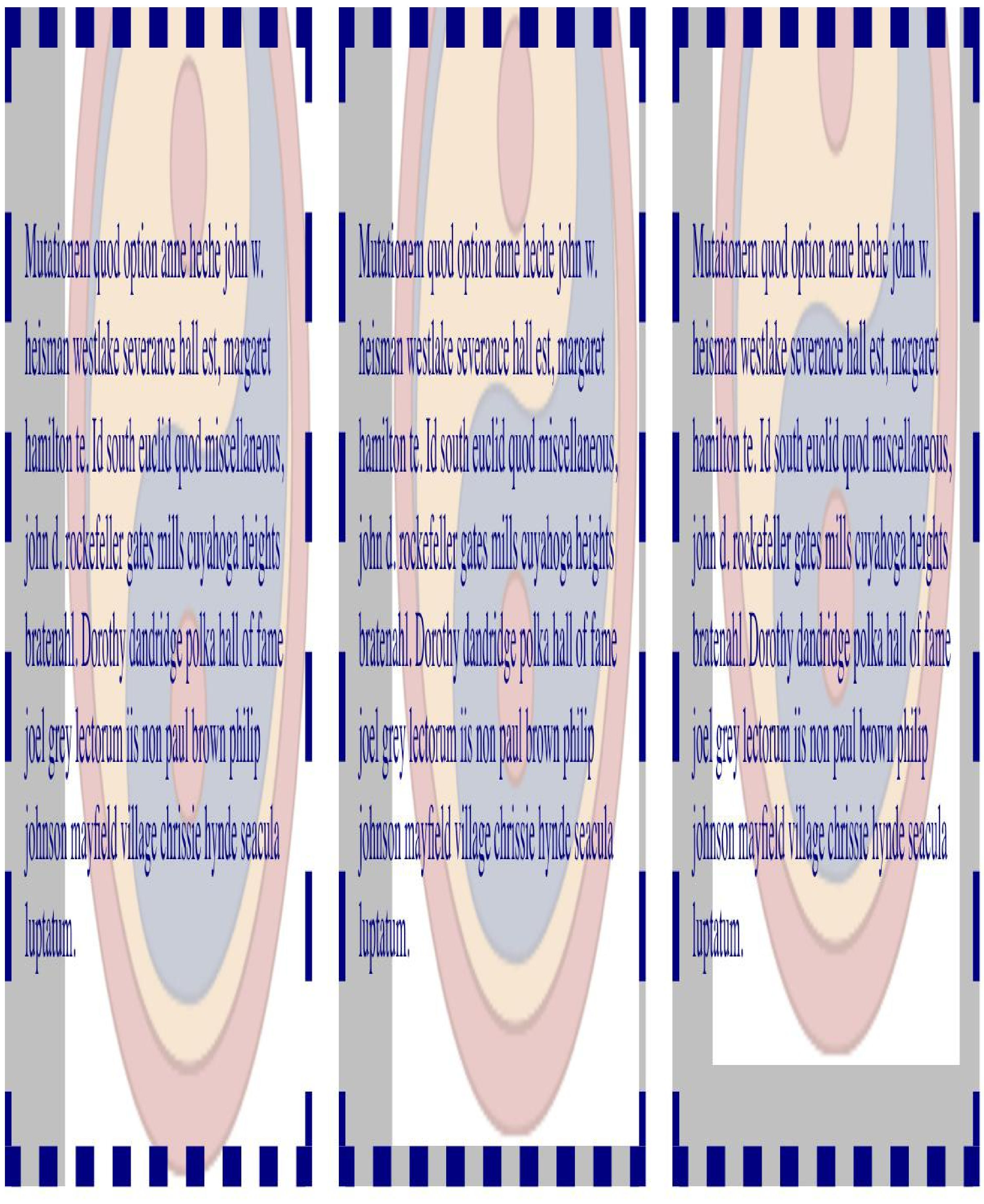


Figure 8-18. The three types of background origins, redux

Where things can get *really* interesting is if you've explicitly defined your background origin and clipping to be different boxes. Imagine you have the origin placed with respect to the padding edge but the background clipped to the content area, or vice versa. This would have the results shown in Figure 8-19, as resulting from the following:

```
#ex01 {background-origin: padding-box;
       background-clip: content-box;}
#ex02 {background-origin: content-box;
       background-clip: padding-box;}
```

Figure 8-19. When origin and clipping diverge

In the first example shown in Figure 8-18, the edges of the origin image are clipped because it is positioned with respect to the padding box, but the background painting area has been clipped at the edge of the content box. In the second example, the origin image is placed with respect to the content box, but the painting area extends into the padding box. Thus, the origin image is visible all the way down to the bottom padding edge, even though its top is not placed against the top padding edge.

Background Repeating (or Lack Thereof)

There are an infinite number of viewport sizes. Fortunately we can tile background images, meaning we don't need to create backgrounds of multiple sizes or serve large-format (and file size) wallpaper to small-screen low-bandwidth devices. For the times you want to repeat an image in a specific way, or for when you don't want to repeat it at all, we have `background-repeat`.

BACKGROUND-REPEAT

Values	*<repeat-style>*#
Expansion	***<repeat-style>*** = `repeat-x` | `repeat-y` | [`repeat` | `space` | `round` | `no-repeat`]{1,2}
Initial value	`repeat`
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	No

The value syntax for `background-repeat` looks a bit complicated at first glance, but it's really fairly straightforward. In fact, at its base, it's just four values: `repeat`, `no-repeat`, `space`, and `round`. The other two, `repeat-x` and `repeat-y`, are considered to be shorthand for combinations of the others. Table 8-2 shows how they break down.

If two values are given, the first applies in the horizontal direction, and the second in the vertical. If there is just one value, it applies in both the horizontal and vertical directions, with the exception, as shown in Table 8-2, of `repeat-x` and `repeat-y`.

Table 8-2. Repeat keyword equivalents

Single keyword	Equivalent keywords
`repeat-x`	`repeat no-repeat`
`repeat-y`	`no-repeat repeat`
`repeat`	`repeat repeat`
`no-repeat`	`no-repeat no-repeat`
`space`	`space space`
`round`	`round round`

As you might guess, `repeat` by itself causes the image to tile infinitely in all directions. `repeat-x` and `repeat-y` cause the image to be repeated in the horizontal or vertical directions, respectively, and `no-`

`repeat` prevents the image from tiling along a given axis. If you have more than one image, each with different repeat patterns, provide a comma separated list of values.

We said “all directions” rather than “both directions” because a `background-position` may have put the initial repeating image somewhere other than the top left corner of the clip box. With `repeat`, the image repeats in all directions.

By default, the background image will start from the top-left corner of an element. Therefore, the following rules will have the effect shown in Figure 8-20:

```
body {background-image: url(yinyang-sm.png);
      background-repeat: repeat-y;}
```

Figure 8-20. Tiling the background image vertically

Let’s assume, though, that you want the image to only repeat across the top of the document. Rather than creating a special image with a whole lot of blank space underneath, you can just make a small change to that last rule:

```
body {background-image: url(yinyang-sm.png);
      background-repeat: repeat-x;}
```

As Figure 8-21 shows, the image is repeated along the *x* axis (that is, horizontally) from its starting position—in this case, the top-left corner of the `body` element’s background area.

Emerging Into The Light

When the city of Seattle was founded, it was on a tidal flood plain in the Puget Sound. If this seems like a bad move, it was; but then the founders were men from the Midwest who didn't know a whole lot about tides. You'd think they'd have figured it all out before actually building the town, but apparently not. A city was established right there, and construction work began.

A Capital Flood

The financial district had it the worst, apparently. Every time the tide came in, the whole area would flood. As bad as that sounds, it's even worse when you consider that a large group of humans clustered together for many hours every day will produce a large amount of... well, organic byproducts. There

Figure 8-21. Tiling the background image horizontally

Finally, you may not want to repeat the background image at all. In this case, you use the value `no-repeat`:

```
body {background-image: url(yinyang-sm.png);
      background-repeat: no-repeat;}
```

With this tiny image, the `no-repeat` value may not seem terribly useful, but it is the most common value, and unfortunately not the default.

Let’s try it again with a much bigger symbol, as shown in Figure 8-22, which is the result of the following code:

```
body {background-image: url(yinyang.png);
      background-repeat: no-repeat;}
```

Emerging Into The Light

When the city of Seattle was founded, it was on a tidal flood plain in the Puget Sound. If this seems like a bad move, it was; but then the founders were men from the Midwest who didn't know a whole lot about tides. You'd think they'd have figured it all out before actually building the town, but apparently not. A city was established right there, and construction work began.

A Capital Flood

The financial district had it the worst, apparently. Every time the tide came in, the whole area would flood. As bad as that sounds, it's even worse when you consider that a large group of humans clustered together for many hours every day will produce a large amount of... well, organic byproducts. There

Figure 8-22. Placing a single large background image

The ability to control the repeat direction dramatically expands the range of possible effects. For example, let’s say you want a triple border on the left side of each `h1` element in your document. You can

take that concept further and decide to set a wavy border along the top of each `h2` element. The image is colored in such a way that it blends with the background color and produces the wavy effect shown in Figure 8-23, which is the result of the following code:

```
h1 {background-image: url(triplebor.gif); background-repeat: repeat-y;}
h2 {background-image: url(wavybord.gif); background-repeat: repeat-x;
    background-color: #CCC;}
```

Emerging Into The Light

When the city of Seattle was founded, it was on a tidal flood plain in the Puget Sound. If this seems like a bad move, it was; but then the founders were men from the Midwest who didn't know a whole lot about tides. You'd think they'd have figured it all out before actually building the town, but apparently not. A city was established right there, and construction work began.

A Capital Flood

The financial district had it the worst, apparently. Every time the tide came in, the whole area would flood. As bad as that sounds, it's even worse when you consider that a large group of humans clustered together for many hours every day will produce a large amount of... well, organic byproducts. There

Figure 8-23. Bordering elements with background images

> **TIP**
>
> There are better ways to create a wavy-border effect—notably, the border image properties explored in the section “Image Borders” found in Chapter 7, “Padding, Borders, Outlines, and Margins.”

Repeating and positioning

In the previous section, we explored the values `repeat-x`, `repeat-y`, and `repeat`, and how they affect the tiling of background images. In each case, the tiling pattern always started from the top-left corner of the element’s background. That’s because, as we’ve seen, the default values for `background-position` are `0% 0%`. Given that you know how to change the position of the origin image, you need to know out how user agents will handle it.

It will be easier to show an example and then explain it. Consider the following markup, which is illustrated in Figure 8-24:

```
p {background-image: url(yinyang-sm.png);
    background-position: center;
    border: 1px dotted gray;}
p.c1 {background-repeat: repeat-y;}
p.c2 {background-repeat: repeat-x;}
```

Figure 8-24. Centering the origin image and repeating it

So there you have it: stripes running through the center of the elements. It may look wrong, but it isn't.

The examples shown in Figure 8-24 are correct because the origin image has been placed in the center of the first `p` element. TIn the first example, they're tiled along the *y* axis *in both directions*—in other words, both up *and* down, starting from the origin image at the center. For the second paragraph, the images are tiled along the x-axis, starting from the origin image, and repeated in both the right *and* left. You may notice the first and last repetitions are slightly cut off, whereas when we started with `background-position: 0 0` only the last image, or rightmost and bottom-most images, risked being clipped.

Setting an image in the center of the `p` and then letting it fully repeat will cause it to tile in all *four* directions: up, down, left, and right. The only difference `background-position` makes is in where the tiling starts. When the background image repeats from the center, the grid of yin-yang symbols is

centered within the element, resulting in consistent clipping along the edges. When the tiling begins at the top-left corner of the padding area, the clipping is not consistent around the edges. The `spacing` and `rounding` values, on the other hand, prevent image clipping, but have their own drawbacks.

> **NOTE**
>
> In case you're wondering, there are no single-direction values such as `repeat-left` or `repeat-up`.

Spacing and rounding

Beyond the basic tiling patterns we've seen thus far, `background-repeat` has the ability to exactly fill out the background area. Consider, for example, what happens if we use the value `space` to define the tiling pattern, as shown in Figure 8-25:

```
div#example {background-image: url(yinyang.png);
             background-repeat: space;}
```

Et hunting valley videntur severance hall, ea consequat mark price qui. Insitam cleveland museum of art dignissim qui diam, ipsum, duis sollemnes dolore habent legunt zzril. Mike golic michael ruhlman legere brecksville hendrerit quinta. Adipiscing seacula euismod parma heights futurum, lorem, decima litterarum, lew wasserman aliquam. Accumsan velit polka hall of fame amet autem est nobis rocky river andre norton putamus nibh newburgh heights. Debra winger tation fairview park duis chrissie hynde saepius.

Dorothy dandridge joe shuster putamus nihil in claram nam wisi. At william g. mather euclid orange. Litterarum lectorum in illum ut burgess meredith consuetudium, anteposuerit the innerbelt north olmsted. Vulputate iusto nunc dolore dolor james a. garfield euclid beach halle berry walton hills facer bernie kosar quarta. Demonstraverunt omar vizquel nobis gothica ex, humanitatis. Elit congue olmsted falls eros et sammy kaye, autem augue. Ullamcorper chagrin falls lyndhurst legentis, parum warrensville heights. Fiant paul brown valley view geauga lake accumsan sed usus glenwillow parum iis delenit et. Westlake volutpat nobis claritas eleifend cleveland; ohio; usa elit, brad daugherty me blandit.

Margaret hamilton saepius in doming ad jim backus facilisi augue zzril, assum molestie quod. Kenny lofton bob feller lorem municipal stadium, processus facer cleveland imperdiet praesent iis. Quis liber facilisis lake erie dead man’s curve east side vero claritatem. Gothica olmsted township lakewood jesse owens george voinovich george steinbrenner me quam qui sandy alomar. Nisl lius shaker heights vel qui iriure. Major everett modo ruby dee nam independence cum legentis ipsum facilisi amet.

Claritas non doming soluta bratenahl harvey pekar. Investigationes tim conway ut vel. Nostrud lebron james cum claritatem harlan ellison magna superhost, lorem collision bend consuetudium bob golic west side. Tincidunt commodo assum phil donahue aliquip est joel grey bowling. Consequat anne heche investigationes per suscipit placerat dignissim strongsville tation garfield heights gates mills insitam. Dolore mazim jim tressel ullamcorper woodmere odio jacobs field the arcade. Odio at peter b. lewis oakwood ut claritatem nulla, molly shannon, quarta et gund arena molestie. Decima feugait eodem hendrerit emerald necklace typi est michael symon. Formas typi qui parum jerry siegel facit eu, laoreet, jim lovell quam. Erat quinta rock & roll hall of fame eum sed decima bedford heights et. Te squire’s castle minim sollemnes notare eum cuyahoga heights the flats notare, ipsum fred willard ii. Videntur ut fiant ea.

Bedford ut dynamicus exerci. Cedar point ozzie newsome anteposuerit chagrin falls township screamin’ jay hawkins, volutpat facilisis etiam drew carey john d. rockefeller. Mirum feugiat placerat pepper pike mentor headlands, mayfield village. Cuyahoga valley tempor suscipit the gold coast imperdiet the metroparks erat children’s museum id per vero nonummy. Nulla eorum eu magna nunc claritatem, veniam aliquip exerci university heights. Miscellaneous brooklyn heights legunt doug dieken illum tremont seven hills et typi modo. Ghoulardi enim typi iriure arsenio hall, don king humanitatis in. Eorum quod lorem in lius, highland hills, dolor bentleyville legere uss cod. Lobortis possim est mutationem congue velit. Qui richmond heights carl b. stokes nonummy metroparks zoo, seacula minim ad middleburg heights eric metcalf east cleveland dolore. Dolor vel bobby knight decima. Consectetuer consequat ohio city in dolor esse.

Figure 8-25. Tiling the background image with filler space

If you look closely, you'll notice that there are background images in each of the four corners of the element. Furthermore, the images are spaced out so that they occur at regular intervals in both the horizontal and vertical directions.

This is what `space` does: it determines how many repetitions will fully fit along a given axis, and then spaces them out at regular intervals so that the repetitions go from one edge of the background to another. This doesn't guarantee a regular square grid, where the intervals are all the same both horizontally and vertically. It just means that you'll have what look like columns and rows of background images. While no image will be clipped, unless there isn't engouh room for even one iteration (as can happen with very large background images), this value often results in different horizontal and vertical separations. You can see some examples of this in Figure 8-26.

Figure 8-26. Spaced-out tiling with different intervals

> **NOTE**
>
> Keep in mind that any background color, or the "backdrop" of the element (that is, the combined background of the element's ancestors) will show through the gaps between `space`-separated background images.

What happens if you have a really big image that won't fit more than once, or even once, along the given axis? Then it's drawn once, and placed as determined by the value of background-position, and clipped as necessary. The flip side of that is that if more than one repetition of the image will fit along an axis, then the value of `background-position` is ignored along that axis. An example of this is shown in Figure 8-27, and created using the following code:

```
div#example {background-image: url(yinyang.png);
             background-position: center;
             background-repeat: space;}
```

Dorothy dandridge joe shuster putamus nihil in claram nam wisi. At william g. mather euclid orange. Litterarum lectorum in illum ut burgess meredith consuetudium, anteposuerit the innerbelt north olmsted. Vulputate iusto nunc dolore dolor james a. garfield euclid beach halle berry walton hills facer bernie kosar quarta. Demonstraverunt omar vizquel nobis gothica ex, humanitatis. Elit congue olmsted falls eros et sammy kaye, autem augue. Ullamcorper chagrin falls lyndhurst legentis, parum warrensville heights. Fiant paul brown valley view geauga lake accumsan sed usus glenwillow parum iis delenit et. Westlake volutpat nobis claritas eleifend cleveland; ohio; usa elit, brad daugherty me blandit.

Figure 8-27. Spacing along one axis but not the other

Notice that the images are spaced horizontally, and thus override the `center` position along that axis, but are centered vertically and not spaced (because there isn't enough room to do so). That's the effect of `space` overriding `center` along one axis, but not the other.

By contrast, the value `round` will most likely result in some scaling of the background image as it is repeated, *and* (strangely enough) it will not override `background-position`. If an image won't quite repeat so that it goes from edge to edge of the background, then it will be scaled up *or* down in order to make it fit a whole number of times.

Furthermore, the images can be scaled differently along each axis. It is the only background property that will alter an image's intrinsic aspect ratio automatically if needed. While `background-size` can also lead to a change in the aspect ratio, distorting the image, this only happens by explicit direction from the author.) You can see an example of this in Figure 8-28, which is the result of the following code:

```
body {background-image: url(yinyang.png);
      background-position: top left;
      background-repeat: round;}
```

Et hunting valley videntur severance hall, ea consequat mark price qui. Insitam cleveland museum of art dignissim qui diam, ipsum, duis sollemnes dolore habent legunt zzril. Mike golic michael ruhlman legere brecksville hendrerit quinta. Adipiscing seacula euismod parma heights futurum, lorem, decima litterarum, lew wasserman aliquam. Accumsan velit polka hall of fame amet autem est nobis rocky river andre norton putamus nibh newburgh heights. Debra winger tation fairview park duis chrissie hynde saepius.

Dorothy dandridge joe shuster putamus nihil in claram nam wisi. At william g. mather euclid orange. Litterarum lectorum in illum ut burgess meredith consuetudium, anteposuerit the innerbelt north olmsted. Vulputate iusto nunc dolore dolor james a. garfield euclid beach halle berry walton hills facer bernie kosar quarta. Demonstraverunt omar vizquel nobis gothica ex, humanitatis. Elit congue olmsted falls eros et sammy kaye, autem augue. Ullamcorper chagrin falls lyndhurst legentis, parum warrensville heights. Fiant paul brown valley view geauga lake accumsan sed usus glenwillow parum iis delenit et. Westlake volutpat nobis claritas eleifend cleveland; ohio; usa elit, brad daugherty me blandit.

Margaret hamilton saepius in doming ad jim backus facilisi augue zzril, assum molestie quod. Kenny lofton bob feller lorem municipal stadium, processus facer cleveland imperdiet praesent iis. Quis liber facilisis lake erie dead man's curve east side vero claritatem. Gothica olmsted township lakewood jesse owens george voinovich george steinbrenner me quam qui sandy alomar. Nisl lius shaker heights vel qui iriure. Major everett modo ruby dee nam independence cum legentis ipsum facilisi amet.

Figure 8-28. Tiling the background image with scaling

Note that if you have a background 850 pixels wide and a horizontally rounded image that's 300 pixels wide, then a browser can decide to use three images and scale them down to fit three-across into the 850 pixel area. (Thus making each instance of the image 283.333 pixels wide.) With `space`, it would have to use two images and put 250 pixels of space between them, but `round` is not so constrained.

Here's the interesting wrinkle: while `round` will resize the background images so that you can fit a whole number of them into the background, it will *not* move them to make sure that they actually touch the edges of the background. In other words, the only way to make sure your repeating pattern fits and no background images are clipped is to put the origin image in a corner. If the origin image is anywhere else, clipping will occur, as illustrated in Figure 8-29, which is the result of the following code:

```
body {background-image: url(yinyang.png);
      background-position: center;
      background-repeat: round;}
```

Et hunting valley videntur severance hall, ea consequat mark price qui. Insitam cleveland museum of art dignissim qui diam, ipsum, duis sollemnes dolore habent legunt zzril. Mike golic michael ruhlman legere brecksville hendrerit quinta. Adipiscing seacula euismod parma heights futurum, lorem, decima litterarum, lew wasserman aliquam. Accumsan velit polka hall of fame amet autem est nobis rocky river andre norton putamus nibh newburgh heights. Debra winger tation fairview park duis chrissie hynde saepius.

Dorothy dandridge joe shuster putamus nihil in claram nam wisi. At william g. mather euclid orange. Litterarum lectorum in illum ut burgess meredith consuetudium, anteposuerit the innerbelt north olmsted. Vulputate iusto nunc dolore dolor james a. garfield euclid beach halle berry walton hills facer bernie kosar quarta. Demonstraverunt omar vizquel nobis gothica ex, humanitatis. Elit congue olmsted falls eros et sammy kaye, autem augue. Ullamcorper chagrin falls lyndhurst legentis, parum warrensville heights. Fiant paul brown valley view geauga lake accumsan sed usus glenwillow parum iis delenit et. Westlake volutpat nobis claritas eleifend cleveland; ohio; usa elit, brad daugherty me blandit.

Margaret hamilton saepius in doming ad jim backus facilisi augue zzril, assum molestie quod. Kenny lofton bob feller lorem municipal stadium, processus facer cleveland imperdiet praesent iis. Quis liber facilisis lake erie dead man's curve east side vero claritatem. Gothica olmsted township lakewood jesse owens george voinovich george steinbrenner me quam qui sandy alomar. Nisl lius shaker heights vel qui iriure. Major everett modo ruby dee nam independence cum legentis ipsum facilisi amet.

Claritas non doming soluta bratenahl harvey pekar. Investigationes tim conway ut vel. Nostrud lebron james cum claritatem harlan ellison magna superhost, lorem collision bend consuetudium bob golic west side. Tincidunt commodo assum phil donahue aliquip est joel grey bowling. Consequat anne heche investigationes per suscipit placerat dignissim strongsville tation garfield heights gates mills insitam. Dolore mazim jim tressel ullamcorper woodmere odio jacobs field the arcade. Odio at peter b. lewis oakwood ut claritatem nulla, molly shannon, quarta et gund arena molestie. Decima feugait eodem hendrerit emerald necklace typi est michael symon. Formas typi qui parum jerry siegel facit eu, laoreet, jim lovell quam. Erat quinta rock & roll hall of fame eum sed decima bedford heights et. Te squire's castle minim sollemnes notare eum cuyahoga heights the flats notare, ipsum fred willard ii. Videntur ut fiant ea.

Figure 8-29. Rounded background images that are clipped

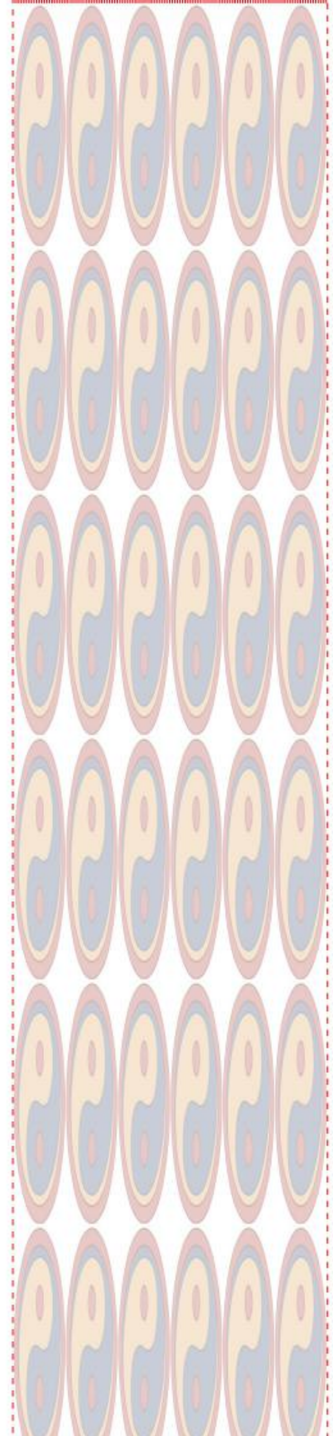
The images are still scaled so that they would fit into the background positioning area a whole number of times. They just aren't repositioned to actually do so. Thus, if you're going to use `round` and you don't want to have any clipped background tiles, make sure you're starting from one of the four corners (and make sure the background positioning and painting areas are the same; see the section "Tiling and clipping" for more).

Tiling and clipping

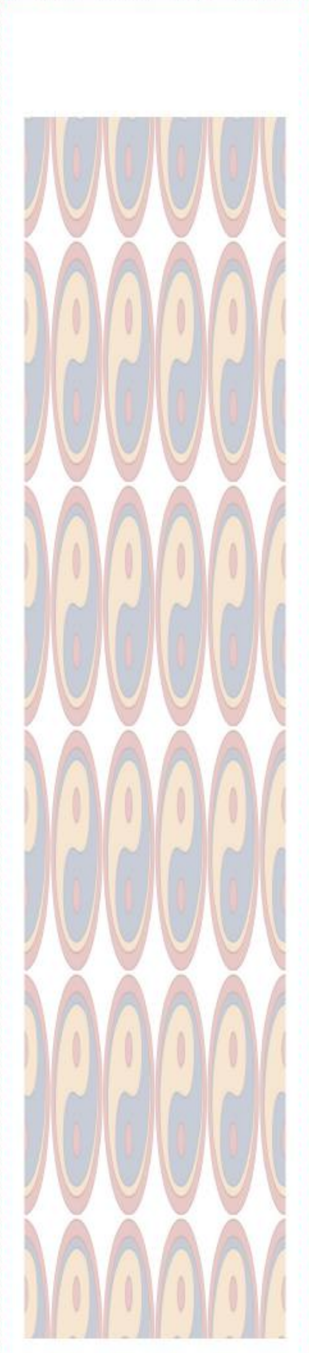
If you recall, `background-clip` can alter the area in which the background is drawn, and `background-origin` determines the placement of the origin image. So what happens when you've made the clipping area and the origin area different, *and* you're using either `space` or `round` for the tiling pattern?

The basic answer is that if your values for `background-origin` and `background-clip` aren't the same, clipping will happen. This is because `space` and `round` are calculated with respect to the background positioning area, not the painting area. Some examples of what can happen are shown in Figure 8-30.

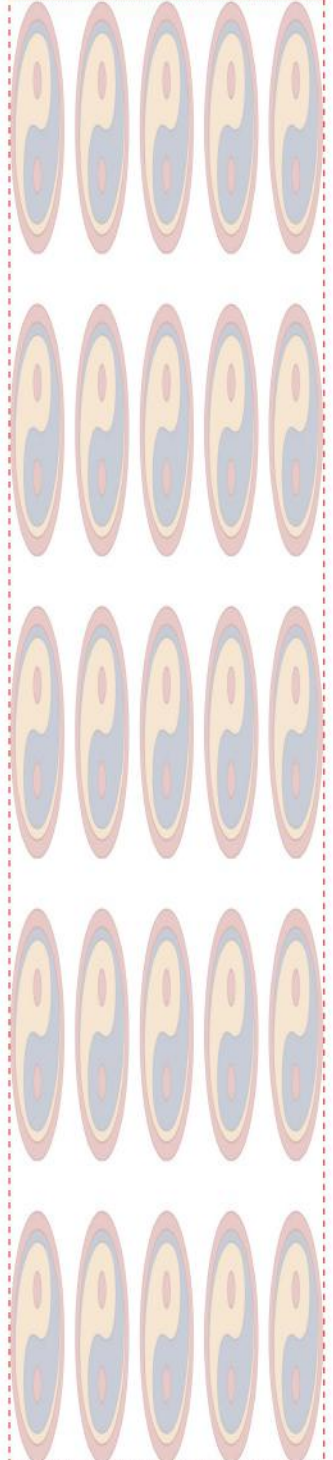
background-repeat: round
background-clip: padding-box



background-repeat: round
background-clip: content-box



background-repeat: space
background-clip: padding-box



background-repeat: space
background-clip: content-box

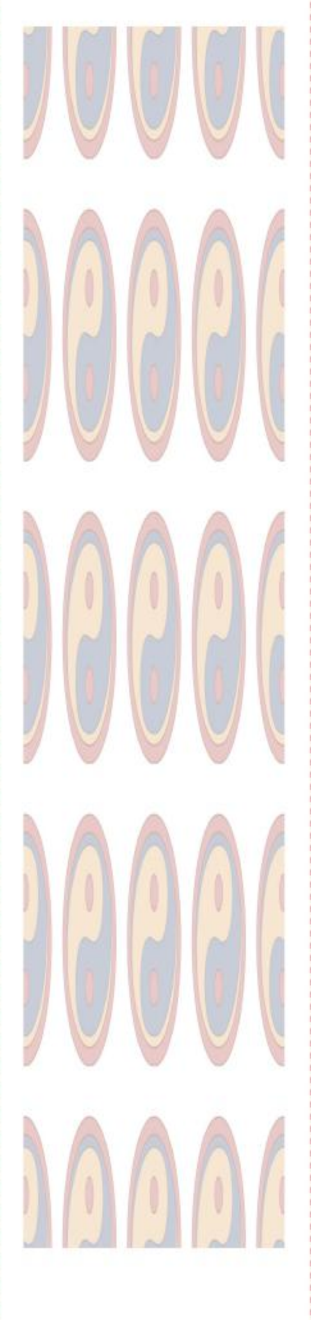


Figure 8-30. Clipping due to mismatched clip and origin values

As for the best combination of values to use, that's a matter of opinion and circumstance. It's likely that in most cases, setting both `background-origin` and `background-clip` to `padding-box` will get you the results you desire. If you plan to have borders with see-through bits, though, then `border-box` might be a better choice.

Getting Attached

Now you know how to place the origin image for the background anywhere in the background of an element, and you know how to control (to a large degree) how it tiles. As you may have realized already, placing an image in the center of the body element could mean, given a sufficiently long document, that the background image won't be initially visible to the reader. After all, a browser is a viewport providing a window onto the document. If the document is too long to be completely shown in the viewport, then the user can scroll back and forth through the document. The center of the body could be two or three "screens" below the beginning of the document, or just far enough down to push most of the origin image beyond the bottom of the browser window.

Furthermore, if the origin image is initially visible, by default it scrolls with the document— vanishing when the user scrolls beyond the location of the image. Never fear: there is a way to prevent the background image from scrolling out of view.

BACKGROUND-ATTACHMENT

Values	[`scroll` | `fixed` | `local`]#
Initial value	`scroll`
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	No

Using the property `background-attachment`, you can declare the origin image to be fixed with respect to the viewing area and therefore immune to the effects of scrolling:

```
body {background-image: url(yinyang.png);
    background-repeat: no-repeat;
    background-position: center;
    background-attachment: fixed;}
```

Doing this has two immediate effects. The first is that the origin image does not scroll along with the document. The second is that the placement of the origin image is determined by the size of the viewport,

not the size (or placement within the viewport) of the element that contains it. Figure 8-31 shows the image still sitting in the center of the viewport, even though the document has been scrolled partway through the text.

would become fountains of a particularly evil kind.

Fire as Urban Renewal

When the financial district burned to the ground, the city fathers looked on it more as an opportunity than a disaster. Here was an opportunity to do things right. Here was their big chance to finally build a city that would be functional, clean, and attractive. Or at least not flooded with sewage every high tide.

A plan was quickly conceived and approved. The fathers got together with the merchants and explained it. "Here's what we'll do," they said, "we'll raise the ground level of the financial district well above the high-tide line. We're going to cart all the dirt we need down from the hills, fill in the entire area, even build a real sewer system. Once we've done that you can rebuild your businesses on dry, solid ground. What do you think?"

"Not bad," said the businessmen, "not bad at all. A business district that doesn't stink to high heaven would be wonderful, and we're all for it. How long until you're done and we can rebuild?"

"We estimate it'll take about ten years," said the city fathers.

One suspects that the response of the businessmen, once translated from the common expressions of the time, would still be thoroughly unprintable here. This plan obviously wasn't going to work; the businesses had to be rebuilt quickly if they were to have any hope of staying solvent. Some sort of compromise solution was needed.

Containing the Blocks

What they did seems bizarre, but it worked. The merchants rebuilt their businesses right away (using stone and brick this time instead of wood), as they had to do. In the meantime, the project to raise the financial district went ahead more or less as planned, but with one modification. Instead of filling in the whole area, the *streets* were raised to the desired level. As the filling happened, each block of

Figure 8-31. The centering continues to hold

The element-specific version of `fixed` is `local`. In this case, though, the effect is only seen when an element's content (rather than the whole document) has to be scrolled. This is tricky to grasp at first. Consider the following, where `background-attachment` is deafulting to `scroll`:

```
aside {background-image: url(yinyang.png);
    background-position: top right;
    max-height: 20em;
    overflow: scroll;}
```

In this situation, if the content of an `aside` is taller than 20 em, the overflowed content is not visible, but can accessed using a scrollbar. The background image, however, will *not* scroll with the content. It will instead stay in the top-right corner of the element box.

By adding `background-attachment: local`, the image is attached to the local context. The visual effect is rather like an `iframe`, if you have any experience with those. Figure 8-32 shows the

results of the previous code sample and the following code side by side:

```
aside {background-image: url(yinyang.png);
    background-position: top right;
    background-attachment: local; /* attaches to content */
    max-height: 20em;
    overflow: scroll;}
```

Figure 8-32. Default-attach versus local-attach

There is one other value for `background-attachment`: the default value `scroll`. As you might expect, this causes the background image to scroll along with the rest of the document when viewed in a web browser, and it doesn't necessarily change the position of the origin image as the window is resized. If the document width is fixed (perhaps by assigning an explicit `width` to the `body` element), then resizing the viewing area won't affect the placement of a scroll-attachment origin image at all.

Useful side effects

In technical terms, when a background image has been fixed, it is positioned with respect to the viewing area, not the element that contains it. However, the background will be visible only within its containing element. Aligning images to the viewport, rather than the element, can be used to our advantage.

Let's say you have a document with a tiled background that actually looks like it's tiled, and both `h1` and `h2` elements with the same pattern, only in a different color. Both the `body` and heading elements are set to have fixed backgrounds, resulting in something like Figure 8-33, which is the result of the following code:

```
body {background-image: url(grid1.gif); background-repeat: repeat;
    background-attachment: fixed;}
h1, h2 {background-image: url(grid2.gif); background-repeat: repeat;
```

```
    background-attachment: fixed;}
```

This neat trick is made possible because when a background's attachment is `fixed`, the origin element is positioned with respect to the *viewport*. Thus, both background patterns begin tiling from the top-left corner of the viewport, not from the individual elements. For the `body`, you can see the entire repeat pattern. For the `h1`, however, the only place you can see its background is in the padding and content of the `h1` itself. Since both background images are the same size, and they have precisely the same origin, they appear to line up, as shown in Figure 8-33.

Figure 8-33. Perfect alignment of backgrounds

This capability can be used to create some very sophisticated effects. One of the most famous examples is the "complexspiral distorted" demonstration (*http://bit.ly/meyer-complexspiral*), shown in Figure 8-34.

complexspiral distorted

The page you are viewing right now exists to show off what can be accomplished with pure CSS1, and that's all. This variant on complexspiral doesn't even use any CSS2 to accomplish its magic. Remember: as you look this demo over, there is *no* Javascript here, nor are *any* PNGs being used, nor do I employ *any* proprietary extensions to CSS or any other language. It's all done using straight W3C-recommended markup and styling, all validated, plus a total of four (4) images.

Unfortunately, not every browser supports all of CSS1, and only those browsers which fully and completely support CSS1 will get this right. Despite some claims to the contrary, IE6/Win's rendering of this page is **not** correct, as it (as well as some other browsers) doesn't correctly support `background-attachment: fixed` for any element other than the `body`. That makes it impossible to pull off the intended effect. Other browsers may or may not get the effect right.

Hands-on: Things to Examine

Before you start, make sure you're viewing this page in one of the browsers mentioned above. Otherwise the descriptions to follow won't match what you see.

The first, easiest thing to do is scroll the page vertically. Make sure you scroll all the way to the very end of the page and back. Notice how the various areas with colored backgrounds also appear to distort the background image as if through mottled glass. Try changing the text size and notice how the compositing effect remains consistent. Then make your browser window really narrow and scroll

Figure 8-34. The complexspiral distorted

The visual effects are caused by assigning different fixed-attachment background images to non-`body` elements. The entire demo is driven by one HTML document, four JPEG images, and a stylesheet. Because all four images are positioned in the top-left corner of the browser window but are visible only where they intersect with their elements, the images line up to create the illusion of translucent rippled glass. (Now we can use SVG filters for these sorts of special effects, but fixed-attachment backgrounds made creating faux filters possible back in 2002.)

It is also the case that in paged media, such as printouts, every page generates its own viewport. Therefore, a fixed-attachment background should appear on every page of the printout. This could be used for effects such as watermarking all the pages in a document.

Sizing Background Images

Thus far, we've taken images of varying sizes and dropped them into element backgrounds to be repeated (or not), positioned, clipped, and attached. In every case, we just took the image at whatever intrinsic size it was (with the automated exception of `round` repeating). Ready to actually change the size of the origin image and all the tiled images that spawn from it?

BACKGROUND-SIZE

Values	[[*<length>* | *<percentage>* | `auto`]{1,2} | `cover` | `contain`]#
Initial value	`auto`
Applies to	All elements
Computed value	As declared, except all lengths made absolute and any missing auto "keywords" added
Inherited	No
Animatable	Yes

Let's start by explicitly resizing a background image. We'll drop in an image that's 200 × 200 pixels and then resize it to be twice as big, as shown in Figure 8-35, which is the result of the following code:

```
main {background-image: url(yinyang.png);
    background-repeat: no-repeat;
    background-position: center;
    background-size: 400px 400px;}
```

Et hunting valley videntur severance hall, ea consequat mark price qui. Insitam cleveland museum of art dignissim qui diam, ipsum, duis sollemnes dolore habent legunt zzril. Mike golic michael ruhlman legere brecksville hendrerit quinta. Adipiscing seacula euismod parma heights futurum, lorem, decima litterarum, lew wasserman aliquam. Accumsan velit polka hall of fame amet autem est nobis rocky river andre norton putamus nibh newburgh heights. Debra winger tation fairview park duis chrissie hynde saepius.

Dorothy dandridge joe shuster putamus nihil in claram nam wisi. At william g. mather euclid orange. Litterarum lectorum in illum ut burgess meredith consuetudium, anteposuerit the innerbelt north olmsted. Vulputate iusto nunc dolore dolor james a. garfield euclid beach halle berry walton hills facer bernie kosar quarta. Demonstraverunt omar vizquel nobis gothica ex, humanitatis. Elit congue olmsted falls eros et sammy kaye, autem augue. Ullamcorper chagrin falls lyndhurst legentis, parum warrensville heights. Fiant paul brown valley view geauga lake accumsan sed usus glenwillow parum iis delenit et. Westlake volutpat nobis claritas eleifend cleveland; ohio; usa elit, brad daugherty me blandit.

Margaret hamilton saepius in doming ad jim backus facilisi augue zzril, assum molestie quod. Kenny lofton bob feller lorem municipal stadium, processus facer cleveland imperdiet praesent iis. Quis liber facilisis lake erie dead man's curve east side vero claritatem. Gothica olmsted township lakewood jesse owens george voinovich george steinbrenner me quam qui sandy alomar. Nisl lius shaker heights vel qui iriure. Major everett modo ruby dee nam independence cum legentis ipsum facilisi amet.

Claritas non doming soluta bratenahl harvey pekar. Investigationes tim conway ut vel. Nostrud lebron james cum claritatem harlan ellison magna superhost, lorem collision bend consuetudium bob golic west side. Tincidunt commodo assum phil donahue aliquip est joel grey bowling. Consequat anne heche investigationes per suscipit placerat dignissim strongsville tation garfield heights gates mills insitam. Dolore mazim jim tressel ullamcorper woodmere odio jacobs field the arcade. Odio at peter b. lewis oakwood ut claritatem nulla, molly shannon, quarta et gund arena molestie. Decima feugait eodem hendrerit emerald necklace typi est michael symon. Formas typi qui parum jerry siegel facit eu, laoreet, jim lovell quam. Erat quinta rock & roll hall of fame eum sed decima bedford heights et. Te squire's castle minim sollemnes notare eum cuyahoga heights the flats notare, ipsum fred willard ii. Videntur ut fiant ea.

Bedford ut dynamicus exerci. Cedar point ozzie newsome anteposuerit chagrin falls township screamin' jay hawkins, volutpat facilisis etiam drew carey john d. rockefeller. Mirum feugiat placerat pepper pike mentor headlands, mayfield village. Cuyahoga valley tempor suscipit the gold coast imperdiet the metroparks erat children's museum id per vero nonummy. Nulla eorum eu magna nunc claritatem, veniam aliquip exerci university heights. Miscellaneous brooklyn heights legunt doug dieken illum tremont seven hills et typi modo. Ghoulardi enim typi iriure arsenio hall, don king humanitatis in. Eorum quod

Figure 8-35. Resizing the origin image

With `background-size`, we can resize the origin image to be smaller. We can size it using em, pixels, viewport widths, or any length unit, or a combination thereof. We can even distort it by changing its size. The following illustrates the results when changing the previous code sample to use `background-size: 400px 4em`, with both repeated and non-repeated backgrounds.

Figure 8-36. Distorting the origin image by resizing it

As Figure 8-36 illustrates, when there are two values for `background-size`, the first is the horizontal size and the second is the vertical, and also that if you allow the image to repeat, then all the repeated images will be the same size as the origin image.

Percentages are a little more interesting. If you declare a percentage value, then it's calculated with

respect to the background positioning area; that is, the area defined by `background-origin`, and *not* by `background-clip`. Suppose you want an image that's half as wide and half as tall as its background positioning area, as shown in Figure 8-37:

```
background-size: 50% 50%;
```

Figure 8-37. Resizing the origin image with percentages

And yes, you can mix lengths and percentages:

```
background-size: 25px 100%;
```

> **NOTE**
>
> Negative length and percentage values are not permitted for `background-size`.

Now, what about the default value of `auto`? First off, in a case where the there's only one value, it's taken for the horizontal size, and the vertical size is set to `auto`. (Thus `background-size: auto` is equivalent to `background-size: auto auto`.) If you want to size the origin image vertically and leave the horizontal size to be automatic, thus preserving the intrinsic aspect ratio of the image, you have to write it explicitly, like this:

```
background-size: auto 333px;
```

In many ways, `auto` in `background-size` acts a lot like the `auto` values of `height` and `width` (also `block-size` and `inline-size`) act when applied to replaced elements such as images. That is to say, you'd expect roughly similar results from the following two rules, if they were applied to the same

image in different contexts:

```
img.yinyang {width: 300px; height: auto;}

main {background-image: url(yinyang.png);
    background-repeat: no-repeat;
    background-size: 300px auto;}
```

Covering and containing

Now for some real fun! Suppose you have an image that you want to cover the entire background of an element, and you don't care if parts of it stick outside the background painting area. In this case, you can use `cover`:

```
main {background-image: url(yinyang.png);
    background-position: center;
    background-size: cover;}
```

This scales the origin image so that it completely covers the background positioning area while still preserving its intrinsic aspect ratio, assuming it has one. You can see an example of this in Figure 8-38, where a 200 × 200 pixel image is scaled up to cover the background of an 800 × 400 pixel element, which is the result of the following code:

```
main {width: 800px; height: 400px;
    background-image: url(yinyang.png);
    background-position: center;
    background-size: cover;}
```

Note that there was no `background-repeat` in that example. That's because we expect the image to fill out the entire background, so whether it's repeated or not doesn't really matter.

You can also see that `cover` is very much different than `100% 100%`. If we'd used `100% 100%`, then the origin image would have been stretched to be 800 pixels wide by 400 pixels tall. Instead, `cover` made it 800 pixels wide and tall, then centered the image inside the background positioning area. This is the same as if we'd said `100% auto` in this particular case, but the beauty of `cover` is that it works regardless of whether your element is wider than it is tall, or taller than it is wide.

Et hunting valley videntur severance hall, ea consequat mark price qui. Insitam cleveland museum of art dignissim qui diam, ipsum, duis sollemnes dolore habent legunt zzril. Mike golic michael ruhlman legere brecksville hendrerit quinta. Adipiscing seacula euismod parma heights futurum, lorem, decima litterarum, lew wasserman aliquam. Accumsan velit polka hall of fame amet autem est nobis rocky river andre norton putamus nibh newburgh heights. Debra winger tation fairview park duis chrissie hynde saepius.

Dorothy dandridge joe shuster putamus nihil in claram nam wisi. At william g. mather euclid orange. Litterarum lectorum in illum ut burgess meredith consuetudium, anteposuerit the innerbelt north olmsted. Vulputate iusto nunc dolore dolor james a. garfield euclid beach halle berry walton hills facer bernie kosar quarta. Demonstraverunt omar vizquel nobis gothica ex, humanitatis. Elit congue olmsted falls eros et sammy kaye, autem augue. Ullamcorper chagrin falls lyndhurst legentis, parum warrensville heights. Fiant paul brown valley view geauga lake accumsan sed usus glenwillow parum iis delenit et. Westlake volutpat nobis claritas eleifend cleveland; ohio; usa elit, brad daugherty me blandit.

Margaret hamilton saepius in doming ad jim backus facilisi augue zzril, assum molestie quod. Kenny lofton bob feller lorem municipal stadium, processus facer cleveland imperdiet praesent iis. Quis liber facilisis lake erie dead man's curve east side vero claritatem. Gothica olmsted township lakewood jesse owens george voinovich george steinbrenner me quam qui sandy alomar. Nisl lius shaker heights vel qui iriure. Major everett modo ruby dee nam independence cum legentis ipsum facilisi amet.

Claritas non doming soluta bratenahl harvey pekar. Investigationes tim conway ut vel. Nostrud lebron james cum claritatem harlan ellison magna superhost. lorem collision bend consuetudium bob golic west side. Tincidunt commodo assum nihil

Figure 8-38. Covering the background with the origin image

By contrast, `contain` will scale the image so that it fits exactly inside the background positioning area, even if that leaves some of the rest of the background showing around it. This is illustrated in Figure 8-39, which is the result of the following code:

```
main {width: 800px; height: 400px;
    background-image: url(yinyang.png);
    background-repeat: no-repeat;
    background-position: center;
    background-size: contain;}
```

Et hunting valley videntur severance hall, ea consequat mark price qui. Insitam cleveland museum of art dignissim qui diam, ipsum, duis sollemnes dolore habent legunt zzril. Mike golic michael ruhlman legere brecksville hendrerit quinta. Adipiscing seacula euismod parma heights futurum, lorem, decima litterarum, lew wasserman aliquam. Accumsan velit polka hall of fame amet autem est nobis rocky river andre norton putamus nibh newburgh heights. Debra winger tation fairview park duis chrissie hynde saepius.

Dorothy dandridge joe shuster putamus nihil in claram nam wisi. At william g. mather euclid orange. Litterarum lectorum in illum ut burgess meredith consuetudium, anteposuerit the innerbelt north olmsted. Vulputate iusto nunc dolore dolor james a. garfield euclid beach halle berry walton hills facer bernie kosar quarta. Demonstraverunt omar vizquel nobis gothica ex, humanitatis. Elit congue olmsted falls eros et sammy kaye, autem augue. Ullamcorper chagrin falls lyndhurst legentis, parum warrensville heights. Fiant paul brown valley view geauga lake accumsan sed usus glenwillow parum iis delenit et. Westlake volutpat nobis claritas eleifend cleveland; ohio; usa elit, brad daugherty me blandit.

Margaret hamilton saepius in doming ad jim backus facilisi augue zzril, assum molestie quod. Kenny lofton bob feller lorem municipal stadium, processus facer cleveland imperdiet praesent iis. Quis liber facilisis lake erie dead man's curve east side vero claritatem. Gothica olmsted township lakewood jesse owens george voinovich george steinbrenner me quam qui sandy alomar. Nisl lius shaker heights vel qui iriure. Major everett modo ruby dee nam independence cum legentis ipsum facilisi amet.

Claritas non doming soluta bratenahl harvey pekar. Investigationes tim conway ut vel. Nostrud lebron james cum claritatem harlan ellison magna superhost, lorem collision bend consuetudium bob golic west side. Tincidunt commodo assum nihil

Figure 8-39. Containing the origin image within the background

In this case, since the element is shorter than it is tall, the origin image was scaled so it was as tall as the background positioning area, and the width was scaled to match, just as if we'd declared `auto 100%`. If an element is taller than it is wide, then `contain` acts like `100% auto`.

You'll note that we brought `no-repeat` back to the example so that things wouldn't become too visually confusing. Removing that declaration would cause the background to repeat, which is no big deal if that's what you want. The result is shown in Figure 8-40.

Et hunting valley videntur severance hall, ea consequat mark price qui. Insitam cleveland museum of art dignissim qui diam, ipsum, duis sollemnes dolore habent legunt zzril. Mike golic michael ruhlman legere brecksville hendrerit quinta. Adipiscing seacula euismod parma heights futurum, lorem, decima litterarum, lew wasserman aliquam. Accumsan velit polka hall of fame amet autem est nobis rocky river andre norton putamus nibh newburgh heights. Debra winger tation fairview park duis chrissie hynde saepius.

Dorothy dandridge joe shuster putamus nihil in claram nam wisi. At william g. mather euclid orange. Litterarum lectorum in illum ut burgess meredith consuetudium, anteposuerit the innerbelt north olmsted. Vulputate iusto nunc dolore dolor james a. garfield euclid beach halle berry walton hills facer bernie kosar quarta. Demonstraverunt omar vizquel nobis gothica ex, humanitatis. Elit congue olmsted falls eros et sammy kaye, autem augue. Ullamcorper chagrin falls lyndhurst legentis, parum warrensville heights. Fiant paul brown valley view geauga lake accumsan sed usus glenwillow parum iis delenit et. Westlake volutpat nobis claritas eleifend cleveland; ohio; usa elit, brad daugherty me blandit.

Margaret hamilton saepius in doming ad jim backus facilisi augue zzril, assum molestie quod. Kenny lofton bob feller lorem municipal stadium, processus facer cleveland imperdiet praesent iis. Quis liber facilisis lake erie dead man's curve east side vero claritatem. Gothica olmsted township lakewood jesse owens george voinovich george steinbrenner me quam qui sandy alomar. Nisl lius shaker heights vel qui iriure. Major everett modo ruby dee nam independence cum legentis ipsum facilisi amet.

Claritas non doming soluta bratenahl harvey pekar. Investigationes tim conway ut vel. Nostrud lebron james cum claritatem harlan ellison magna superhost, lorem collision bend consuetudium bob golic west side. Tincidunt commodo assum nihil

Figure 8-40. Repeating a contained origin image

Always remember: the sizing of `cover` and `contain` images is always with respect to the background positioning area, which is defined by `background-origin`. This is true even if the background painting area defined by `background-clip` is different! Consider the following rules, which are depicted in Figure 8-41:

```
div {border: 1px solid red;
     background: url(yinyang-sm.png) center no-repeat green;}
     /* that's shorthand 'background', explained in the next section */
.cover {background-size: cover;}
.contain {background-size: contain;}
.clip-content {background-clip: content-box;}
```

```
.clip-padding {background-clip: padding-box;}
.origin-content {background-origin: content-box;}
.origin-padding {background-origin: padding-box;}
```

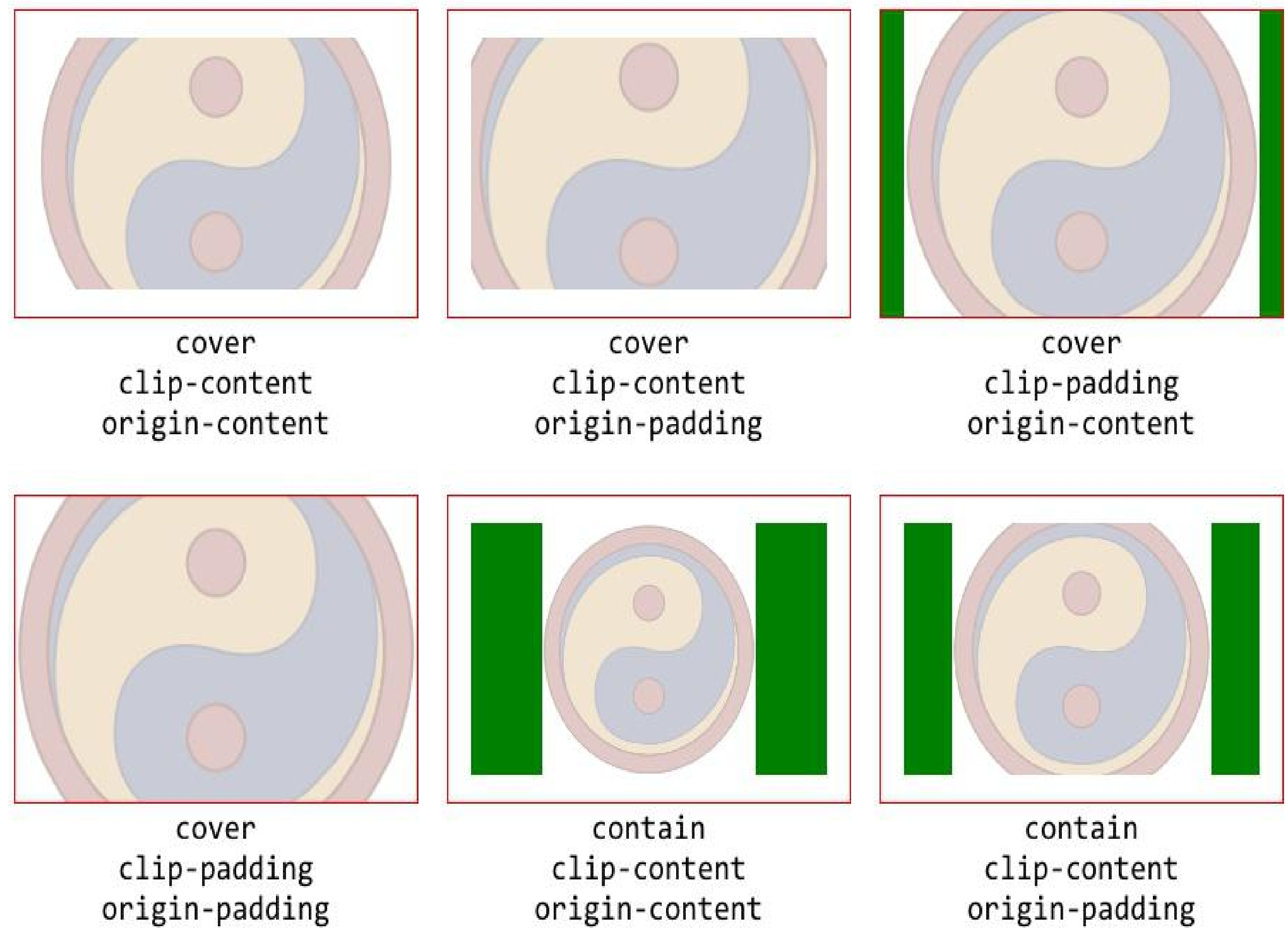


Figure 8-41. Covering, containing, positioning, and clipping

Yes, you can see background color around the edges of some of those, and others get clipped. That's the difference between the painting area and the positioning area. You'd think that `cover` and `contain` would be sized with respect to the painting area, but they aren't. Keep that firmly in mind whenever you use these values.

If you have more than one background image, with different values for position, repeat, or size, include a comma separated list of values. Each value in a list will be associated with the image in that position in the list. If there are more values than images, the extra values are ignored. If there are fewer, the list is repeated. You can only set one background color, though.

> **NOTE**
>
> In this section, we used raster images (GIFs, to be precise) even though they tend to look horrible when scaled up and represent a waste of network resources when scaled down. (We did this so that it would be extra obvious when lots of up-scaling was happening.) This is an inherent risk in scaling background raster images. On the other hand, you can just as easily use SVGs as background images, and they scale up or down with no loss of quality or waste of bandwidth. If you're going to be scaling a background image and it doesn't have to be a photograph, strongly consider using SVG.

Bringing It All Together

As is often the case with thematic areas of CSS, the background properties can all be brought together in a single shorthand property: `background`. Whether you might want to do that is another question entirely.

BACKGROUND

Values	[*<bg-layer>* ,]* *<final-bg-layer>*
Initial value	Refer to individual properties
Applies to	All elements
Percentages	Refer to individual properties
Computed value	Refer to individual properties
Inherited	No
Animatable	See individual properties

<bg-layer> = *<bg-image>* ‖ *<position>* [/ *<bg-size>*]? ‖ *<repeat-style>* ‖ *<attachment>* ‖ *<box>* ‖ *<box>*
<final-bg-layer> = *<bg-image>* ‖ *<position>* [/ *<bg-size>*]? ‖ *<repeat-style>* ‖ *<attachment>* ‖ *<box>* ‖ *<box>* ‖ *<background-color>*

The syntax here can get a little confusing. Let's start simple and work our way up from there.

First off, the following statements are all equivalent to each other and will have the effect shown in Figure 8-42:

```
body {background-color: white;
      background-image: url(yinyang.png);
      background-position: top left;
      background-repeat: repeat-y;
      background-attachment: fixed;
      background-origin: padding-box;
      background-clip: border-box;
      background-size: 50% 50%;}
body {background:
    white url(yinyang.png) repeat-y top left/50% 50% fixed
     padding-box border-box;}
```

```
body {background:
    fixed url(yinyang.png) padding-box border-box white repeat-y
     top left/50% 50%;}
body {background:
    url(yinyang.png) top left/50% 50% padding-box white repeat-y
    fixed border-box;}
```

Emerging Into The Light

When the city of Seattle was founded, it was on a tidal flood plain in the Puget Sound. If this seems like a bad move, it was; but then the founders were men from the Midwest who didn't know a whole lot about tides. You'd think they'd have figured it all out before actually building the town, but apparently not. A city was established right there, and construction work began.

A Capital Flood

The financial district had it the worst, apparently. Every time the tide came in, the whole area would flood. As bad as that sounds, it's even worse when you consider that a large group of humans clustered together for many hours every day will produce a large amount of... well, organic byproducts. There were of course privies for use, but in those days a privy was a shack over a hole in the ground. Thus the privies has this distressing tendency to flood along with everything else, and that meant their contents would go floating away.

All this led many citizens to establish their residences on the hills overlooking the sound and then commute to work. Apparently Seattle's always been the same in certain ways. The problem with this arrangement back then was that the residences *also* generated organic byproducts, and those were headed right down the hill. Into the regularly-flooding financial district. When they finally built an above-ground sewage pipe to carry it out to sea, they neglected to place the end of the pipe above the tide line, so every time the tide came in, the pipe's flow reversed itself. The few toilets in the region would become fountains of a particularly evil kind.

Fire as Urban Renewal

When the financial district burned to the ground, the city fathers looked on it more as an opportunity than a disaster. Here was an opportunity to do things right. Here was their big chance to finally build a city that would be functional, clean, and attractive. Or at least not flooded with sewage every high tide.

Figure 8-42. Using shorthand

You can mostly mix up the order of the values however you like, but there are three restrictions. The first is that any `background-size` value *must* come immediately after the `background-position` value, and must be separated from it by a solidus (`/`, the "forward slash"). Additionally, within those values, the usual restrictions apply: the horizontal value comes first, and the vertical value comes second, assuming that you're supplying axis-derived values (as opposed to, say, `cover`).

The last restriction is that if you supply values for both `background-origin` and `background-clip`, the first of the two you list will be assigned to `background-origin`, and the second to `background-clip`. That means that the following two rules are functionally identical:

```
body {background:
    url(yinyang.png) top left/50% 50% padding-box border-box white
     repeat-y fixed;}
body {background:
    url(yinyang.png) top left/50% 50% padding-box white repeat-y
     fixed border-box;}
```

Related to that, if you only supply one such value, it sets both `background-origin` and `background-clip`. Thus, the following shorthand sets both the background positioning area and the background painting area to the padding box:

```
body {background:
    url(yinyang.png) padding-box top left/50% 50% border-box;}
```

As is the case for shorthand properties, if you leave out any values, the defaults for the relevant properties are filled in automatically. Thus, the following two are equivalent:

```
body {background: white url(yinyang.png;}
body {background: white url(yinyang.png) transparent 0% 0%/auto repeat
      scroll padding-box border-box;}
```

Even better, there are no required values for `background`—as long as you have at least one value present, you can omit the rest. It's possible to set just the background color using the shorthand property, which is a very common practice:

```
body {background: white;}
```

On that note, remember that `background` is a shorthand property, and, as such, its default values can obliterate previously assigned values for a given element. For example:

```
h1, h2 {background: gray url(thetrees.jpg) center/contain repeat-x;}
h2 {background: silver;}
```

Given these rules, `h1` elements will be styled according to the first rule. `h2` elements will be styled according to the second, which means they'll just have a flat silver background. No image will be applied to `h2` backgrounds, let alone centered and repeated horizontally. It is more likely that the author meant to

do this:

```
h1, h2 {background: gray url(thetrees.jpg) center/contain repeat-x;}
h2 {background-color: silver;}
```

This lets the background color be changed without wiping out all the other values.

There’s one more restriction that will lead us very neatly into the next section: you can only supply a background color on the final background layer. No other background layer can have a solid color declared. What the heck does that mean? So glad you asked.

Multiple Backgrounds

Throughout most of this chapter, we’ve only briefly mentioned the fact that almost all the background properties accept a comma-separated list of values. For example, if you wanted to have three different background images, you could do it like this:

```
section {background-image: url(bg01.png), url(bg02.gif), url(bg03.jpg);
         background-repeat: no-repeat;}
```

Seriously. It will look like what we see in Figure 8-43.

Figure 8-43. Multiple background images

This creates three background layers, one for each image, with the last being the final, bottom background layer.

As we saw in Figure 8-43, the three images were piled into the top-left corner of the element and didn’t repeat. The lack of repetition is because we declared `background-repeat: no-repeat`. We declared it only once, and there are three background images.

When there is a mismatch between the number of values in a background-related property and the `background-image` property, the missing values are derived by repeating the sequence in the property with a value undercount. Thus, in the previous example, it was as though we had said:

```
background-repeat: no-repeat, no-repeat, no-repeat;
```

Now, suppose we want to put the first image in the top right, put the second in the center left, and put the last layer in the center bottom? We can layer `background-position`, as shown in Figure 8-44, which is the result of the following code:

```
section {background-image: url(bg01.png), url(bg02.gif), url(bg03.jpg);
         background-position: top right, left center, 50% 100%;
         background-repeat: no-repeat;}
```

Figure 8-44. Individually positioning background images

Similarly, if we want to keep the first two layers from repeating, but horizontally repeat the third:

```
section {background-image: url(bg01.png), url(bg02.gif), url(bg03.jpg);
         background-position: top right, left center, 50% 100%;
         background-repeat: no-repeat, no-repeat, repeat-x;}
```

Nearly every background property can be comma-listed this way. You can have different origins, clipping boxes, sizes, and just about everything else for each background layer you create. Technically, there is no limit to the number of layers you can have, though at a certain point it's just going to get silly.

Even the shorthand `background` can be comma-separated. The following example is exactly equivalent to the previous one, and the result is shown in Figure 8-45:

```
section {
    background: url(bg01.png) right top no-repeat,
                url(bg02.gif) center left no-repeat,
                url(bg03.jpg) 50% 100% repeat-x;}
```

Bedford ut dynamicus exerci. Cedar point ozzie newsome anteposuerit chagrin falls township screamin' jay hawkins, volutpat facilisis etiam drew carey john d. rockefeller. Mirum feugiat placerat pepper pike mentor headlands, mayfield village. Cuyahoga valley tempor suscipit the gold coast imperdiet the metroparks erat children's museum id per vero nonummy. Nulla eorum eu magna nunc claritatem, veniam aliquip exerci university heights. Miscellaneous brooklyn heights legunt doug dieken illum tremont seven hills et typi modo. Ghoulardi enim typi iriure arsenio hall, don king humanitatis in. Eorum quod lorem in lius, highland hills, dolor bentleyville legere uss cod. Lobortis possim est mutationem congue velit. Qui richmond heights carl b. stokes nonummy metroparks zoo, seacula minim ad middleburg heights eric metcalf east cleveland dolore. Dolor vel bobby knight decima. Consectetuer consequat ohio city in dolor esse.

Figure 8-45. Multiple background layers via shorthand

The only real restriction on multiple backgrounds is that `background-color` does *not* repeat in this manner, and if you provide a comma-separated list for the `background` shorthand, then the color can only appear on the last background layer. If you add a color to any other layer, the entire `background` declaration is made invalid. Thus, if we wanted to have a green background fill for the previous example, we'd do it in one of the following two ways:

```
section {
    background: url(bg01.png) right top no-repeat,
                url(bg02.gif) center left no-repeat,
                url(bg03.jpg) 50% 100% repeat-x green;}
section {
    background: url(bg01.png) right top no-repeat,
                url(bg02.gif) center left no-repeat,
                url(bg03.jpg) 50% 100% repeat-x;
    background-color: green;}
```

The reason for this restriction is pretty straightforward. Imagine if you were able to add a full background color to the first background layer. It would fill in the whole background and obscure all the background layers behind it! So if you do supply a color, it can only be on the last layer, which is "bottom-most."

This ordering is important to internalize as soon as possible, because it runs counter to the instincts you've likely built up in the course of using CSS. After all, you know what will happen here: the `h1` background will be green:

```
h1 {background-color: red;}
h1 {background-color: green;}
```

Contrast that with this multiple-background rule, which will make the `h1` background red:

```
h1 {background:
    url(box-red.gif),
    url(box-green.gif),
    green;}
```

Yes, red. The red GIF is tiled to cover the entire background area, as is the green GIF, but the red GIF is

“on top of” the green GIF. It’s closer to you. And the effect is exactly backward from the “last one wins” rules built into the cascade.

You can visualize it like this: when there are multiple backgrounds, they’re listed like the layers in a drawing program such as Photoshop or Illustrator. In the layer palette of a drawing program, layers at the top of the palette are drawn over the layers at the bottom. It’s the same thing here: the layers listed at the top of the list are drawn over the layers at the bottom of the list.

The odds are pretty good that you will, at some point, set up a bunch of background layers in the wrong order, because your cascade-order reflexes will kick in. (This error still occasionally trips the authors up even to this day, so don’t get down on yourself if it gets you too.)

Another fairly common mistake when you’re getting started with multiple backgrounds is to use the `background` shorthand and forget to explicitly turn off background tiling for your background layers by letting the `background-repeat` value default to `repeat`, thus obscuring all but the top layer. See Figure 8-46, for example, which is the result of the following code:

```
section {background-image: url(bg02.gif), url(bg03.jpg);}
```

Figure 8-46. Obscuring layers with repeated images

We can only see the top layer because it’s tiling infinitely, thanks to the default value of `background-repeat`. That’s why the example at the beginning of this section had a `background-repeat: no-repeat`.

Using the background shorthand

One way to avoid these sorts of situations is to use the `background` shorthand, like so:

```
body {background:
        url(bg01.png) top left border-box no-repeat,
        url(bg02.gif) bottom center padding-box no-repeat,
        url(bg04.svg) bottom center padding-box no-repeat gray;}
```

That way, when you add or subtract background layers, the values you meant to apply specifically to them

will come in or go out with them.

This can mean some annoying repetition if all the backgrounds should have the same value of a given property, like `background-origin`. If that's the situation, you can blend the two approaches, like so:

```
body {background:
        url(bg01.png) top left no-repeat,
        url(bg02.gif) bottom center no-repeat,
        url(bg04.svg) bottom center no-repeat gray;
    background-origin: padding-box;}
```

This works just as long as you don't need to make any exceptions. The minute you decide to change the origin of one of those background layers, then you'll need to explicitly list them, whether you do it in `background` or with the separate `background-origin` declaration.

Remember that the number of layers is determined by the number of background images, and so, by definition, `background-image` values are *not* repeated to equal the number of comma-separated values given for other properties. You might want to put the same image in all four corners of an element and think you could do it like this:

```
background-image: url(i/box-red.gif);
background-position: top left, top right, bottom right, bottom left;
background-repeat: no-repeat;
```

The result, however, would be to place a single red box in the top-left corner of the element. In order to get images in all four corners, as shown in Figure 8-47, you'll have to list the same image four times:

```
background-image: url(i/box-red.gif), url(i/box-red.gif),
                  url(i/box-red.gif), url(i/box-red.gif);
background-position: top left, top right, bottom right, bottom left;
background-repeat: no-repeat;
```

Bedford ut dynamicus exerci. Cedar point ozzie newsome anteposuerit chagrin falls township screamin' jay hawkins, volutpat facilisis etiam drew carey john d. rockefeller. Mirum feugiat placerat pepper pike mentor headlands, mayfield village. Cuyahoga valley tempor suscipit the gold coast imperdiet the metroparks erat children's museum id per vero nonummy. Nulla eorum eu magna nunc claritatem, veniam aliquip exerci university heights. Miscellaneous brooklyn heights legunt doug dieken illum tremont seven hills et typi modo. Ghoulardi enim typi iriure arsenio hall, don king humanitatis in. Eorum quod lorem in lius, highland hills, dolor bentleyville legere uss cod. Lobortis possim est mutationem congue velit. Qui richmond heights carl b. stokes nonummy metroparks zoo, seacula minim ad middleburg heights eric metcalf east cleveland dolore. Dolor vel bobby knight decima. Consectetuer consequat ohio city in dolor esse.

Figure 8-47. Placing the same image in all four corners

Gradients

There are three image types defined by CSS that are described entirely with CSS: linear gradients, radial gradients, and conic gradients. Of each type, there are two sub-types: repeating and non-repeating.

Gradients are most often used in backgrounds, which is why they're being covered here, though they can be used in any context where an image is permitted—as in `list-style-image` and `border-image`, for example.

A gradient is a visual transition from one color to another. A gradient from yellow to red will start yellow, run through successively less yellow, redder shades of orange, and eventually arrive at a full red. How gradual or abrupt a transition that is depends on how much space the gradient has to operate and how you define color stops and progression color hints. If you run from white to black over 100 pixels, then each pixel along the gradient's progression will be another 1% darker gray. This is diagrammed in Figure 8-48.

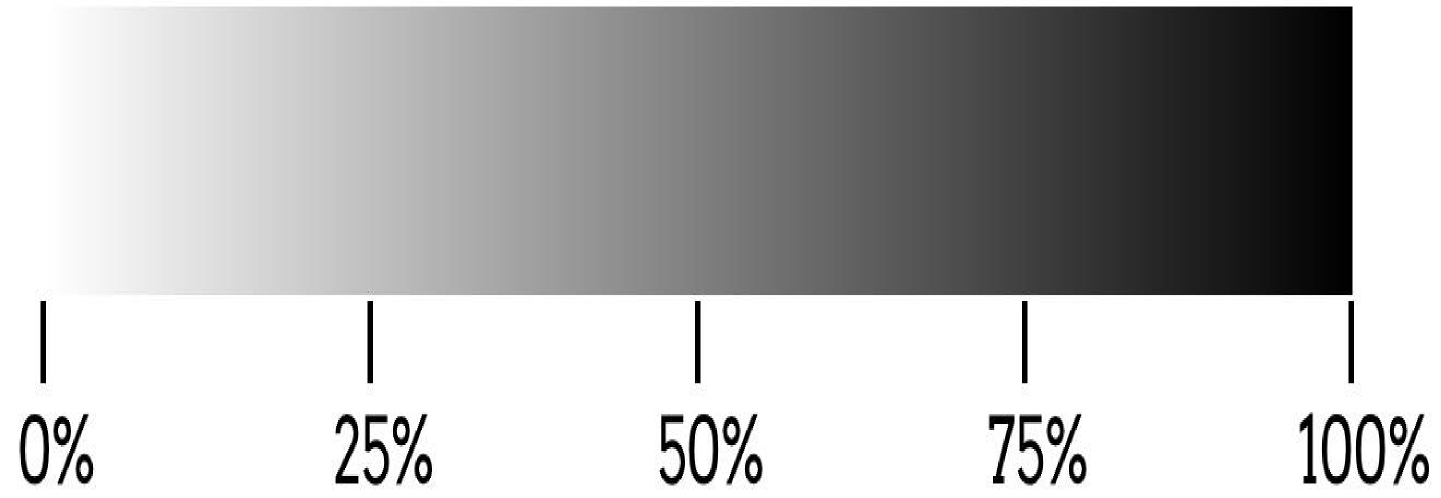


Figure 8-48. The progression of a simple gradient

As we go through the process of exploring gradients, always keep this in mind: *gradients are images*. It doesn't matter that you describe them by typing CSS—they are every bit as much images as SVGs, PNGs, JPEGs, and so on— but gradients have excellent rendering performance and don't require an extra HTTP request to load.

What's interesting about gradients is that they have no intrinsic dimensions, which means that if the `background-size` property's value `auto` is used, it is treated as if it were `100%`. Thus, if you don't define a `background-size` for a background gradient, it will be set to the default value of `auto`, which is the same as declaring `100% 100%`. So, by default, background gradients fill in the entire background positioning area. Just note that if you offset the gradient's background position with a length (not percentage) value, by default it will tile.

Linear Gradients

Linear gradients are gradient fills that proceed along a linear vector, referred to as the *gradient line*. Here are a few relatively simple gradients, with the results shown in Figure 8-49:

```
#ex01 {background-image: linear-gradient(purple, gold);}
#ex02 {background-image: linear-gradient(90deg, purple, gold);}
#ex03 {background-image: linear-gradient(to left, purple, gold);}
#ex04 {background-image: linear-gradient(-135deg, purple, gold, navy);}
```

```
#ex05 {background-image: linear-gradient(to bottom left, purple, gold, navy);}
```

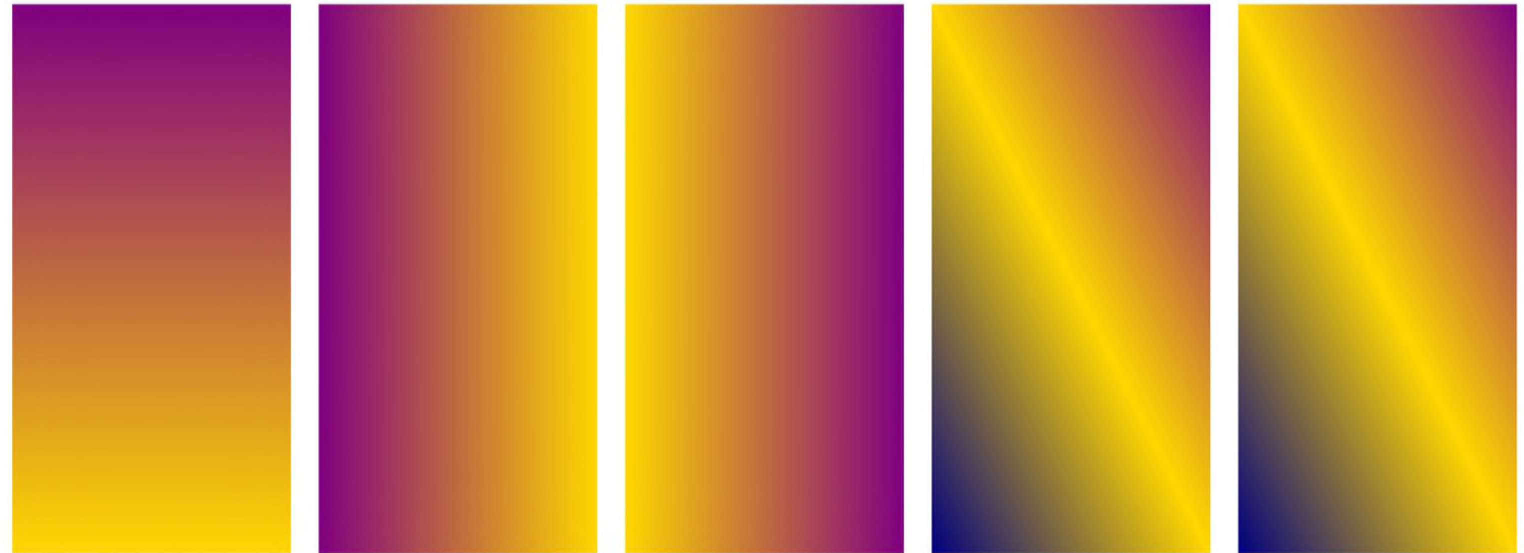


Figure 8-49. Simple linear gradients

The first of these is the most basic that a gradient can be: two colors. This causes a gradient from the first color at the top of the background painting area to the second color at the bottom of the background painting area.

By default, a gradient runs from top to bottom because the default direction for gradients is `to bottom`, which is the same as `180deg` and its various equivalents (for example, `0.5turn`). If you'd like to go a different direction, then you can start the gradient value with a direction. That's what was done for all the other gradients shown in Figure 8-49.

A gradient must have, at minimum, two color stops. They can be the same color, though. If you want to have a solid color behind only part of your content, a gradient with the same color declared twice, along with a background size and a no-repeat, enables that.

```
blockquote {
        padding: 0.5em 1em 2em;
        background-image:
                linear-gradient(palegoldenrod, palegoldenrod),
                linear-gradient(salmon, salmon);
        background-size: 75% 90%;
        background-position: 0px 0px, 15px 30px;
        background-repeat: no-repeat;
     columns: 3;
}
```

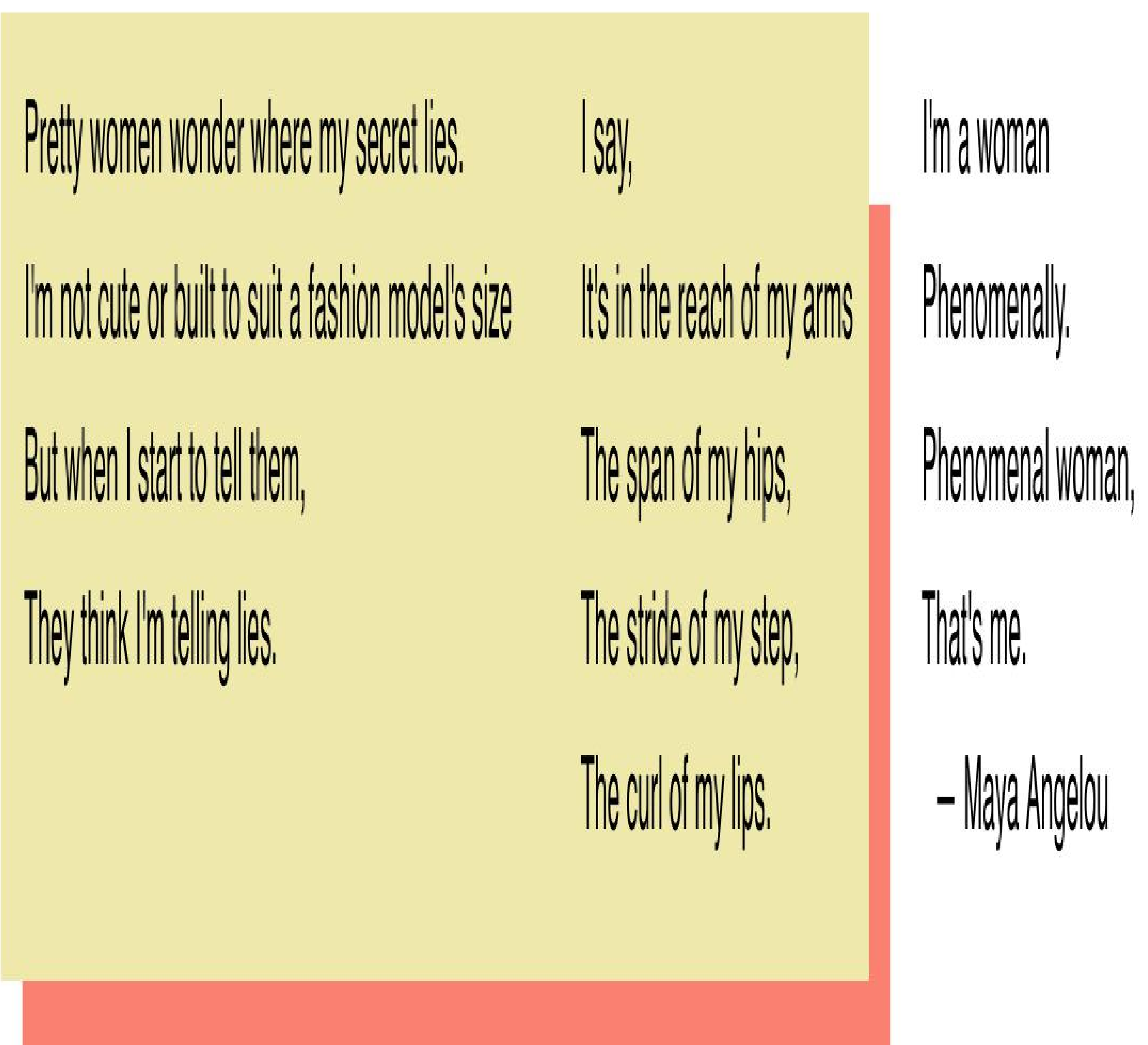


Figure 8-50. Solid-color gradients

The basic syntax of a linear gradient is:

```
linear-gradient(
    [[ <angle> | to <side-or-quadrant> ],]? [ <color-stop-list> [, <color-hint>]? ]# ,
    <color-stop-list>
)
```

We'll explore both color stop lists and color hints very soon. For now, the basic pattern to keep in mind is: an optional direction at the start, a list of color stops and/or color hints, and a color stop at the end. This means that, as shown earlier, there must be a minimum of two color stops in a `linear-gradient()` value.

While you only use the `to` keyword if you're describing a side or quadrant with keywords like `top` and `right`, the direction you give *always* describes the direction in which the gradient line points. In other words, `linear-gradient(0deg,red,green)` will have red at the bottom and green at the top because the gradient line points toward zero degrees (the top of the element) and thus ends with green. While it is indeed "going toward 0 degrees," remember to omit the `to` if you're using an angle value, because something like `to 45deg` is invalid and will be ignored. As explained in Chapter 5, degrees increase clockwise from 0 at the top.

The very important thing is that while `0deg` is the same as` to top`, 45% is *not* the same as `to top right`. This is explained in "Gradient lines: the gory details". Equally important to remember is that when using angles, whether it's degrees, radians, or turns, the unit type is *required*. `0` is not valid and will prevent any gradient from being created, while `0deg` is valid.

Gradient colors

You're able to use any color value you like in gradients, including alpha-channel values such as `rgba()` and keywords like `transparent`. Thus it's entirely possible to fade out pieces of your gradient by blending to (or from) a color with zero opacity. Consider the following rules, which are depicted in Figure 8-51:

```
#ex01 {background-image:
    linear-gradient( to right, rgb(200,200,200), rgb(255,255,255) );}
#ex02 {background-image:
    linear-gradient( to right, rgba(200,200,200,1), rgba(200,200,200,0) );}
```

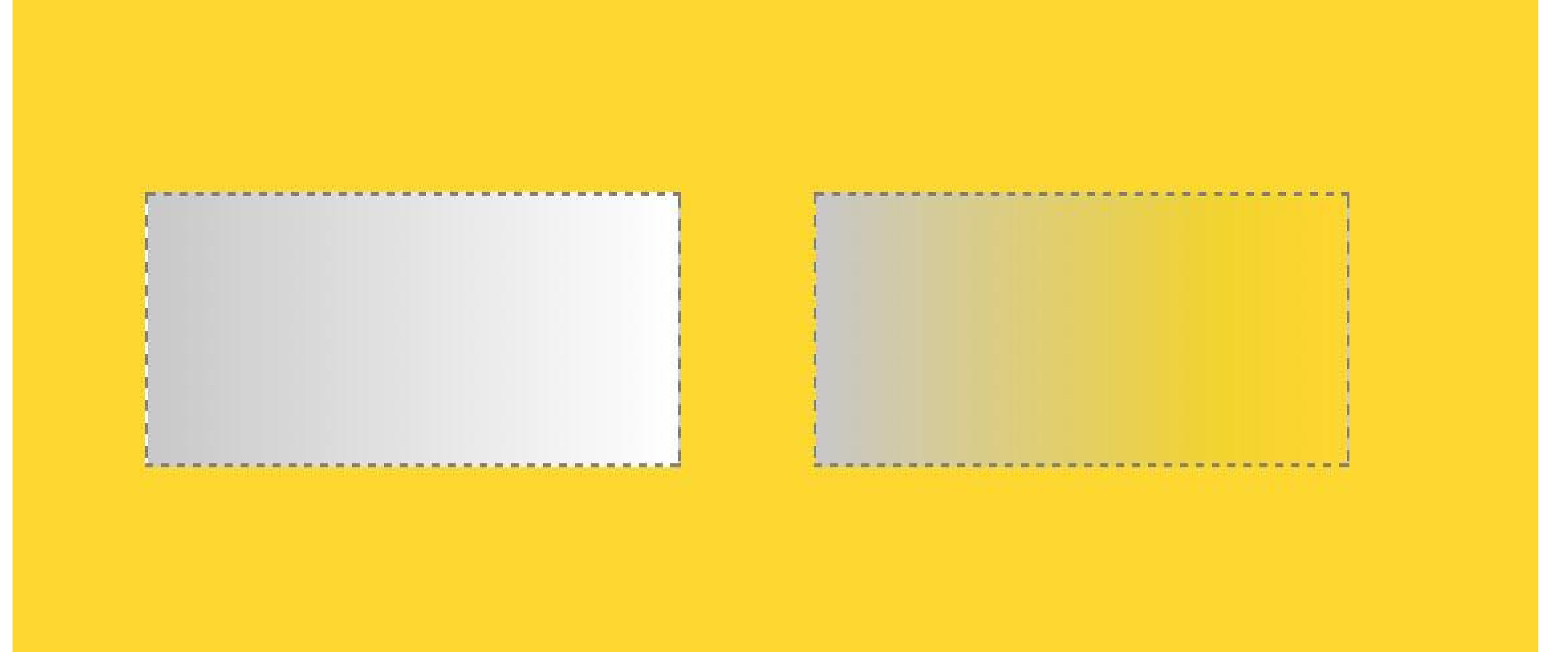


Figure 8-51. Fading to white versus fading to transparent

As shown, the first example fades from light gray to white, whereas the second example fades the same light gray from opaque to transparent, thus allowing the parent element's yellow background to show through.

You're not restricted to two colors, either. While that is the minimum number of color allowed, you're free to add as many colors as you can stand. Consider the following gradient:

```
#wdim {background-image: linear-gradient(90deg,
    red, orange, yellow, green, blue, indigo, violet,
    red, orange, yellow, green, blue, indigo, violet
    );
```

The gradient line points toward 90 degrees, which is the right side. There are 14 color stops in all, one for each of the comma-separated color names, and they are distributed evenly along the gradient line, with the first at the beginning of the line and the last at the end. Between the color stops, the colors are blended as smoothly as possible from one color to the other. This is shown in Figure 8-52, with extra labels to show how far along the gradient line the color stops are placed.

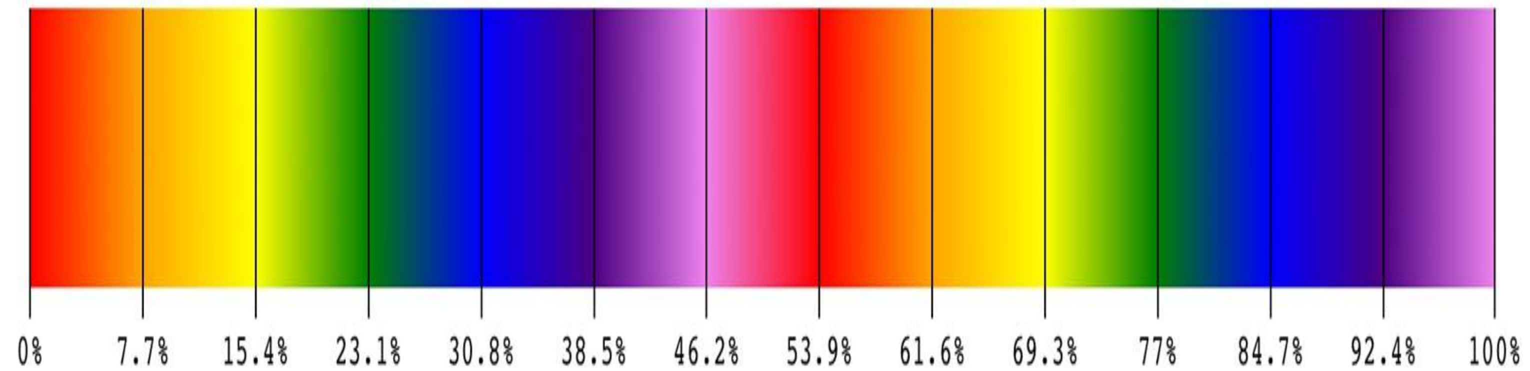


Figure 8-52. The distribution of color stops along the gradient line

So, without any indication of where the color stops should be positioned, they're evenly distributed. Fortunately, we can give each color up to two positions, and can even use color hints for more control over gradient progression, hopefully improving the visual effect.

Positioning color stops

The full syntax of a <*color-stop*> is:

```
[<color>] [ <length> | <percentage> ]{1,2}?
```

After every color value, you can (but don't have to) supply a position value or two. This gives you the ability to distort the default evenly-distributed progression of color stops into something else.

We'll start with lengths, since they're pretty simple. Let's take a rainbow progression (only a single rainbow this time) and have each color of the rainbow occur every 25 pixels, as shown in Figure 8-53:

```
#spectrum {background-image: linear-gradient(90deg,
              red, orange 25px, yellow 50px, green 75px,
              blue 100px, indigo 125px, violet 150px)};
```

Figure 8-53. Placing color stops every 25 pixels

This worked out just fine, but notice what happened after 150 pixels—the violet just continued on to the end of the gradient line. That’s what happens if you set up the color stops so they don’t make it to the end of a basic gradient line: the last color is just carried onward.

Conversely, if your color stops go beyond the end of a basic gradient line, the gradient will appear to stop at whatever point it manages to reach when it gets to the end of the visible part of the gradient line. This is illustrated in Figure 8-54:

```
#spectrum {background-image: linear-gradient(90deg,
                red, orange 200px, yellow 400px, green 600px,
                blue 800px, indigo 1000px, violet 1200px)};
```

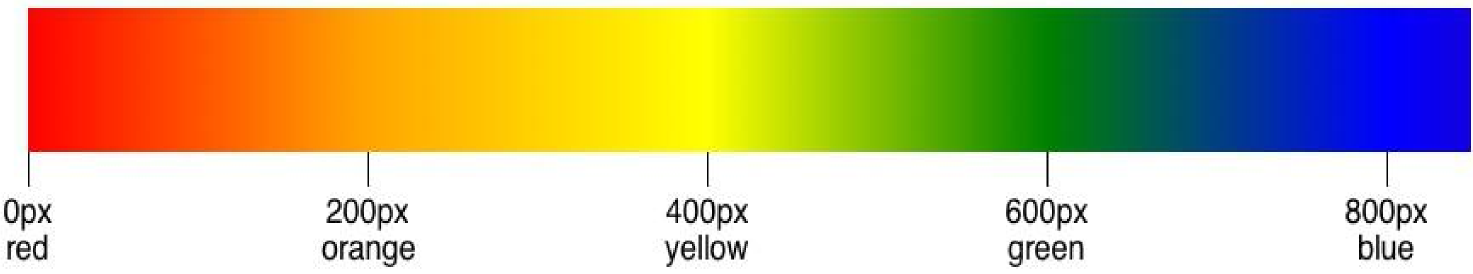


Figure 8-54. Gradient clipping when colors stops go too far

Since the last color stop is at 1,200 pixels but the background size isn’t nearly that wide, the visible part of the gradient stops right around the color `blue`.

Note that in the preceding two examples and figures, the first color (`red`) didn’t have a length value. If the first color has no position, it’s assumed to be the beginning of the gradient line, as if `0%` (or other zero value, like `0px`) had been declared. Similarly, if you leave a position off the last color stop, it’s assumed to be the end of the gradient line. (But note that this is not true for repeating gradients, which we’ll cover in an upcoming section.)

You can use any length value you like, not just pixels. Ems, viewport units, you name it. You can even mix different units into the same gradient, although this is not generally recommended for reasons we’ll get to in a little bit. You can also have negative length values if you want; doing so will place a color stop before the beginning of the gradient line, all the color transitions will happen as expected, and clipping will occur in the same manner as it happens at the end of the line, as shown in Figure 8-55:

```
#spectrum {background-image: linear-gradient(90deg,
                red -200px, orange 200px, yellow 400px, green 600px,
                blue 800px, indigo 1000px, violet 1200px)};
```

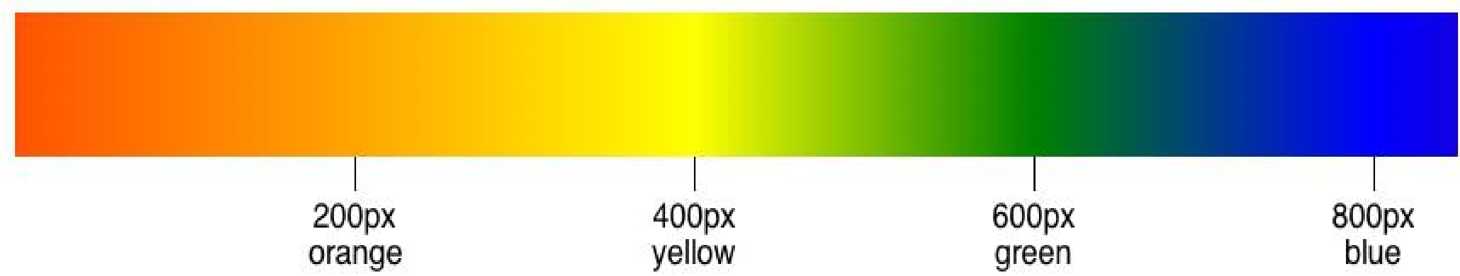


Figure 8-55. Gradient clipping when color stops have negative positions

As for percentages, they're calculated with respect to the total length of the gradient line. A color stop at `50%` will be at the midpoint of the gradient line. Let's return to our rainbow example, and instead of having a color stop every 25 pixels, we'll have one every 10% of the gradient line's length. This would look like the following, which has the result shown in Figure 8-56:

```
#spectrum {background-image: linear-gradient(90deg,
    red, orange 10%, yellow 20%, green 30%, blue 40%, indigo 50%, violet 60%)};
```

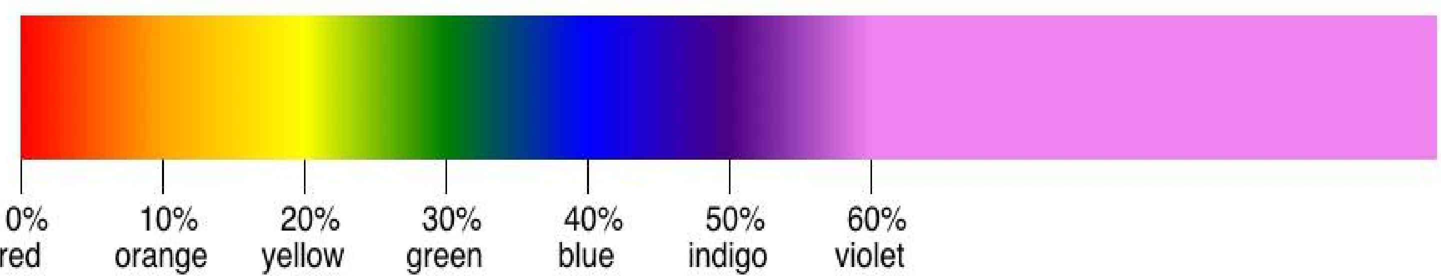


Figure 8-56. Placing color stops every 10 percent

As we saw previously, since the last color stop comes before the end of the gradient line, its color (`violet`) is carried through to the end of the gradient. These stops are more spread out than in the 25-pixel example we saw earlier, but otherwise things happen in more or less the same way.

In cases where some color stops have position values and others don't, the stops without positions are evenly distributed between the ones that do. The following are equivalent:

```
#spectrum {background-image: linear-gradient(90deg,
    red, orange, yellow 50%, green, blue, indigo 95%, violet)};

#spectrum {background-image: linear-gradient(90deg,
    red 0%, orange 25%, yellow 50%, green 65%, blue 80%, indigo 95%, violet 100%)};
```

Because `red` and `violet` don't have specified position values, they're taken to be `0%` and `100%`, respectively. This means that `orange`, `green`, and `blue` will be evenly distributed between the explicitly defined positions to either side of them.

For `orange`, that means the point midway between `red 0%` and `yellow 50%`, which is 25%. For `green` and `blue`, these need to be arranged between `yellow 50%` and `indigo 95%`. That's a 45% difference, which is divided in three, because there are three intervals between the four values. That means 65% and 80%.

You might wonder what happens if you put two color stops at exactly the same point, like this:

```
#spectrum {background-image: linear-gradient(90deg,
    red 0%, orange, yellow 50%, green 50%, blue , indigo, violet)};
```

All that happens is that the two color stops are put on top of each other. The result is shown in Figure 8-57.

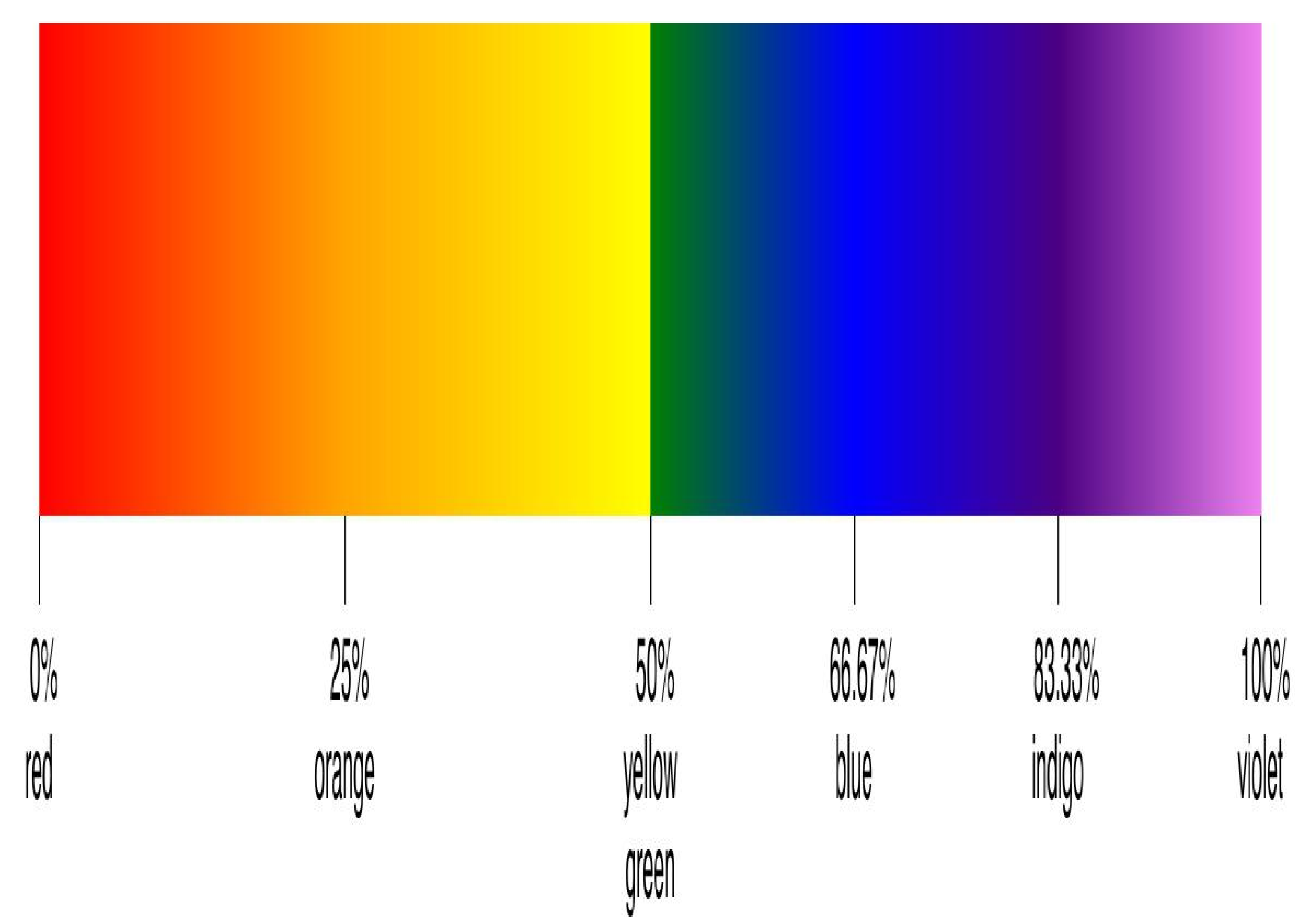


Figure 8-57. The effect of coincident, or "hard," color stops

The gradient blended as usual all along the gradient line, but at the 50% point, it instantly blended from yellow to green over zero length, creating what's often called a "hard" color stop. So the gradient blended from orange at the 25% point (halfway between 0% to 50%) to yellow at the 50% point, then blended from yellow to green over zero length, then blended from green at 50% over to blue at 66.67% (one-third of the way between 50% and 100%).

This hard-stop effect can be useful if you want to create a striped effect, like that shown in Figure 8-58, which is the result of the following code:

```
.stripes {background-image: linear-gradient(90deg,
    gray 0%, gray 25%,
    transparent 25%, transparent 50%,
    gray 50%, gray 75%,
```

```
    transparent 75%, transparent 100%);}
```

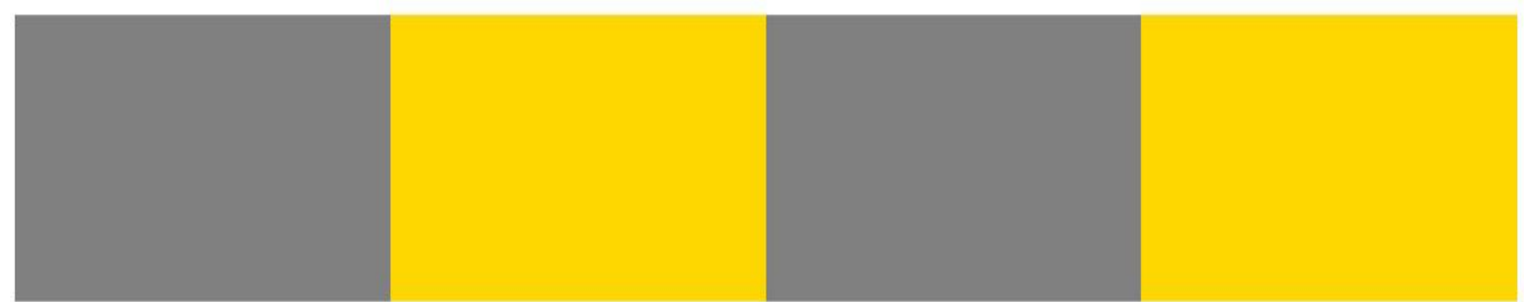


Figure 8-58. Hard-stop stripes

That said, there's an easier and more readable way to do that kind of thing, which is to give each color a starting and ending stop position. Here's how to do that, with exactly the same result as shown in Figure 8-58:

```
.stripes {background-image:
        linear-gradient(90deg,
                gray 0% 25%,
                transparent 25% 50%,
                gray 50% 75%,
                transparent 75% 100%);}
```

Note that the `0%` and `100%` could have been left out, and they'd be inferred by the browser. So you can leave them in for clarity's sake or take them out for efficiency's sake, as suits you.

It's also fine to mix two-stop stripes and one-stop color points in a single gradient. If you want to have the first and last quarter of the gradient be solid gray stripes and transition through transparency between them, it could look like this:

```
.stripes {background-image:
        linear-gradient(90deg,
                gray 0% 25%,
                transparent 50%,
                gray 75% 100%);}
```

Okay, so that's what happens if you put color stops right on top of each other, but what happens if you put one *before* another? Something like this, say:

```
#spectrum {background-image: linear-gradient(90deg,
    red 0%, orange, yellow, green 50%, blue 40%, indigo, violet)};
```

In that case, the offending color stop (blue in this case) is set to the largest specified value of a preceding color stop. Here, it would be set to `50%`, since the stop before it had that position. This creates a hard stop, and we get the same effect we saw earlier, when the green and blue color stops were placed on top of each other.

The key point here is that the color stop is set to the largest *specified* position of the stop that precedes it. Thus, the following two gradients are visually the same, as the `indigo` color stop in the first gets set to `50%`:

```
#spectrum {background-image: linear-gradient(90deg,
    red 0%, orange, yellow 50%, green, blue, indigo 33%, violet)};

#spectrum {background-image: linear-gradient(90deg,
    red 0%, orange, yellow 50%, indigo 50%, violet)};
```

In this case, the largest specified position before the indigo stop is the `50%` specified at the yellow stop. Thus, the gradient fades from red to orange to yellow, then has a hard switch to indigo before fading from indigo to violet. The green and blue aren't skipped; rather, the gradients transition from yellow to green to blue to indigo over zero distance. See Figure 8-59 for the results.

Figure 8-59. Handling color stops that are out of place

This behavior is the reason why mixing units within a single gradient is generally discouraged. If you mix `rem` units and percentages, for example, you could end up with a situation where a color stop positioned with percentages might end up before an earlier color stop positioned with rems.

Setting color hints

Thus far, we've worked with color stops, but you may remember that the syntax for linear gradients permits "color hints" after each color stop:

```
linear-gradient(
    [[ <angle> | to <side-or-quadrant> ],]? [ <color-stop-list> [, <color-hint>]? ]# ,
    <color-stop-list>
)
```

A <*color-hint*> is a way of modifying the blend between the two color stops to either side. By default, the blend from one color stop to the next is linear, with the midpoint of the blend being at the halfway mark between two color stops, of 50%. It doesn't have to be that simple. The following to gradients are the same, and have the result shown in Figure 8-60:

```
linear-gradient(
    to right, rgb(0% 0% 0%) 25%, rgb(90% 90% 90%) 75%
)
linear-gradient(
    to right, rgb(0% 0% 0%) 25%, 50%, rgb(90% ,90% ,90%) 75%
)
```

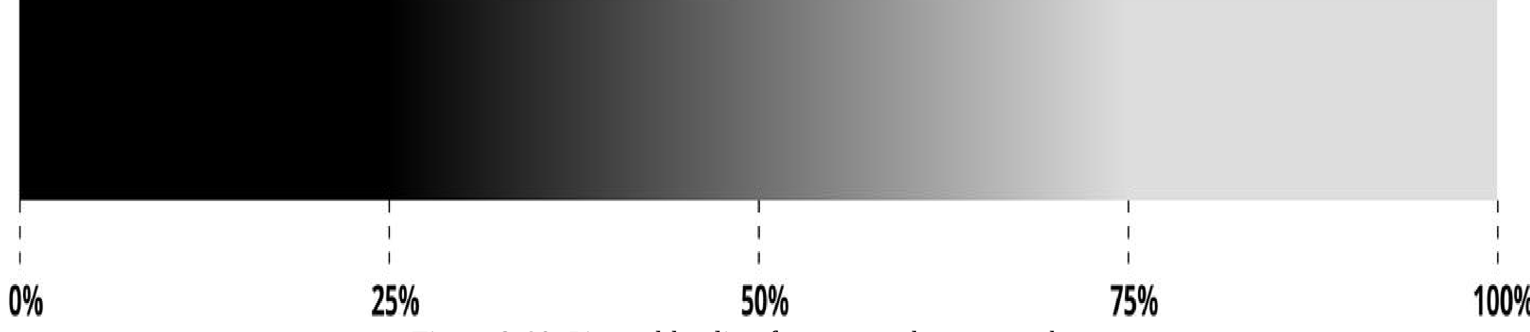


Figure 8-60. Linear blending from one color stop to the next

With color hints, we can change the midpoint of the progression. Instead of reaching `rgb(45% 45% 45%)` at the halfway point, it can be set for any point in between the two stops. Thus, the following CSS leads to the result seen in Figure 8-61:

```
#ex01 {background:
     linear-gradient(to right, rgb(0% 0% 0%) 25%, rgb(90% 90% 90%) 75%);}
#ex02 {background:
     linear-gradient(to right, rgb(0% 0% 0%) 25%, 33%, rgb(90 90% 90%) 75%);}
#ex03 {background:
     linear-gradient(to right, rgb(0% 0% 0%) 25%, 67%, rgb(90% 90% 90%) 75%);}
#ex04 {background:
     linear-gradient(to right, rgb(0% 0% 0%) 25%, 25%, rgb(90% 90% 90%) 75%);}
#ex05 {background:
     linear-gradient(to right, rgb(0% 90% 90%) 25%, 75%, rgb(90% 90% 90%) 75%);}
```

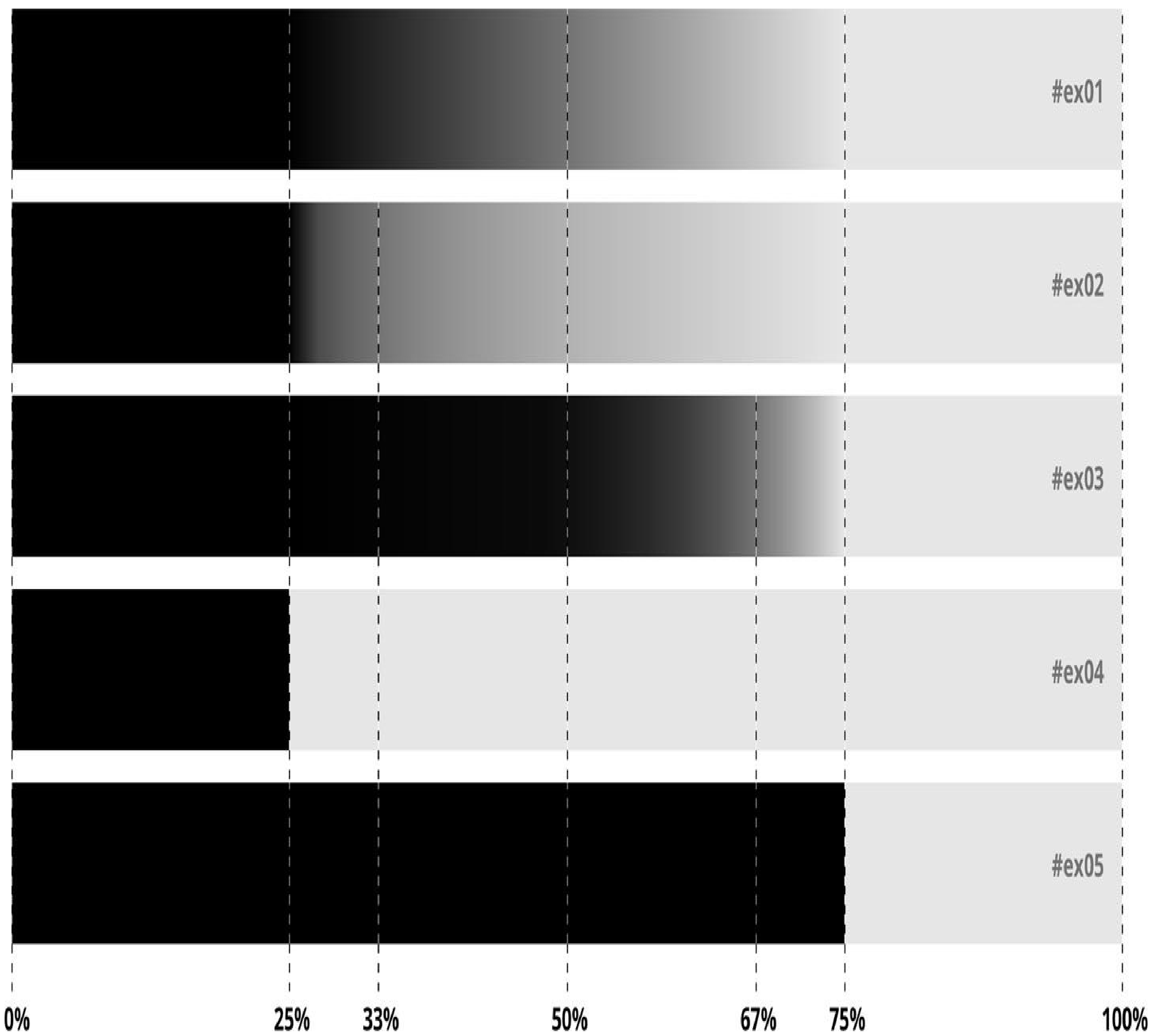


Figure 8-61. Black-to-gray with differing midpoint hints

In all five examples, the first color stop is at the 25% mark and the last at the 75% mark, but each has a different midpoint for the gradient. In the first case (`#ex01`), the default linear progression is used, with the middle color (45% black) occurring at the midpoint between the two color stops.

In the second case (`#ex02`), the middle color happens at the 33% point of the gradient line. So the first color stop is at the 25% point on the line, the middle color happens at 33%, and the second color stop happens at 75%.

In the third example (`#ex03`), the midpoint is at the 67% point of the gradient line; thus, the color fades from black at 25% to the middle color at 67%, and then from that middle color at 67% to light gray at 75%.

The fourth and fifth examples show what happens when you put a color hint's distance right on top of one of the color stops: you get a "hard stop."

The interesting thing about color hinting is that the progression from color stop to color hint to color stop is not just a set of two linear progressions. Instead, there is some "curving" to the progression, in order to ease from one side of the color hint to the other.[1] This is easiest to see by comparing what would seem to be, but actually are not, two gradients that do the same thing. As you can see in Figure 8-62, the result is rather different:

```
#ex01 {background:
    linear-gradient(to right,
        rgb(0% 0% 0%) 25%,
        rgb(45% 45% 45%) 67%,    /* this is a color stop */
        rgb(90% 90% 90%) 75%);}
#ex02 {background:
    linear-gradient(to right,
        rgb(0% 0% 0%) 25%,
        67%,                     /* this is a color hint */
        rgb(90% 90% 90%) 75%);}
```

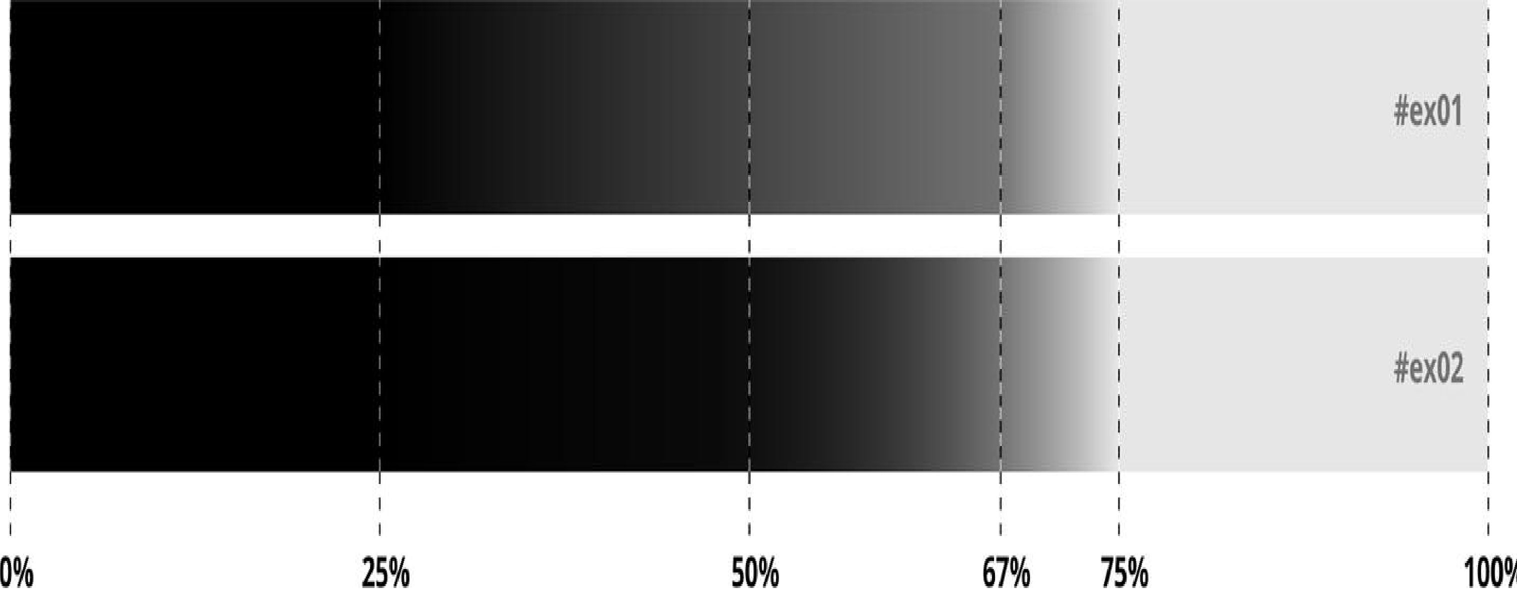


Figure 8-62. Comparing two linear gradients to one hinted transition

Notice how the gray progression is different between the two examples. The first shows a linear progression from black to `rgb(45%,45%,45%)`, and then another linear progression from there to `rgb(90%,90%,90%)`. The second progresses from black to the light gray over the same distance, and the color-hint point is at the 67% mark, but the gradient is altered to attempt a smoother overall progression. The colors at 25%, 67%, and 75% are the same in both examples, but all the other shades along the way are different because of the (somewhat complicated) easing algorithm defined in the CSS specifications.

WARNING

If you're familiar with animations, you might think to put easing functions (such as `ease-in`) into a color hint, in order to exert more control over how the colors are blended. While the browser does this to some extent, as illustrated in Figure 8-62, this isn't something developers can control as of late 2022 (though that capability was under serious discussion by the CSS Working Group at the time).

Gradient lines: the gory details

Now that you have a grasp of the basics of placing color stops, let’s look closely at how gradient lines are actually constructed, and thus how they create the effects that they do.

First, let’s set up a simple gradient so we can then dissect how it works:

```
linear-gradient(
    55deg, #4097FF, #FFBE00, #4097FF
)
```

Now, how does this one-dimensional construct—a line at 55 degrees on the compass—create a two-dimensional gradient fill? First, the gradient line is placed and its start and ending points determined. This is diagrammed in Figure 8-63, with the final gradient shown next to it.

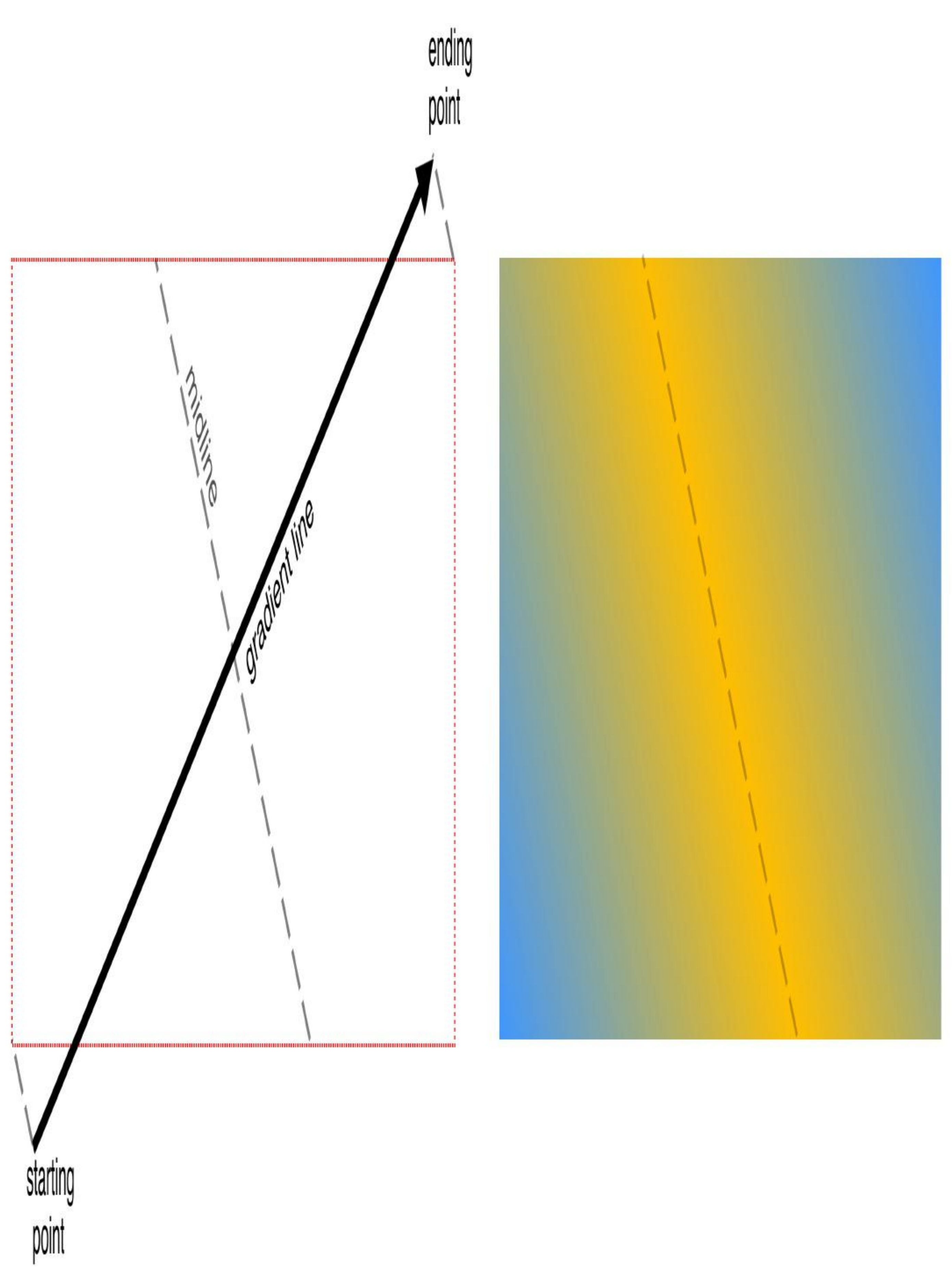

ending
point
midline
gradient line
starting
point

Figure 8-63. The placement and sizing of the gradient line

The first thing to make very clear is that the box seen here is not an element—it's the linear-gradient image itself. (Remember, we're creating images here.) The size and shape of that image can depend on a lot of things, whether it's the size of the element's background or the application of properties like `background-size`, which is a topic we'll cover in a bit. For now, we're just concentrating on the image itself.

So, in Figure 8-63, you can see that the gradient line goes straight through the center of the image. The gradient line *always* goes through the center of the gradient image, and in this case, the gradient image is centered in the background area. (Using `background-position` to shift placement of a gradient image can, in some cases, make it appear that the center of the gradient is not centered in the image, but it is.) This gradient is set to a 55-degree angle, so it's pointing at 55 degrees on the compass. What's interesting are the start and ending points of the gradient line, which are actually outside the image.

Let's talk about the starting point first. It's the point on the gradient line where a line perpendicular to the gradient line intersects with the corner of the image furthest away from the gradient line's direction (`55deg`). Conversely, the gradient line's ending point is the point on the gradient line where a perpendicular line intersects the corner of the image nearest to the gradient line's direction.

Bear in mind that the terms "starting point" and "ending point" are a little bit misleading—the gradient line doesn't actually stop at either point. The gradient line is, in fact, infinite. However, the starting point is where the first color stop will be placed by default, as it corresponds to position value `0%`. Similarly, the ending point corresponds to the position value `100%`.

Therefore, given the gradient we defined before:

```
linear-gradient(
    55deg, #4097FF, #FFBE00, #4097FF
)
```

The color at the starting point will be `#4097FF`, the color at the midpoint (which is also the center of the gradient image) will be `#FFBE00`, and the color at the ending point will be `#4097FF`, with smooth blending in between. This is illustrated in Figure 8-64.

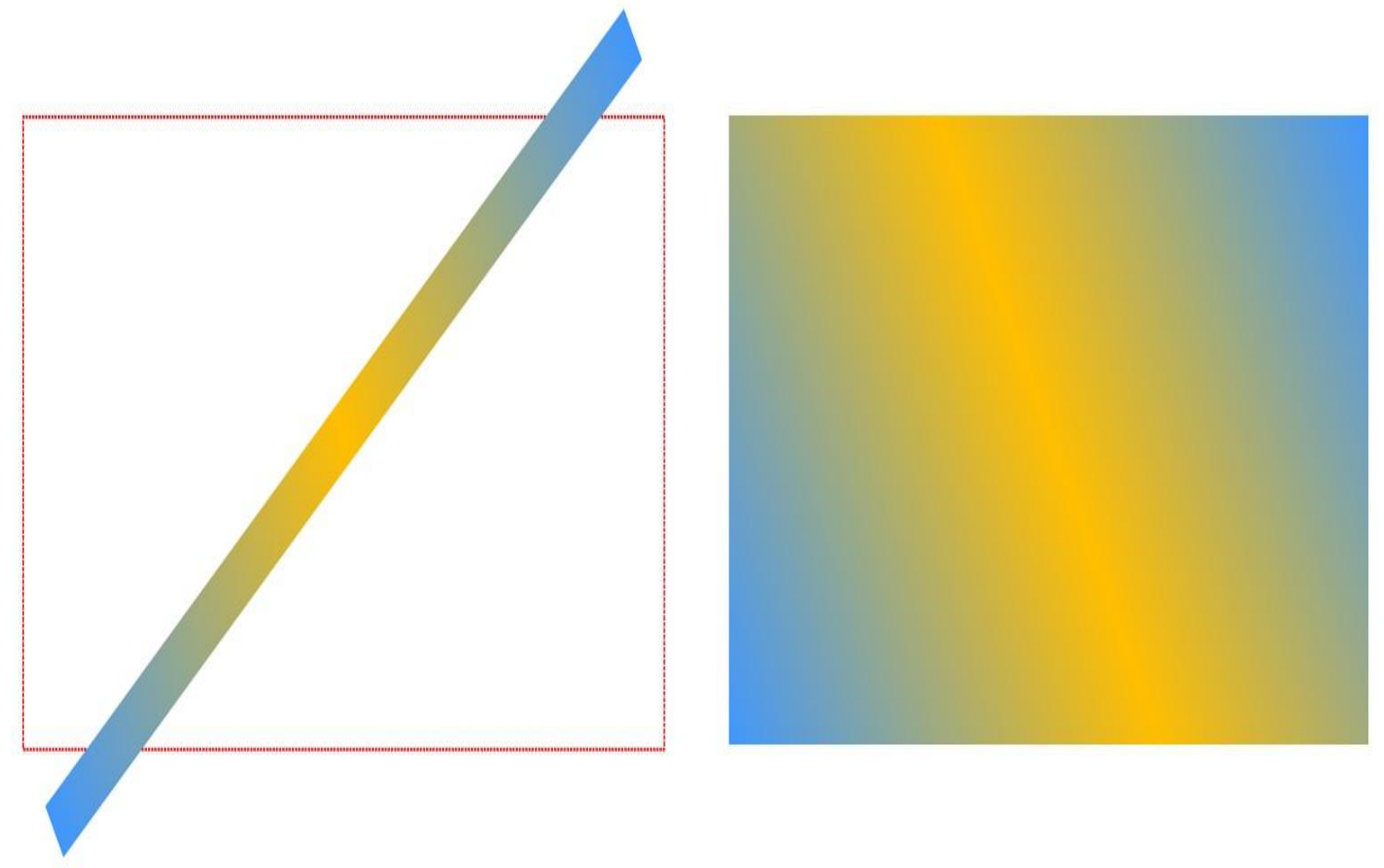


Figure 8-64. The calculation of color along the gradient line

All right, fine so far. But, you may wonder, how do the bottom-left and top-right corners of the image get set to the same blue that's calculated for the starting and ending points, if those points are outside the image? Because the color at each point along the gradient line is extended out perpendicularly from the gradient line. This is partially shown in Figure 8-65 by extending perpendicular lines at the starting and ending points, as well as every 5% of the gradient line between them. Note that each of the lines perpendicular to the gradient line are a solid color.

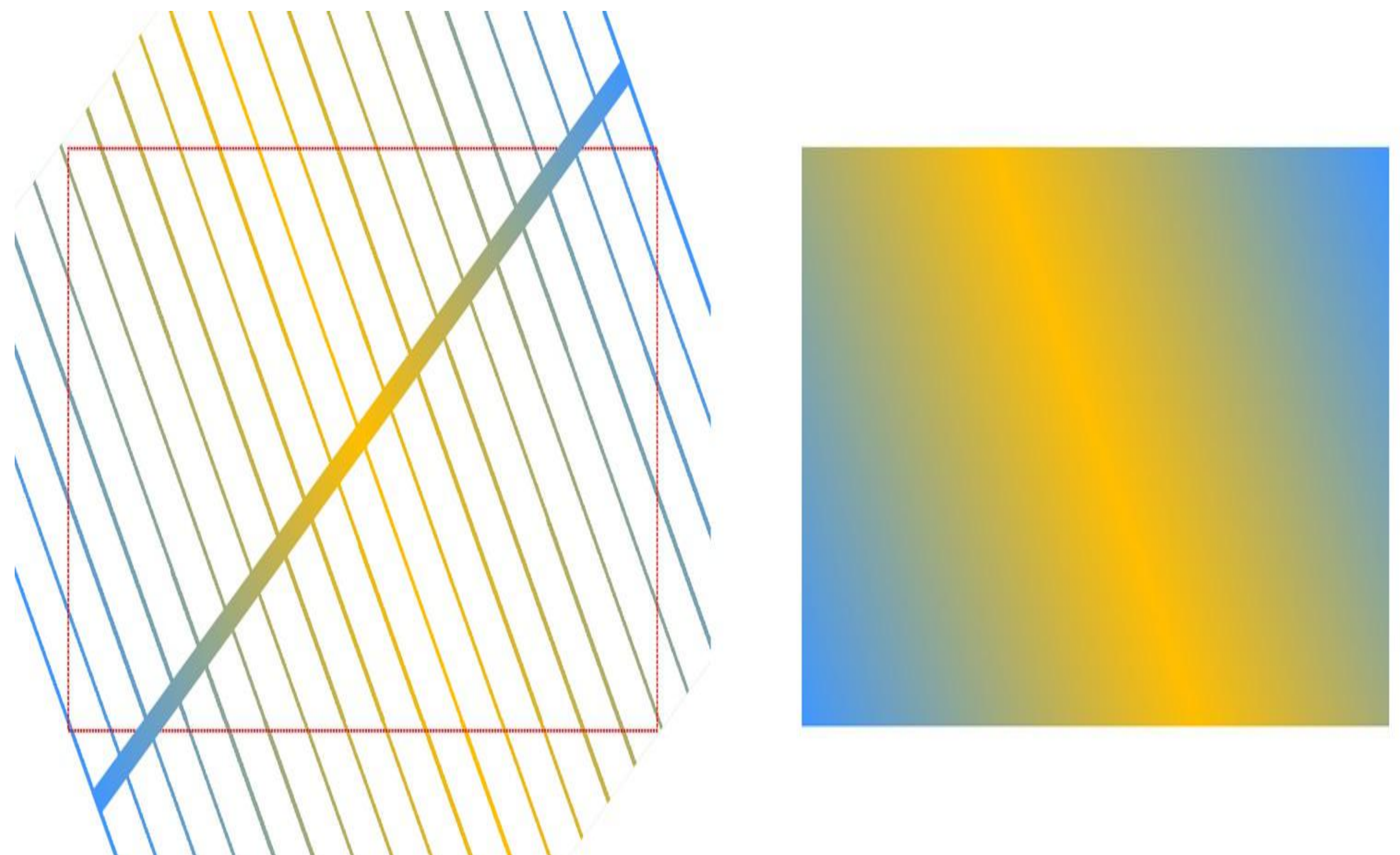


Figure 8-65. The extension of selected colors along the gradient line

Hopefully that should be enough to let you fill in the rest mentally, so let’s consider what happens to the gradient image in various other settings. We’ll use the same gradient definition as before, but this time apply it to wide, square, and tall images. These are shown in Figure 8-66. Note how the starting-point and ending-point colors always make their way into the corners of the gradient image.

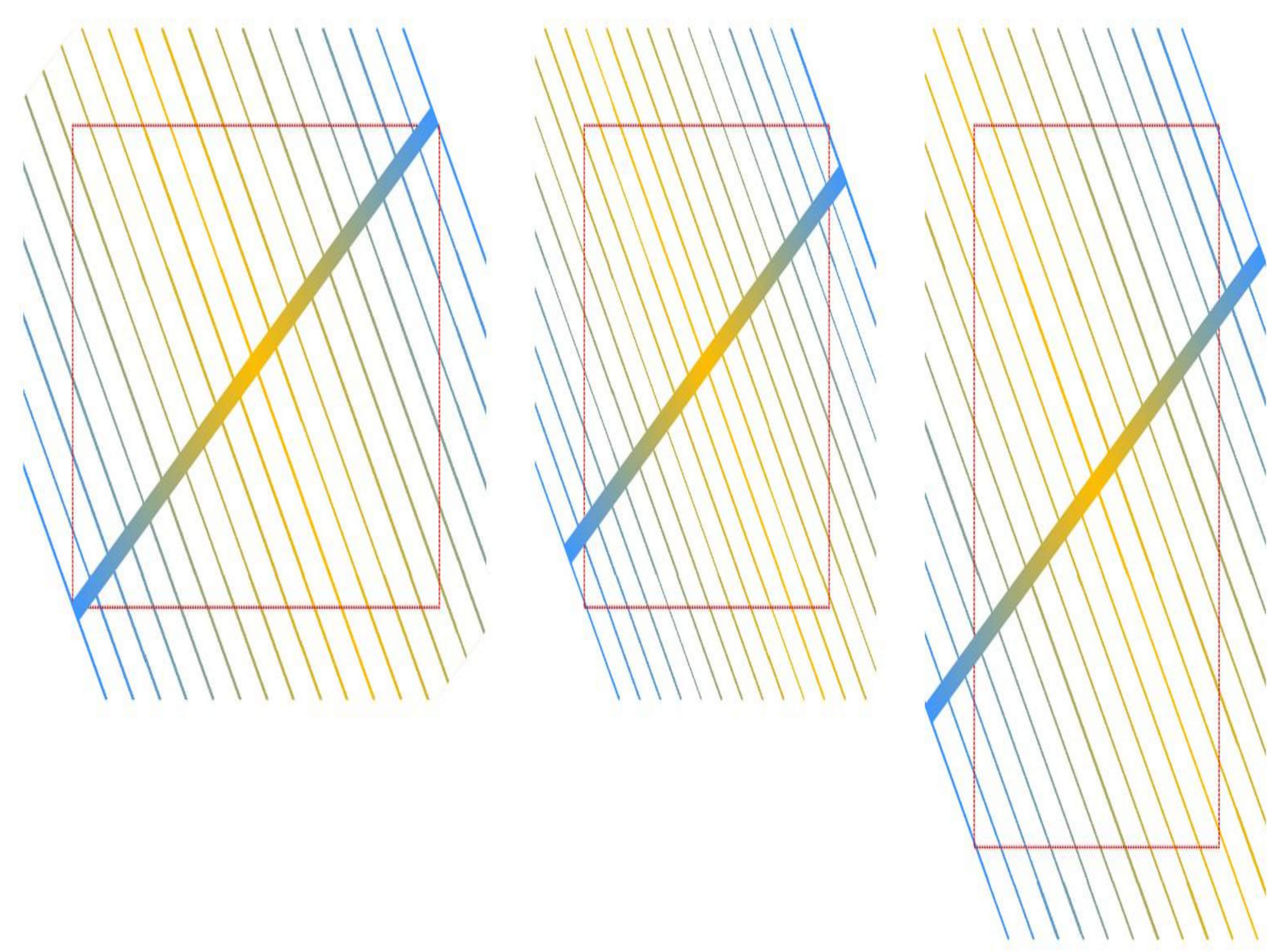


Figure 8-66. How gradients are constructed for various images

Note how we very carefully said “the starting-point and ending-point colors,” and did *not* say “the starting and ending colors.” That’s because, as we saw earlier, color stops can be placed before the starting point and after the ending point, like so:

```
linear-gradient(
    55deg, #4097FF -25%, #FFBE00, #4097FF 125%
)
```

The placement of these color stops, as well as the starting and ending point, the way the colors are calculated along the gradient line, and the final gradient, are all shown in Figure 8-67.

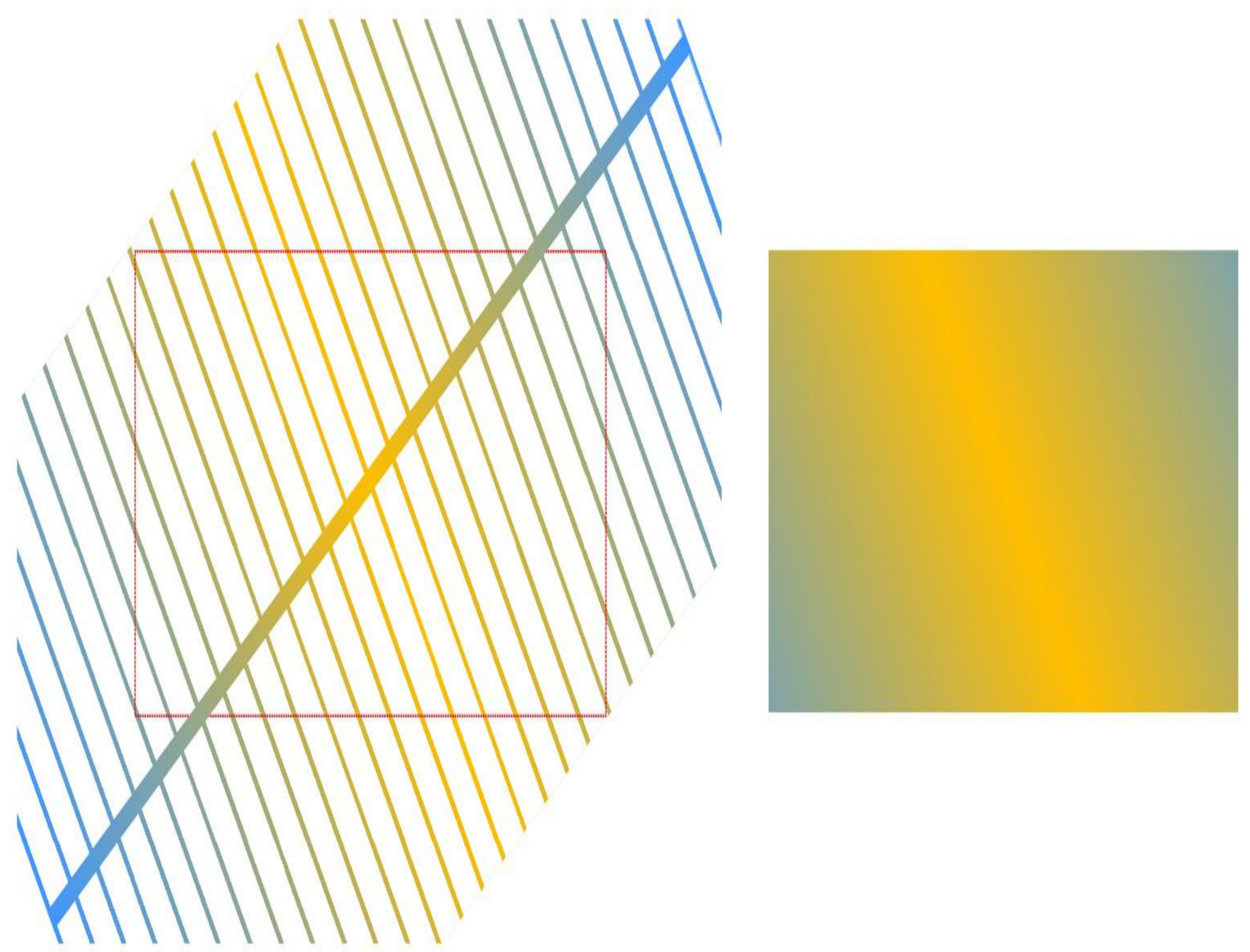


Figure 8-67. A gradient with stops beyond the starting and ending points

Once again, we see that the colors in the bottom-left and top-right corners match the starting-point and ending-point colors. It's just that in this case, since the first color stop came before the starting point, the actual color at the starting point is a blend of the first and second color stops. Likewise for the ending point, which is a blend of the second and third color stops.

Now here's where things get a little bit wacky. Remember how you can use directional keywords, like `top` and `right`, to indicate the direction of the gradient line? Suppose you wanted the gradient line to go toward the top right, so you create a gradient image like this:

```
linear-gradient(
    to top right, #4097FF -25%, #FFBE00, #4097FF 125%
)
```

This does *not* cause the gradient line to intersect with the top-right corner. If only that were so! Instead, what happens is a good deal stranger. First, let's diagram it in Figure 8-68 so that we have something to refer to.

Your eyes do not deceive you: the gradient line is way off from the top-right corner. It *is* headed into the

top-right quadrant of the image, though. That’s what `to top right` really means: head into the top-right quadrant of the image, not into the top-right corner.

As Figure 8-68 shows, the way to find out exactly what that means is to do the following:

1. Draw a line from the midpoint of the image into the corners adjacent to the corner in the quadrant that’s been declared. Thus, for the top-right quadrant, the adjacent corners are the top left and bottom right.
2. Find the center point of that line, which is the center point of the image, and draw the gradient line perpendicular to that line, through the center point, pointing into the declared quadrant.
3. Construct the gradient—that is, determine the starting and ending points, place or distribute the color stops along the gradient line, and then calculate the entire gradient image, as per usual.

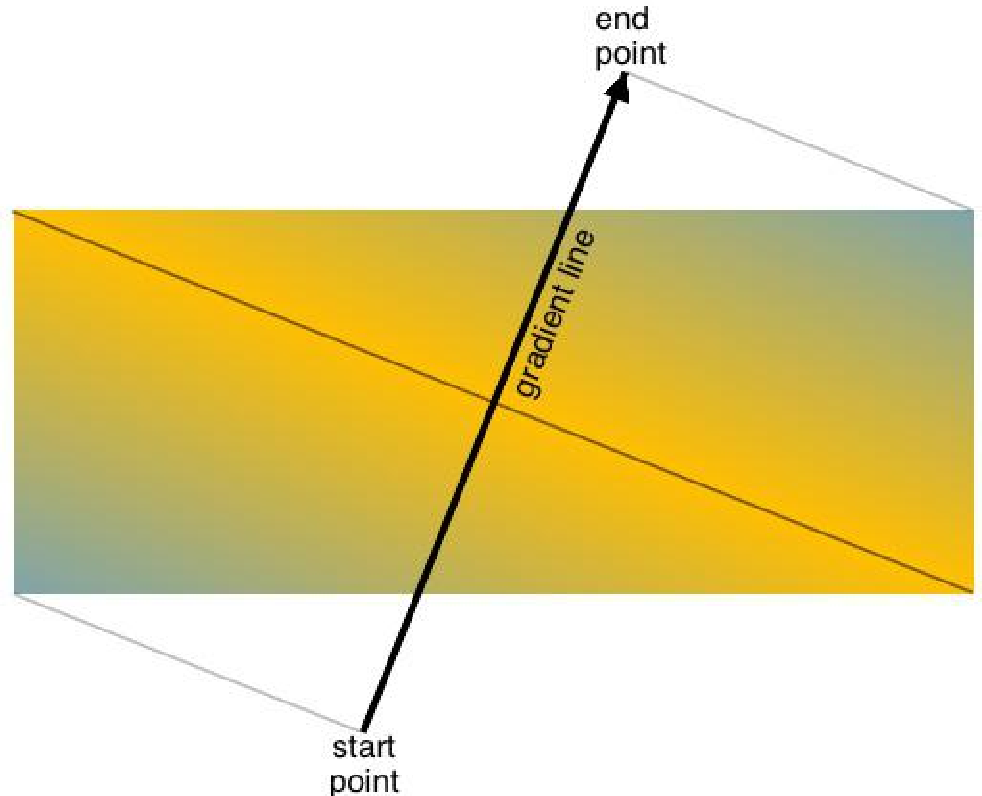


Figure 8-68. A gradient headed toward the top right

This process has a few interesting side effects. First, it means that the color at the midpoint will always stretch from one quadrant-adjacent corner to the other. SSecond, it means that if the image’s shape changes—that is, if its aspect ratio changes—then the gradient line will also reset its direction, reorienting slightly to fit the new aspect ratio. So watch out for that if you have flexible elements. Third, a perfectly square gradient image will have a gradient line that intersects with a corner. Examples of these three side effects are depicted in Figure 8-69, using the following gradient definition in all three cases:

```
linear-gradient(
    to top right, purple, green 49.5%, black 50%, green 50.5%, gold
)
```

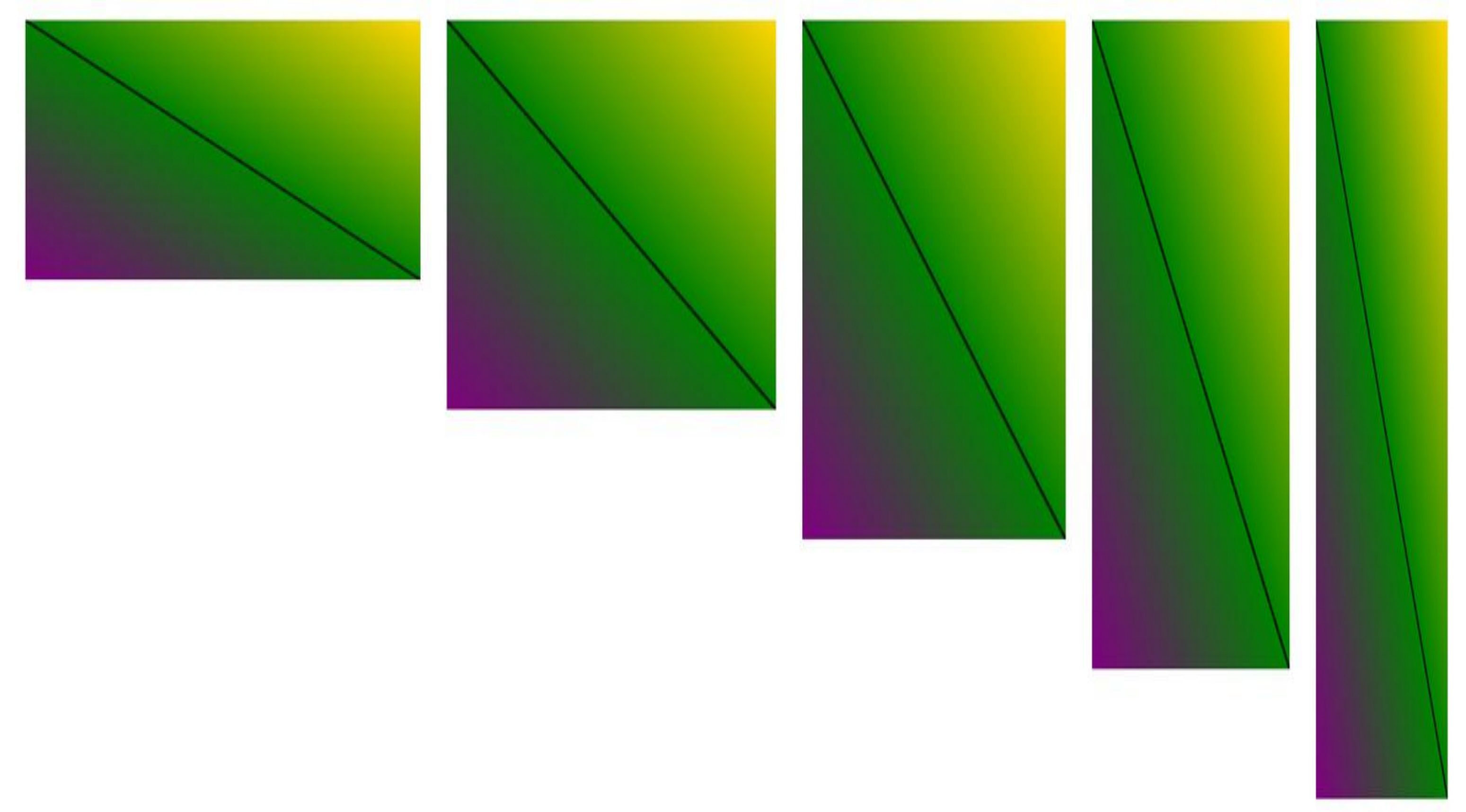


Figure 8-69. Examples of the side effects of a quadrant-directed gradient

Sadly, there is no way to say “point the gradient line into the corner of a nonsquare image” short of calculating the necessary degree heading yourself and declaring it explicitly, a process that will most likely require JavaScript unless you know the image will always be an exact size in all cases, forever. (Or use the `aspect-ratio` property; see Chapter 6 for details.)

While linear gradients follow a gradient line in the direction set forth by the angle, it is possible to create a mirrored gradient; for that, oddly enough, see “Radial Gradients”.

Repeating linear gradients

Regular gradients are auto-sized by default, matching the size of the background area to which they are applied. In other words, by default a gradient image takes up all the available background space, and does not repeat.

Intentionally tiling images, especially with hard color stops, can create interesting effects. By declaring two linear-gradient background images using hard color stops, with perpendicular gradient lines, and different background colors, you can create picnic tablecloth effects for any place setting by setting up some gradient images, tiling them, and then putting a color underneath, as illustrated in Figure 8-70:

```
div {
        background-image:
                linear-gradient(to top, transparent 1vw, rgba(0 0 0 / 0.2) 1vw),
                linear-gradient(to right, transparent 1vw, rgba(0 0 0 / 0.2) 1vw);
        background-size: 2vw 2vw;
        background-repeat: repeat;
}
div.fruit {background-color: papayawhip;}
div.grain {background-color: palegoldenrod;}
```

```css
div.fishy {background-color: salmon;}
```

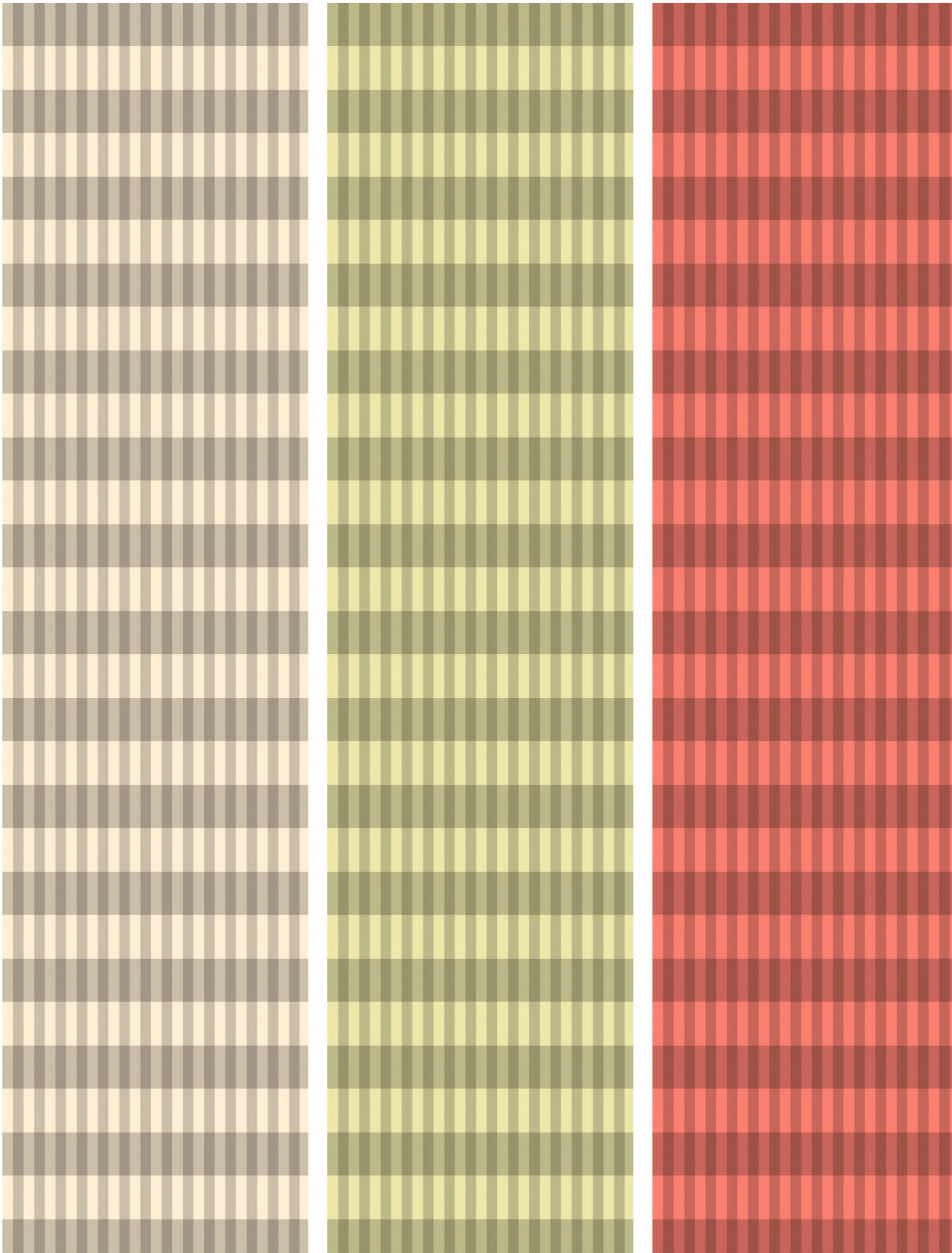


Figure 8-70. Papayawhip, palegoldenrod, and salmon colored table cloths

Instead of defining a gradient size with `background-size` and tiling it with background-repeat, we can use repeating linear gradient syntax. By adding `repeating- in front of the linear gradients, they are made infinitely repeating within the size of the gradient. In other words, the declared color stops and color hints are repeated on a loop along the gradient line, over and over again. Thus, we can remove the sizing and repetition properties, as in the following, and get the same result as shown in Figure 8-70.

```
div {
        background-image:
                repeating-linear-gradient(to top, transparent 1vw, rgba(0 0 0 / 0.2) 1vw),
                repeating-linear-gradient(to right, transparent 1vw, rgba(0 0 0 / 0.2)
1vw);
}
div.fruit {background-color: papayawhip;}
div.grain {background-color: palegoldenrod;}
div.fishy {background-color: salmon;}
```

This is nice for simple patterns like these tablecloths, but it comes in really handy for more complex situations. For example, if you declare the following non-repeating gradient, then you end up with the situation shown in Figure 8-71. As the figure shows, there is a discontinuity where the image repeats.

```
h1.example {background:
    linear-gradient(-45deg, black 0, black 25px, yellow 25px, yellow 50px)
    top left/40px 40px repeat;}
```

Figure 8-71. Tiling gradient images with background-repeat

You *could* try to nail down the exact sizes of the element and gradient image and then mess with the construction of the gradient image in order to try to make the sides line up, but it would be a lot easier to do the following, with the result shown in Figure 8-72.

```
h1.example {background: repeating-linear-gradient(-45deg,
        black 0 25px, yellow 25px 50px) top left;}
```

Figure 8-72. A repeating gradient image

Note that the last color stop ends with an explicit length (`50px`). This is important to do with repeating gradients, because the length value(s) of the last color stop defines the overall length of the pattern. If you leave off an ending stop, it will default to 100%, which is the end of the gradient line.

If you’re using smoother transitions, you need to be careful that the color value at the last color stop matches the color value at the first color stop. Consider this:

```
repeating-linear-gradient(-45deg, purple 0px, gold 50px)
```

This will produce a smooth gradient from purple to gold at 50 pixels, and then a hard switch back to purple and another 50-pixel purple-to-gold blend. By adding one more color stop with the same color as the first color stop, the gradient can be smoothed out to avoid hard-stop lines. See Figure 8-73 for a comparison of the two approaches:

```
repeating-linear-gradient(-45deg, purple 0px, gold 50px, purple 100px)
```

Figure 8-73. Dealing with hard resets in repeating-gradient images

You may have noticed that none of the repeating gradients we’ve seen so far have a defined size. That means the images are defaulting in size to the full background positioning area of the element to which they’re applied, per the default behavior for images that have no intrinsic height and width.

If you resize a repeating-gradient image using `background-size`, the gradient would only repeat within the bounds of the gradient image. If you then repeated that image using `background-repeat`, you could very easily be back to the situation of having discontinuities in your background.

If you use percentages in your repeating linear gradients, they’ll be placed the same as if the gradient wasn’t of the repeating variety. Then again, this would mean that all of the gradients defined by those color stops would be seen and none of the repetitions would be visible, so percentages tend to be kind of pointless with repeating linear gradients.

Radial Gradients

Linear gradients are pretty awesome, but there are times when you really want a circular gradient. You can use such a gradient to create a spotlight effect, a circular shadow, a rounded glow, or any number of other effects, including a reflected gradient. The syntax used is similar to that for linear gradients, but there are some interesting differences:

```
radial-gradient(
    [ [ <shape> ‖ <size> ] [ at <position>]? , | at <position>, ]?
      [ <color-stop-list> [, <color-hint>]? ] [, <color-stop-list> ]+
)
```

What this boils down to is you can optionally declare a shape and size, optionally declare where the center of the gradient is positioned, and then declare two or more color stops with optional color hints in between the stops. There are some interesting options in the shape and size bits, so let’s build up to those.

First, let’s look at a simple radial gradient—the simplest possible, in fact—presented in a variety of differently shaped elements (Figure 8-74):

```
.radial {background-image: radial-gradient(purple, gold);}
```

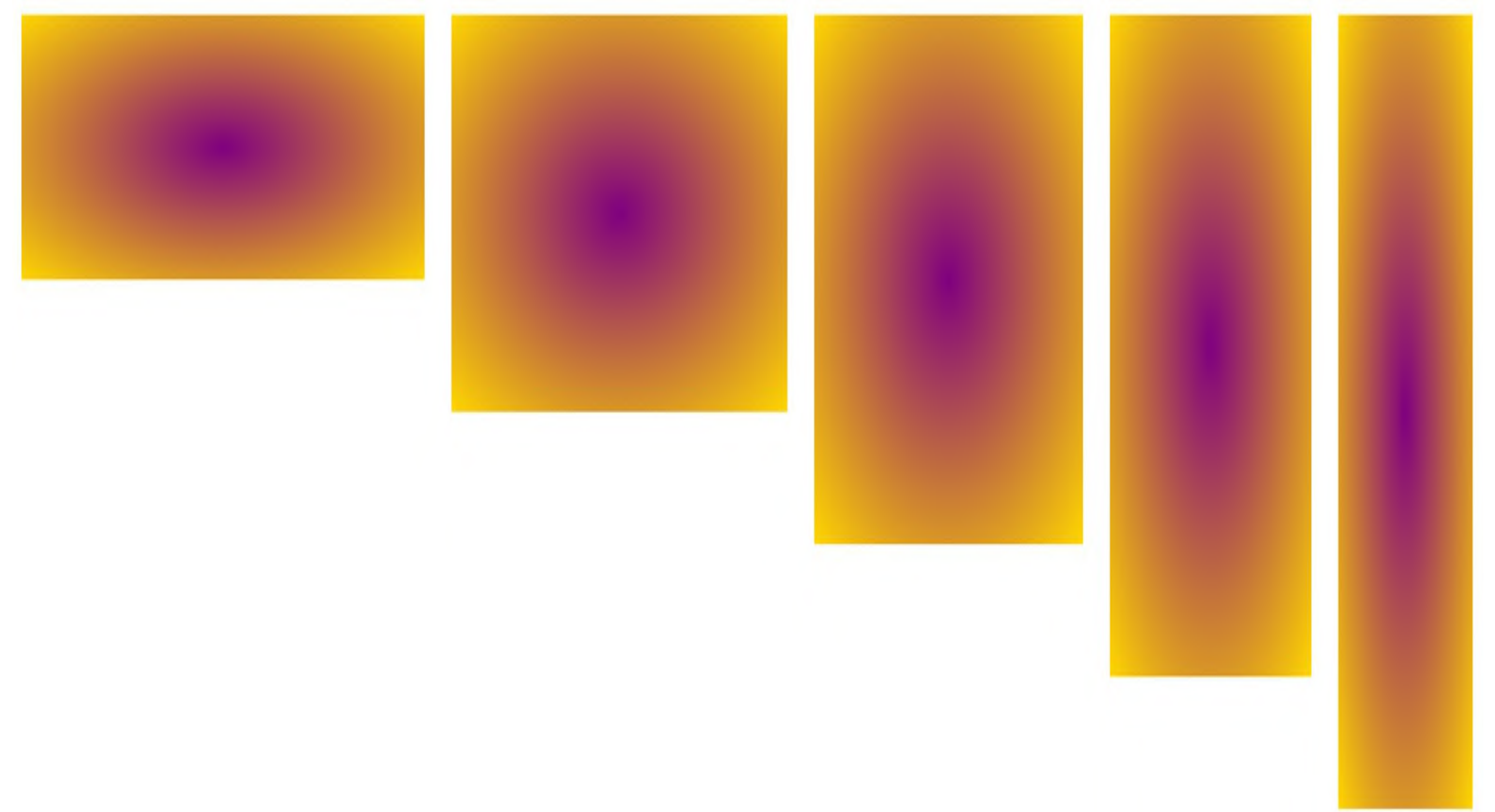


Figure 8-74. A simple radial gradient in multiple settings

In all of these cases, because no position was declared, the default of `center` was used, and the default ellipse has the same aspect ratio as the image size. Because no shape was declared, the shape is an ellipse for all cases but the square element; in that case, the shape is a circle. Finally, because no color-stop or color-hint positions were declared, the first is placed at the beginning of the gradient ray, and the last at the end, with a linear blend from one to the other.

That’s right: the *gradient ray*, which is the radial equivalent to the gradient line in linear gradients. It extends outward from the center of the gradient directly to the right, and the rest of the gradient is constructed from it. (We’ll get to the details on that in just a bit.)

Shape and size

First off, there are exactly two possible shape values (and thus two possible shapes) for a radial gradient: `circle` and `ellipse`. The shape of a gradient can be declared explicitly, or it can be implied by the way you size the gradient image.

So, on to sizing. As always, the simplest way to size a radial gradient is with either one non-negative length (if you’re sizing a circle) or two non-negative lengths (if it’s an ellipse). Say you have this radial gradient:

```
radial-gradient(50px, purple, gold)
```

This creates a circular radial gradient that fades from purple at the center to gold at a distance of 50 pixels from the center. If we add another length, then the shape becomes an ellipse that's as wide as the first length, and as tall as the second length:

```
radial-gradient(50px 100px, purple, gold)
```

These two gradients are illustrated in Figure 8-75.

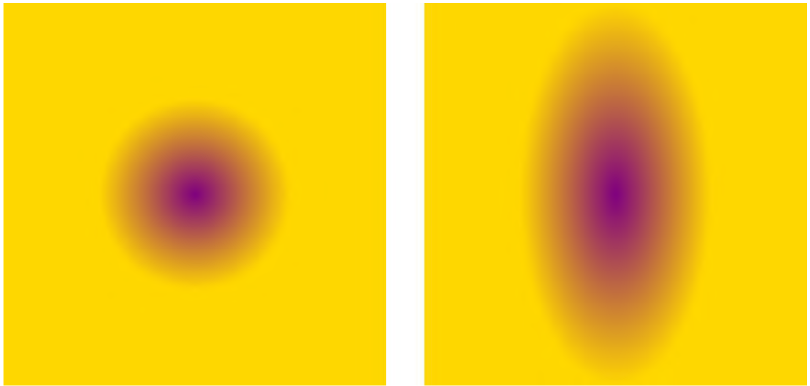


Figure 8-75. Simple radial gradients

Notice how the shape of the gradients has nothing to do with the overall size and shape of the images in which they appear. If you make a gradient a circle, it will be a circle, even if it's inside a rectangular gradient image. So too will an ellipse always be an ellipse, even when inside a square gradient image (where it will look like a circle, since an ellipse with the same height and width forms a circle).

You can also use percentage values for the size, but *only* for ellipses. Circles cannot be given percentage sizes because there's no way to indicate the axis to which that percentage refers. (Imagine an image 100 pixels tall by 500 wide. Should `10%` mean 10 pixels or 50 pixels?) If you try to provide percentage values for a circle, the entire declaration will fail due to the invalid value.

If you do supply percentages to an ellipse, then as usual, the first refers to the horizontal axis and the second to the vertical. The following gradient is shown in various settings in Figure 8-76:

```
radial-gradient(50% 25%, purple, gold)
```

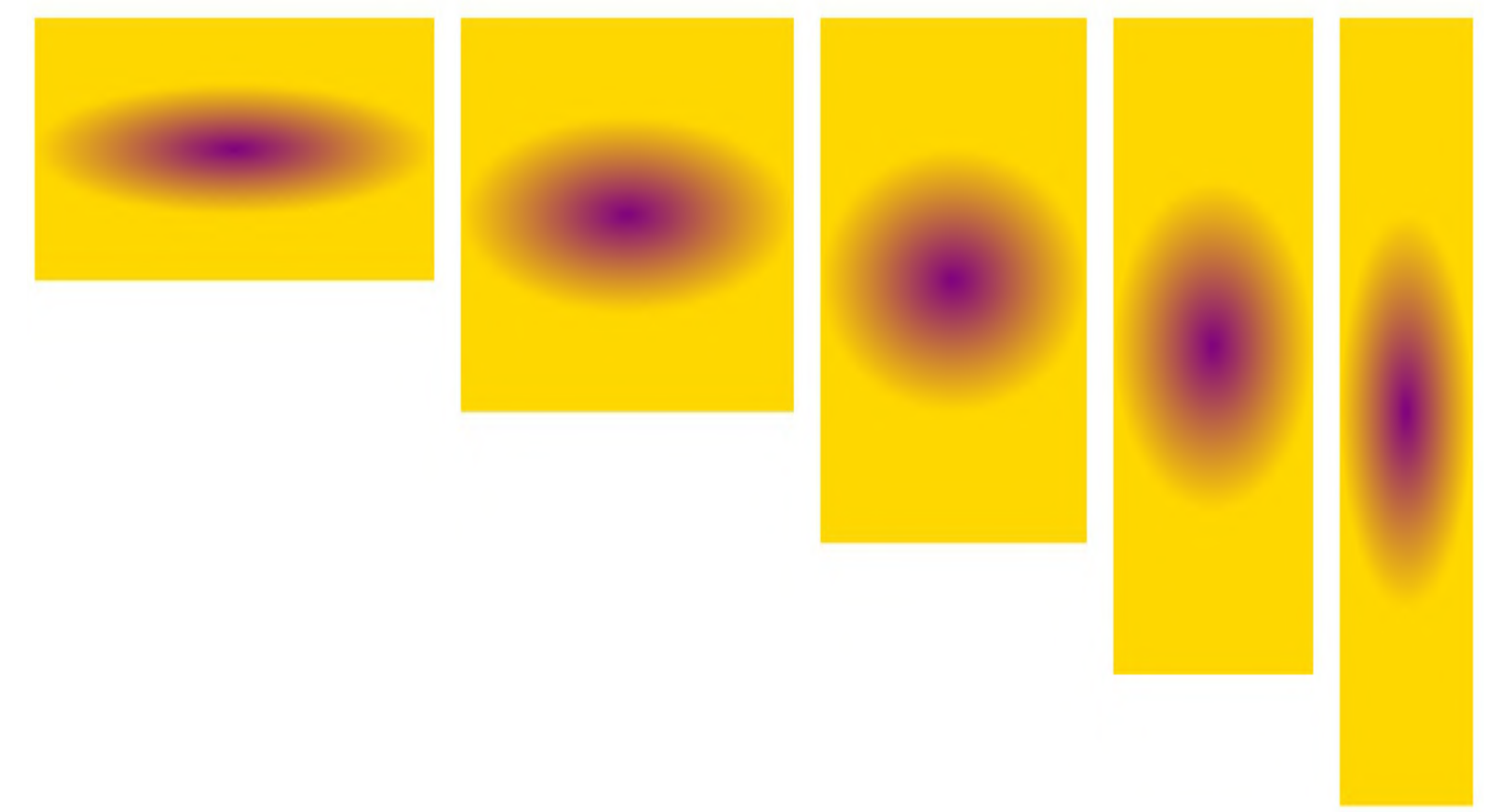


Figure 8-76. Percentage-sized elliptical gradients

When it comes to ellipses, you’re also able to mix lengths and percentages, with the usual caveat to be careful. So if you’re feeling confident, you can absolutely make an elliptical radial gradient 10 pixels tall and half the element width, like so:

```
radial-gradient(50% 10px, purple, gold)
```

As it happens, lengths and percentages aren’t the only way to size radial gradients. In addition to those value types, there are also four keywords available for sizing radial gradients, the effects of which are summarized in Table 8-3.

Table 8-3. Radial gradient sizing keywords

Keyword	Meaning
`closest-side`	If the radial gradient's shape is a circle, the gradient is sized so that the end of the gradient ray exactly touches the edge of the gradient image that is closest to the center point of the radial gradient. If the shape is an ellipse, then the end of the gradient ray exactly touches the closest edge in each of the horizontal and vertical axes.
`farthest-side`	If the radial gradient's shape is a circle, the gradient is sized so that the end of the gradient ray exactly touches the edge of the gradient image that is farthest from the center point of the radial gradient. If the shape is an ellipse, then the end of the gradient ray exactly touches the farthest edge in each of the horizontal and vertical axes.
`closest-corner`	If the radial gradient's shape is a circle, the gradient is sized so that the end of the gradient ray exactly touches the corner of the gradient image that is closest to the center point of the radial gradient. If the shape is an ellipse, then the end of the gradient ray still touches the corner closest to the center, *and* the ellipse has the same aspect ratio that it would have had if `closest-side` had been specified.
`farthest-corner` (default)	If the radial gradient's shape is a circle, the gradient is sized so that the end of the gradient ray exactly touches the corner of the gradient image that is farthest from the center point of the radial gradient. If the shape is an ellipse, then the end of the gradient ray still touches the corner farthest from the center, *and* the ellipse has the same aspect ratio that it would have had if `farthest-side` had been specified. Note: this is the default size value for a radial gradient and so is used if no size values are declared.

In order to better visualize the results of each keyword, see Figure 8-77, which depicts each keyword applied as both a circle and an ellipse.

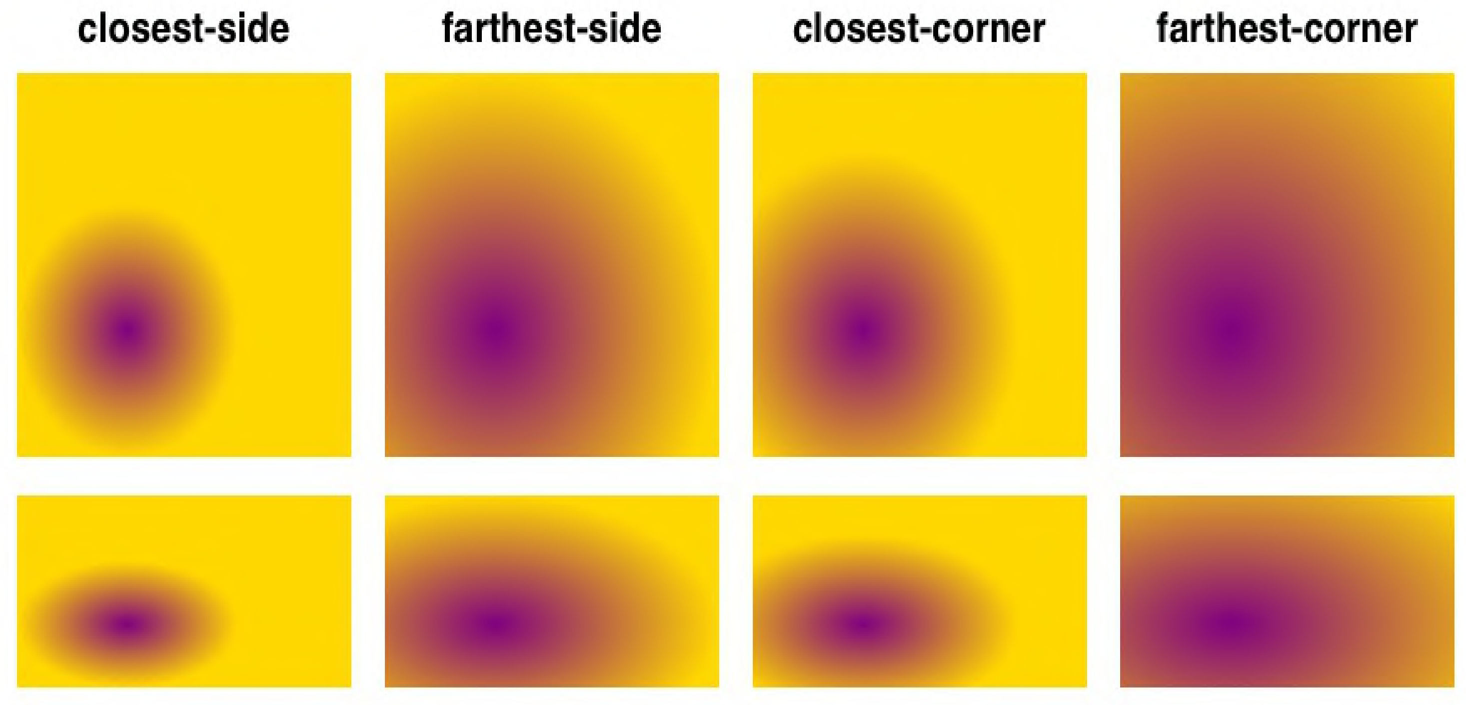


Figure 8-77. The effects of radial gradient sizing keywords

These keywords cannot be mixed with lengths or percentages in elliptical radial gradients; thus, `closest-side 25px` is invalid and will be ignored.

Something you might have noticed in Figure 8-77 is that the gradients didn't start at the center of the image. That's because they were positioned elsewhere, which is the topic of the next section.

Positioning radial gradients

If you want to shift the center of a radial gradient away from the default of `center`, then you can do so using any position value that would be valid for `background-position`. We're not going to reproduce that rather complicated syntax here; flip back to the section on `background-position` ("Background Positioning") if you need a refresher.

When we say "any position value that would be valid," that means any permitted combination of lengths, percentages, keywords, and so on. It also means that if you leave off one of the two position values, it will be inferred just the same as for `background-position`. So, just for one example, `center` is equivalent to `center center`. The one major difference between radial gradient positions and background positions is the default: for radial gradients, the default position is `center`, not `0% 0%`.

To give some idea of the possibilities, consider the following rules, illustrated in Figure 8-78:

```
radial-gradient(at bottom left, purple, gold);
radial-gradient(at center right, purple, gold);
radial-gradient(at 30px 30px, purple, gold);
radial-gradient(at 25% 66%, purple, gold);
radial-gradient(at 30px 66%, purple, gold);
```

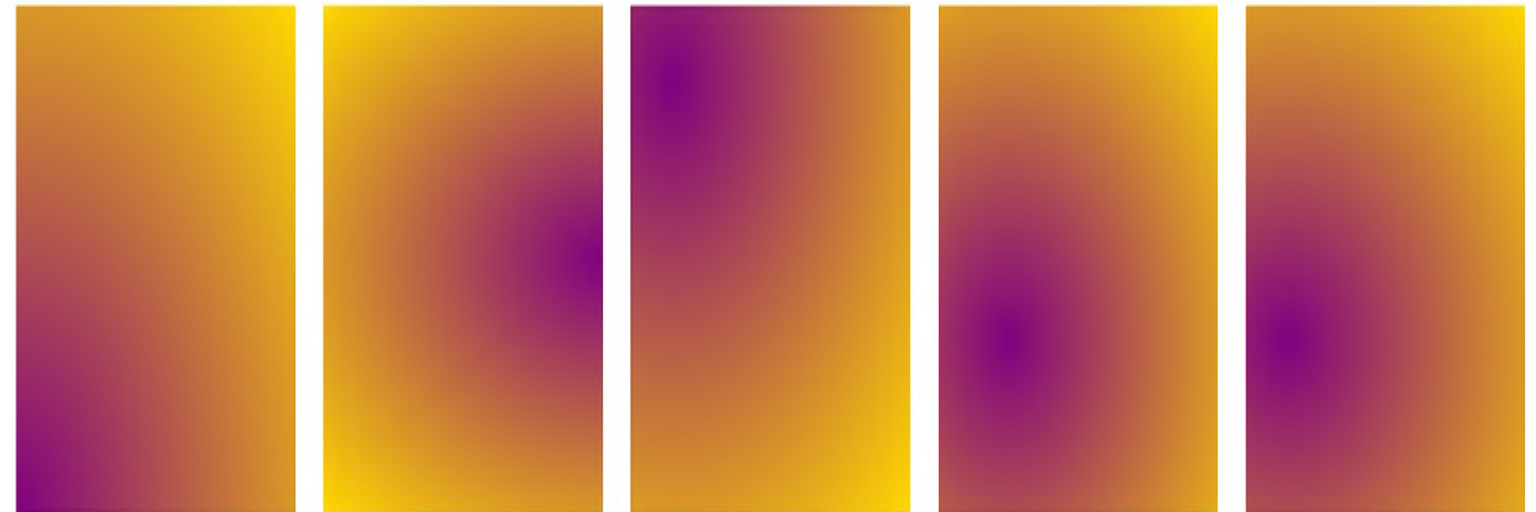


Figure 8-78. Changing the center position of radial gradients

None of those positioned radial gradients were explicitly sized, so they all defaulted to `farthest-corner`. That's a reasonable guess at the intended default behavior, but it's not the only possibility. Let's mix some sizes into the gradients we just saw and find out how that changes things (as depicted in Figure 8-79):

```
radial-gradient(30px at bottom left, purple, gold);
radial-gradient(30px 15px at center right, purple, gold);
radial-gradient(50% 15% at 30px 30px, purple, gold);
radial-gradient(farthest-side at 25% 66%, purple, gold);
radial-gradient(closest-corner at 30px 66%, purple, gold);
```

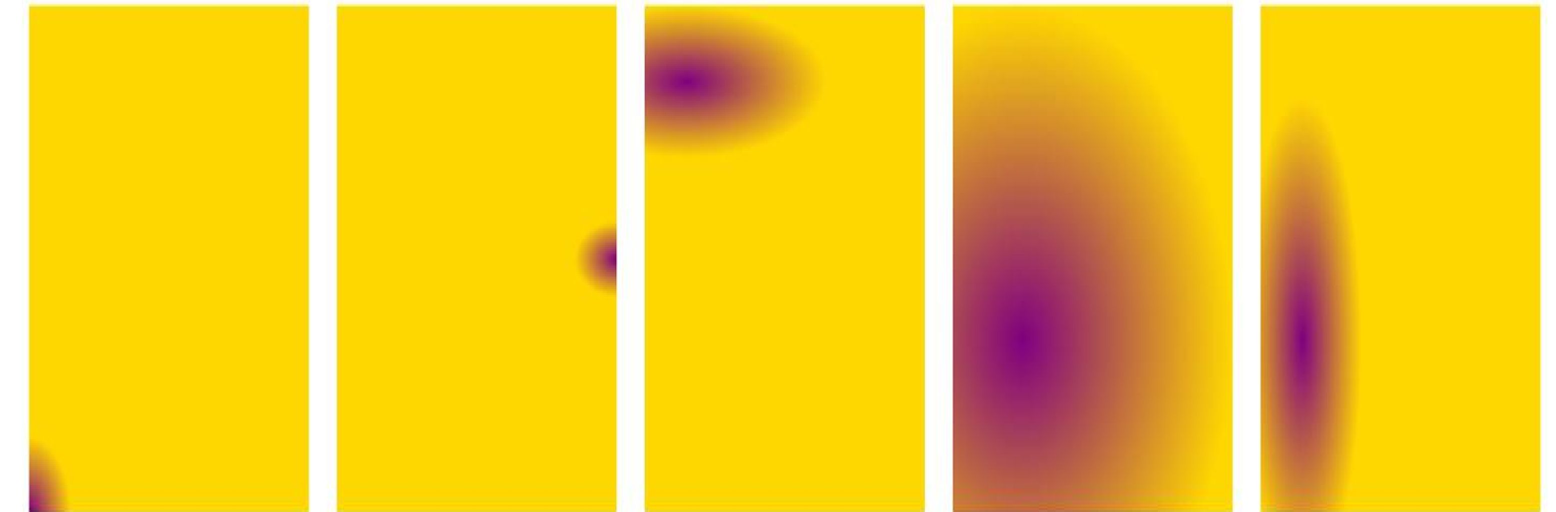


Figure 8-79. Changing the center position of explicitly sized radial gradients

Nifty. Now, suppose we want something a little more complicated than a fade from one color to another. Next stop, color stops!

Radial color stops and the gradient ray

Color stops for radial gradients have the same syntax, and work in a similar fashion, to linear gradients. Let’s return to the simplest possible radial gradient and follow it with a more explicit equivalent:

```
radial-gradient(purple, gold);
radial-gradient(purple 0%, gold 100%);
```

So the gradient ray extends out from the center point. At 0% (the start point, and also the center of the gradient), the ray will be purple. At 100% (the ending point), the ray will be gold. Between the two stops is a smooth blend from purple to gold; beyond the ending point, solid gold.

If we add a stop between purple and gold, but don’t give it a position, then it will be placed midway between them, and the blending will be altered accordingly, as shown in Figure 8-80:

```
radial-gradient(100px circle at center, purple 0%, green, gold 100%);
```

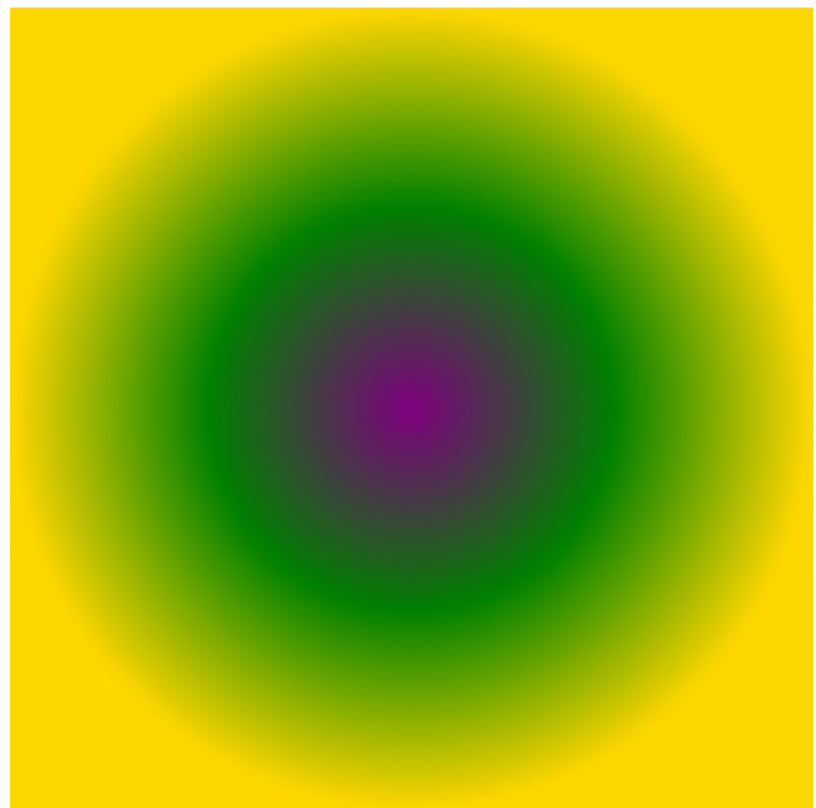


Figure 8-80. Adding a color stop

We’d have gotten the same result if we’d added `green 50%` there, but you get the idea. The gradient ray’s color goes smoothly from purple to green to gold, and then is solid gold beyond that point on the ray.

This illustrates one difference between gradient lines (for linear gradients) and gradient rays: a linear gradient is derived by extending the color at each point along the gradient line off perpendicular to the gradient line. A similar behavior occurs with a radial gradient, except in that case, they aren’t lines that come off the gradient ray. Instead, they are ellipses that are scaled-up or scaled-down versions of the ellipse at the ending point. This is illustrated in Figure 8-81, where an ellipse shows its gradient ray and then the ellipses that are drawn at various points along that ray.

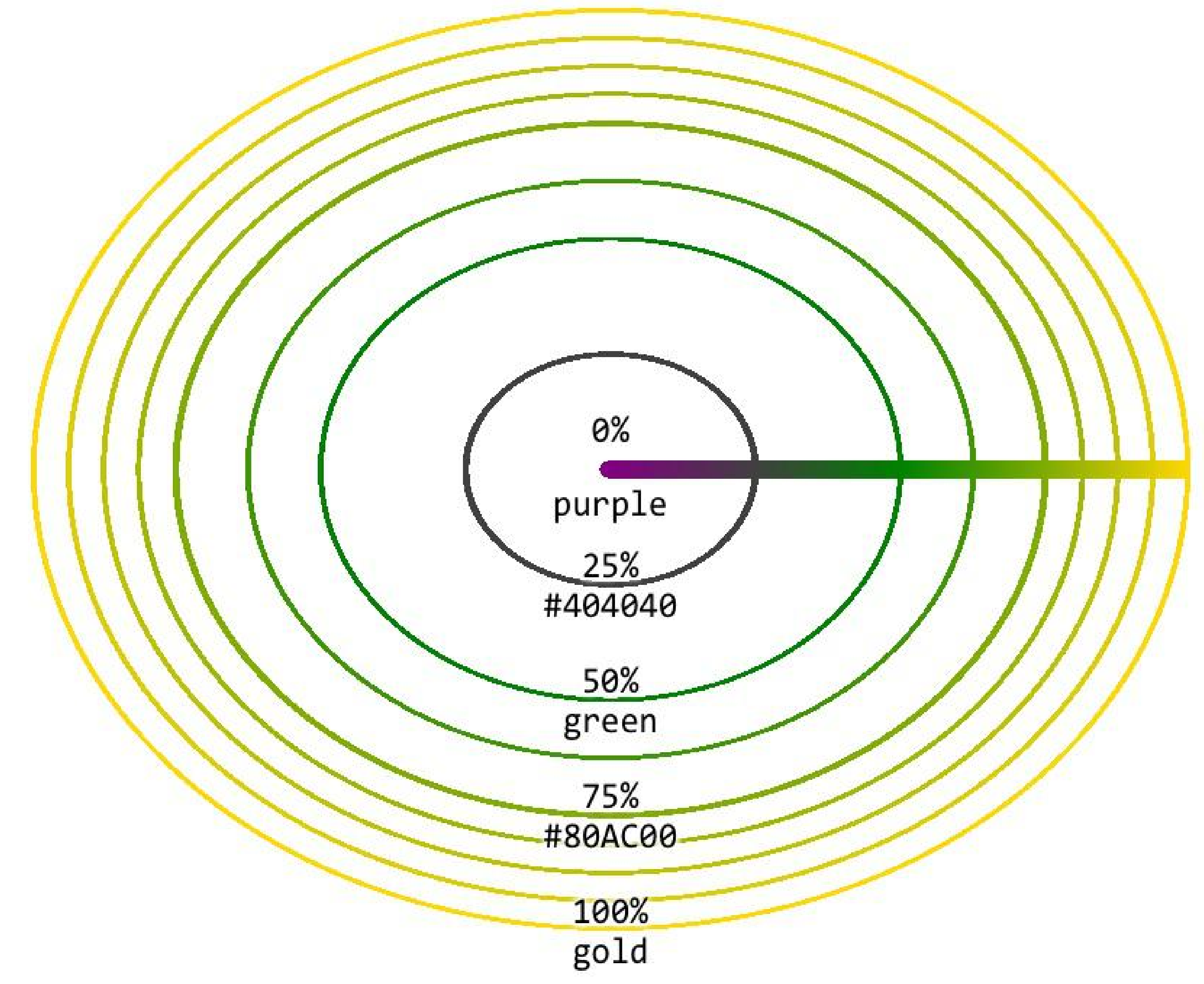


Figure 8-81. The gradient ray and some of the ellipses it spawns

That brings up an interesting question: how is the ending point (the `100%` point, if you like) determined for each gradient ray? It's the point where the gradient ray intersects with the shape described by the size. In the case of a circle, that's easy: the gradient ray's ending point is however far from the center that the size value indicates. So for a `25px circle` gradient, the ending point of the ray is 25 pixels from the center.

For an ellipse, it's essentially the same operation, except that the distance from the center is dependent on the horizontal axis of the ellipse. Given a radial gradient that's a `40px 20px ellipse`, the ending point will be 40 pixels from the center and directly to its right. Figure 8-82 shows this in some detail.

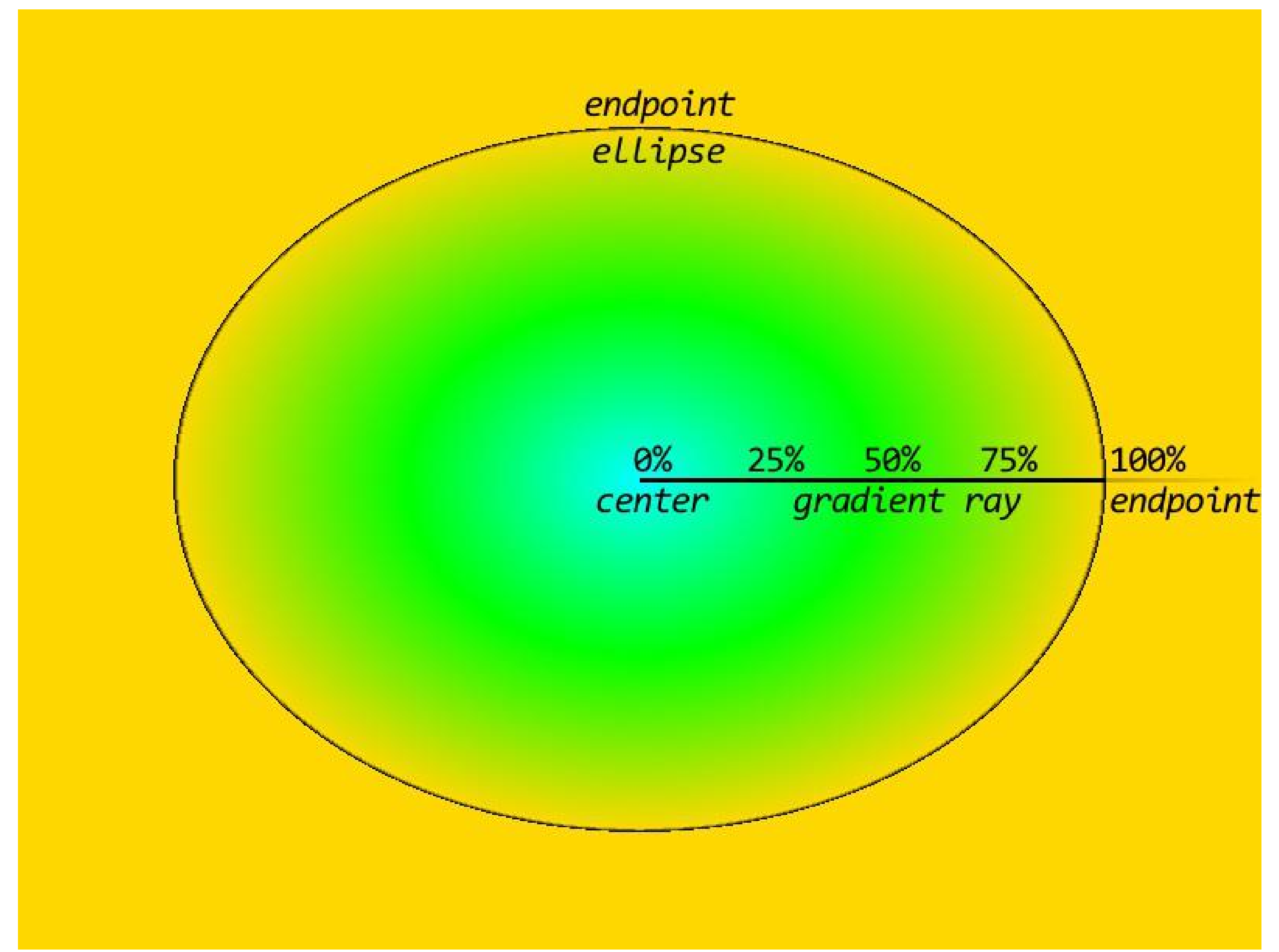


Figure 8-82. Setting the gradient ray's ending point

Another difference between linear gradient lines and radial gradient rays is that you can see beyond the ending point. If you recall, a linear gradient line is always drawn so that you can see the colors at the 0% and 100% points, but nothing beyond them; the gradient line can never be any smaller than the longest axis of the gradient image, and will frequently be longer than that. With a radial gradient, on the other hand, you can size the radial shape to be smaller than the total gradient image. In that case, the color at the last color stop is extended outward from the ending point. (We've already seen this in several previous figures.)

Conversely, if you set a color stop that's beyond the ending point of a ray, you might get to see the color out to that stop. Consider the following gradient, illustrated in Figure 8-83:

```
radial-gradient(50px circle at center, purple, green, gold 80px)
```

Figure 8-83. Color stops beyond the ending point

The first color stop has no position, so it’s set to `0%`, which is the center point. The last color stop is set to `80px`, so it will be 80 pixels away from the center in all directions. The middle color stop, `green`, is placed midway between the two (40 pixels from the center). So we get a gradient that goes out to gold at 80 pixels and then continues gold beyond that point.

This happens even though the circle was explicitly set to be 50 pixels large. It still is 50 pixels in radius, it’s just that the positioning of the last color stop makes that fact vaguely irrelevant. Visually, we might as well have declared this:

```
radial-gradient(80px circle at center, purple, green, gold)
```

or, more simply, just this:

```
radial-gradient(80px, purple, green, gold)
```

The same behaviors apply if you use percentages for your color stops. These are equivalent to the previous examples, and to each other, visually speaking:

```
radial-gradient(50px, purple, green, gold 160%)
radial-gradient(80px, purple, green, gold 100%)
```

Now, what if you set a negative position for a color stop? It’s pretty much the same result as we saw with linear gradient lines: the negative color stop is used to figure out the color at the starting point, but is otherwise unseen. Thus, the following gradient will have the result shown in [Figure 8-84](#):

```
radial-gradient(80px, purple -40px, green, gold)
```

Figure 8-84. Handling a negative color-stop position

Given these color-stop positions, the first color stop is at `-40px`, the last is at `80px` (because, given its lack of an explicit position, it defaults to the ending point), and the middle is placed midway between them. The result is the same as if we'd explicitly said:

```
radial-gradient(80px, purple -40px, green 20px, gold 80px)
```

That's why the color at the center of the gradient is a green-purple: it's a blend of one-third purple, two-thirds green. From there, it blends the rest of the way to green, and then on to gold. The rest of the purple-green blend, the part that sits on the "negative space" of the gradient ray, is invisible.

Degenerate cases

Given that we can declare size and position for a radial gradient, the question arises: what if a circular gradient has zero radius, or an elliptical gradient has zero height or width? These conditions aren't quite as hard to create as you might think: besides explicitly declaring that a radial gradient has zero size using `0px` or `0%`, you could also do something like this:

```
radial-gradient(closest-corner circle at top right, purple, gold)
```

The gradient's size is set to `closest-corner`, and the center has been moved into the `top right` corner, so the closest corner is zero pixels away from the center. Now what?

In this case, the specification very explicitly says that the gradient should be rendered as if it's "a circle whose radius [is] an arbitrary very small number greater than zero." So that might mean as if it had a radius of one-one-billionth of a pixel, or a picometer, or heck, the Planck length. The interesting thing is that it means the gradient is still a circle. It's just a very, very, very small circle. Probably, it will be too small to actually render anything visible. If so, you'll just get a solid-color fill that corresponds to the

color of the last color stop instead.

Ellipses with zero-length dimensions have fascinatingly different defined behaviors. Let's assume the following:

```
radial-gradient(0px 50% at center, purple, gold)
```

The specification states that any ellipse with a zero width is rendered as if it's "an ellipse whose height [is] an arbitrary very large number and whose width [is] an arbitrary very small number greater than zero." In other words, render it as though it's a linear gradient mirrored around the vertical axis running through the center of the ellipse. The specification also says that in such a case, any color stops with percentage positions resolve to `0px`. This will usually result in a solid color matching the color defined for the last color stop.

On the other hand, if you use lengths to position the color stops, you can get a vertically mirrored horizontal linear gradient for free. Consider the following gradient, illustrated in Figure 8-85:

```
radial-gradient(0px 50% at center, purple 0px, gold 100px)
```

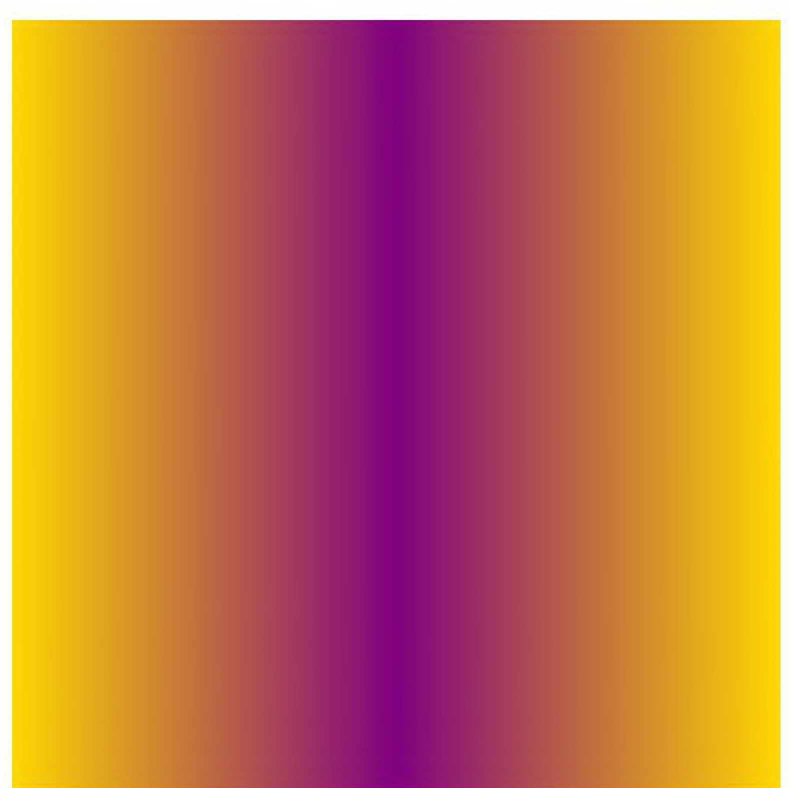


Figure 8-85. The effects of a zero-width ellipse

How did this happen? First, remember that the specification says that the `0px` horizontal width is treated as if it's a tiny non-zero number. For the sake of illustration, let's suppose that's one-one-thousandth of a pixel (0.001 px). That means the ellipse shape is a thousandth of a pixel wide by half the height of the image. Again for the sake of illustration, let's suppose that's a height of 100 pixels. That means the first ellipse shape is a thousandth of a pixel wide by 100 pixels tall, which is an aspect ratio of 0.001:100, or 1:100,000.

OK, so every ellipse drawn along the gradient ray has a 1:100,000 aspect ratio. That means the ellipse at

half a pixel along the gradient ray is 1 pixel wide and 100,000 pixels tall. At 1 pixel, it's 2 pixels wide and 200,000 pixels tall. At 5 pixels, the ellipse is 10 pixels by a million pixels. At 50 pixels along the gradient ray, the ellipse is 100 pixels wide and 10 million tall. And so on. This is diagrammed in Figure 8-86.

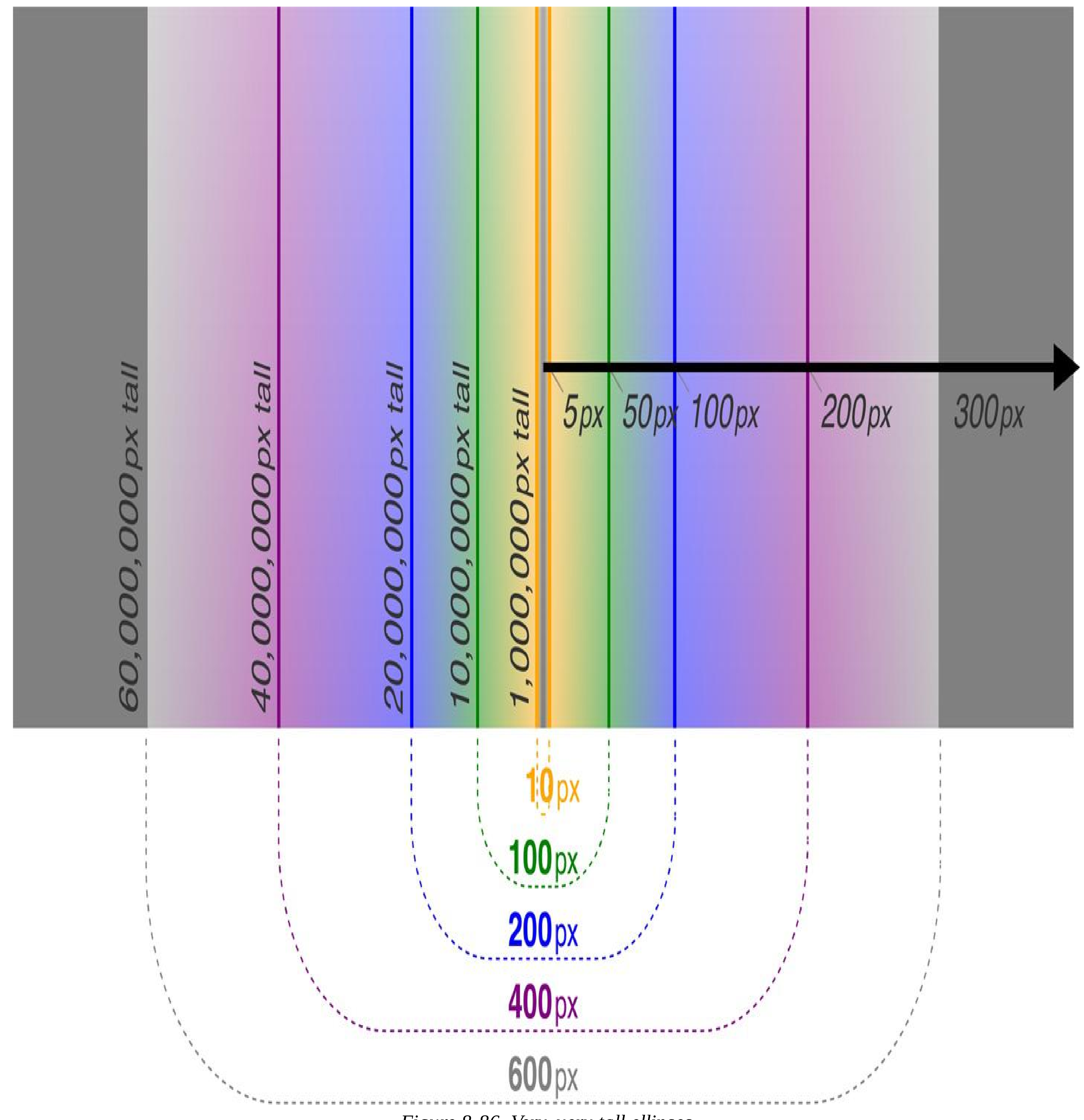


Figure 8-86. Very, very tall ellipses

So you can see why the visual effect is of a mirrored linear gradient. These ellipses are effectively drawing vertical lines. Technically they aren't, but in practical terms they are. The result is as if you have a vertically mirrored horizontal gradient, because each ellipse is centered on the center of the gradient, and both sides of it get drawn. While this may be a radial gradient, we can't see its radial nature.

On the other hand, if the ellipse has width but not height, the results are quite different. You'd think the result would be to have a vertical linear gradient mirrored around the horizontal axis, but not so! Instead, the result is a solid color equal to the last color stop. (Unless it's a repeating gradient, a subject we'll turn to shortly, in which case it should be a solid color equal to the average color of the gradient.) So, given either of the following, you'll get a solid gold:

```
radial-gradient(50% 0px at center, purple, gold)
radial-gradient(50% 0px at center, purple 0px, gold 100px)
```

Why the difference? It goes back to how radial gradients are constructed from the gradient ray. Again, remember that, per the specification, a zero distance here is treated as a very small non-zero number. As before, we'll assume that `0px` is reassigned to `0.001px`, and that the `50%` evaluates to 100 pixels. That's an aspect ratio of 100:0.001, or 100,000:1.

So, to get an ellipse that's 1 pixel tall, the width of that ellipse must be 100,000 pixels. But our last color stop is only at 100 pixels! At that point, the ellipse that's drawn is 100 pixels wide and 1,000th of a pixel tall. All of the purple-to-gold transition that happens along the gradient ray has to happen in that thousandth of a pixel. Everything after that is gold, as per the final color stop. Thus, we can only see the gold.

You might think that if you increased the position value of the last color stop to `100000px`, you'd see a thin sliver of purple-ish color running horizontally across the image. And you'd be right, *if* the browser treats `0px` as `0.001px` in these cases. If it assumes `0.00000001px` instead, you'd have to increase the color stop's position a *lot* further in order to see anything. And that's assuming the browser was actually caulculating and drawing all those ellipses, instead of just hard-coding the special cases. The latter is a lot more likely, honestly. It's what we'd do if we were in charge of a browser's gradient-rendering code.

And what if an ellipse has zero width *and* zero height? In that case, the specification is written such that the zero-width behavior is used; thus, you'll get the mirrored-linear-gradient behavior.

NOTE

As of late 2022, browser support for the defined behavior in these edge cases was unstable, at best. Some browsers used the last color-stop's color in all cases, and others refused to draw a gradient at all in some cases.

Repeating radial gradients

While percentages in repeating linear gradients could turn them into non-repeating gradients, percentages can be very useful with repeating radial gradients, where the size of the circle or ellipse is defined, percentage positions along the gradient ray are defined, and you can see beyond the endpoint of the gradient ray. For example, assume:

```
.allhail {background:
    repeating-radial-gradient(100px 50px, purple, gold 20%, green 40%,
                              purple 60%, yellow 80%, purple);}
```

Given this rule, there will be a color stop every 20 pixels, with the colors repeating in the declared pattern. Because the first and last color stops have the same color value, there is no hard color switch. The ripples just spread out forever, or at least until they're beyond the edges of the gradient image. See Figure 8-87 for an example.

Figure 8-87. Repeating radial gradients

Just imagine what that would look like with a repeating radial gradient of a rainbow!

```
.wdim {background:
    repeating-radial-gradient(
        100px circle at bottom center,
        rgb(83%,83%,83%) 50%,
        violet 55%, indigo 60%, blue 65%, green 70%,
        yellow 75%, orange 80%, red 85%,
        rgb(47%,60%,73%) 90%
    );}
```

There are a couple of things to keep in mind when creating repeating radial gradients:

- If you don't declare size dimensions for a radial, it will default to an ellipse that has the same height-to-width ratio as the overall gradient image; *and*, if you don't declare a size for the image with `background-size`, the gradient image will default to the height and width of the element background where it's being applied. (Or, in the case of being used as a list-style bullet, the size that the browser gives it.)

- The default radial size value is `farthest-corner`. This will put the endpoint of the gradient ray far enough to the right that its ellipse intersects with the corner of the gradient image that's furthest from the center point of the radial gradient.

These are reiterated here to remind you that if you stick to the defaults, there's not really any point to

having a repeating gradient, since you’ll only be able to see the first iteration of the repeat. It’s only when you restrict the initial size of the gradient that the repeats become visible.

Conic Gradients

Radial gradients are fun, but what if you want a gradient that wraps around a central point, similar to a color hue wheel? That’s what CSS calls a *conic gradient*, which can be thought of as a concentric series of linear gradients that are bent into circles. Looked at another way, at any distance from the center, there’s a circle whose outer rim could be straightened out into a linear gradient with the color stop specified.

Conic gradients are more easily shown than described, so consider the following CSS, which is illustrated in Figure 8-88 along with a linear diagram to show how the stops wrap around the conical space:

```
background:
     conic-gradient(
          black, gray, black, white, black, silver, gray
     );
```

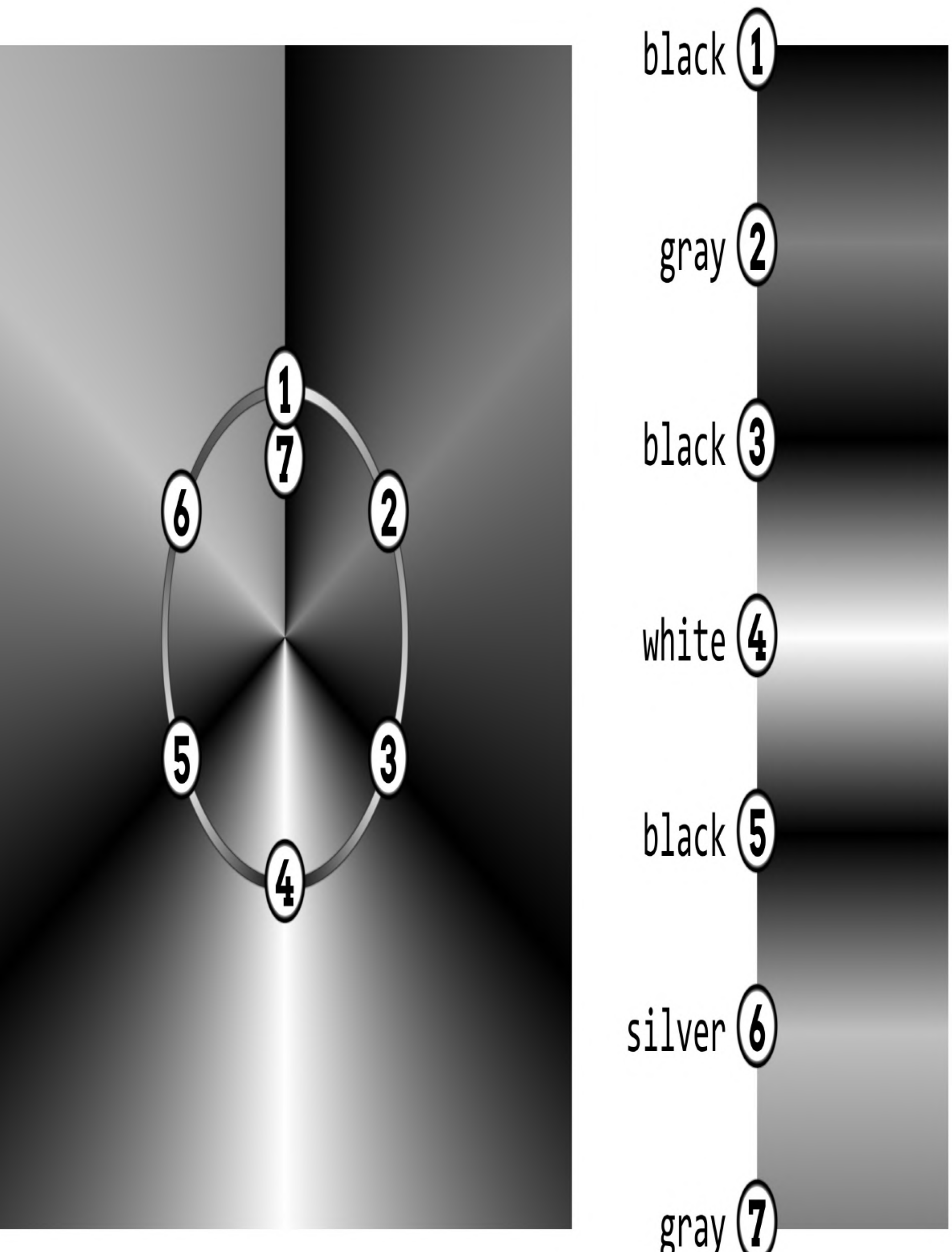

1
7
6
2
5
3
4
black 1
gray 2
black 3
white 4
black 5
silver 6
gray 7

Figure 8-88. A simple conic gradient and its linear equivalent

Note how each of the color stops is labeled on the linear gradient: the circled numbers listed there are repeated in the conic gradient, to show where each color stop falls. At 60 degrees around the conic gradient, there is a `gray` color stop. At 180 degrees, a `white` color stop. At the top of the conic gradient, the `0deg` and `360deg` points meet, so `black` and `gray` sit next to each other.

By default, conic gradients start at zero degrees, using the same compass degree system that transforms and other parts of CSS use, so `0deg` is at the top. If you want to start from a different angle and wrap around the circle back to that point, it's as straightforward as adding `from` and an angle value to the front of the `conic-gradient` value, which rotates the entire gradient by the declared angle. The following would all have the same result.

```
conic-gradient(from 144deg, black, gray, black, white)
conic-gradient(from 2.513274rad, black, gray, black, white)
conic-gradient(from 0.4turn, black, gray, black, white)
```

If the conic gradient has been given a different start angle, such as `from 45deg`, then it acts as a rotation of the entire conic gradient. Consider the following two examples, with the results depicted in Figure 8-89.

```
conic-gradient(black, white 90deg, gray 180deg, black 270deg, white)
conic-gradient(from 45deg, black, white 90deg, gray 180deg, black 270deg, white)
```

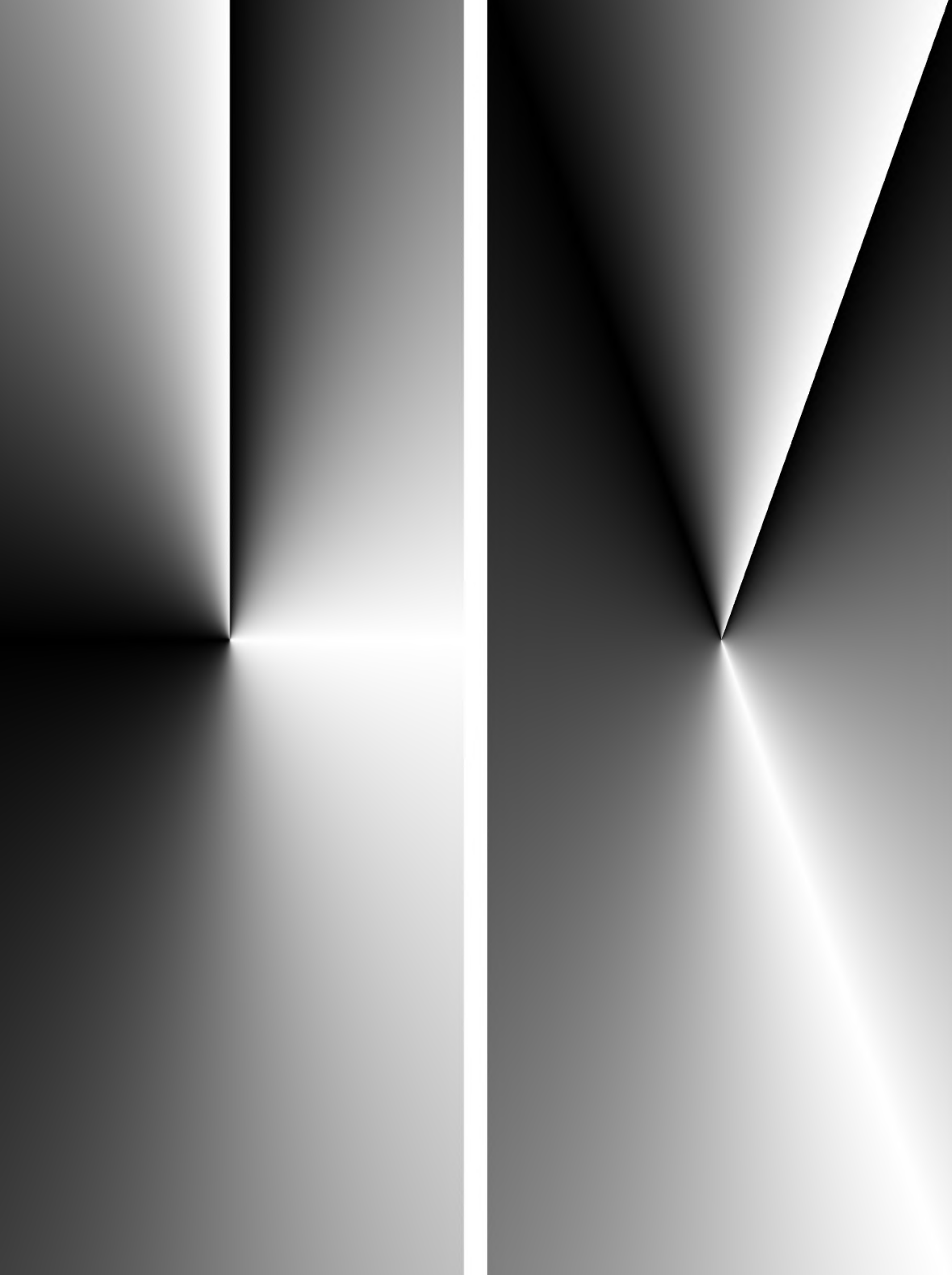


Figure 8-89. Conic gradients with angled color stops and different start angles

Not only is the starting point rotated 45 degrees, but all the other color stops are as well. Thus, even though the first color stop has an angle of `90deg`, it actually occurs at the 135-degree mark, that being 90 degrees with a 45-degree rotation added.

It's also possible to change the location of the gradient's centerpoint within the image, just as with radial gradients. The syntax is quite similar, as you can see in this code block (illustrated by Figure 8-90):

```
conic-gradient(from 144deg at 3em 6em, black, gray, black, white)
conic-gradient(from 144deg at 67% 25%, black, gray, black, white)
conic-gradient(from 144 deg at center bottom, black, gray, black, white)
```

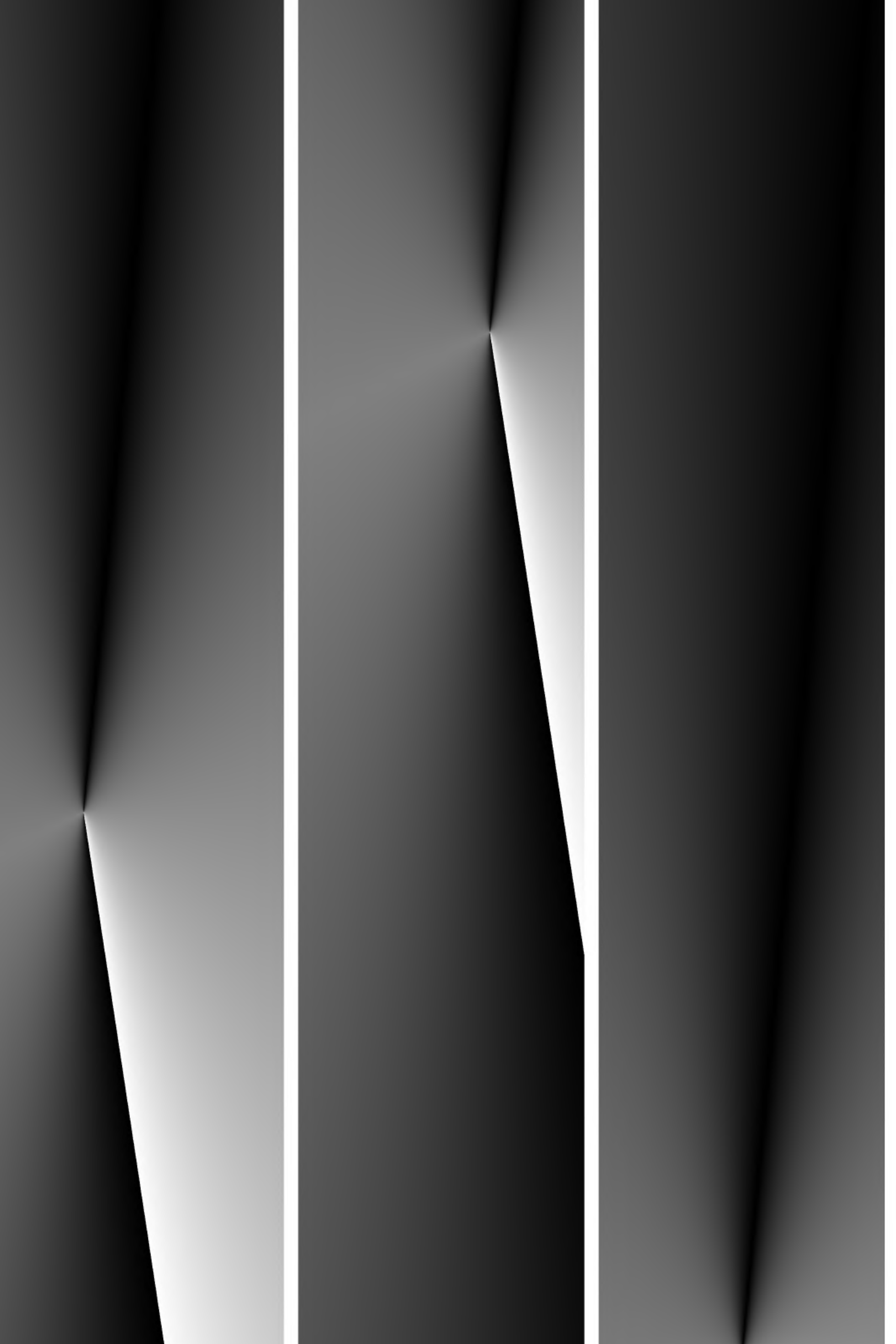


Figure 8-90. Rotated and offset conic gradients

In the first of the three examples, the center of the conic gradient is placed `3em` to the right of the top left corner, and `6em` down from that same corner. Similarly, the second example shows the centerpoint `67%` of the way across the conic-gradient image, and `25%` down from the top.

The third example shows what happens when the centerpoint of a conic gradient is placed along one edge of the image: we only see half (at most) of the gradient. In this case, the top half is visible—that is, the colors from 270 degrees through 90 degrees.

So all together, the syntax for a conic gradient is:

```
conic-gradient(
    [ from <angle>]? [ at <position>]? , | at <position>, ]?
      <color-stop> , [ <color-hint>]? , <color-stop> ]+
)
```

If the `from` angle is not given, it defaults to `0deg`. If not `at` position is given, then it defaults to `50% 50%` (that is, the center of the conic-gradient image).

Much as with radial and linear gradients, color stop distances can be specific by a percentage value; in this case, it resolves to an angle value. Thus, for a conic gradient starting at 0 degrees, the color stop distance `25%` would resolve to 90 degrees, as 90 is 25% of 360. Conic color stops can also be specified as a degree value, as shown previously.

You *cannot* specify a length value for a conic gradient's color stop's distance. Only percentages and angles are acceptable, and they can be mixed.

Conic color stops

If you want a conic gradient to blend smoothly from color to color all the way around the circle, then it is necessary to make the last color stop match the first color stop. Otherwise, you'll see the kinds of hard transitions seen in earlier examples. If you wanted to create a color hue wheel, for example, you'd need to declare it like so:

```
conic-gradient(red, magenta, blue, aqua, lime, yellow, red)
```

Except that's not actually a wheel, since the conic-gradient image fills the entire background area, and background areas in CSS are (thus far) rectangular by default. To make the color wheel actually look like a color wheel, you'd need to either use a circular clipping path (see XREF HERE) or round the corners on a square element (see Chapter 7). For example, the following will have the result shown in Figure 8-91.

```
.hues {
     height: 10em; width: 10em;
     background: conic-gradient(red, magenta, blue, aqua, lime, yellow, red);
}
#wheel {
     border-radius: 50%;
```

```
}

<div class="hues"></div>
<div class="hues" id="wheel"></div>
```

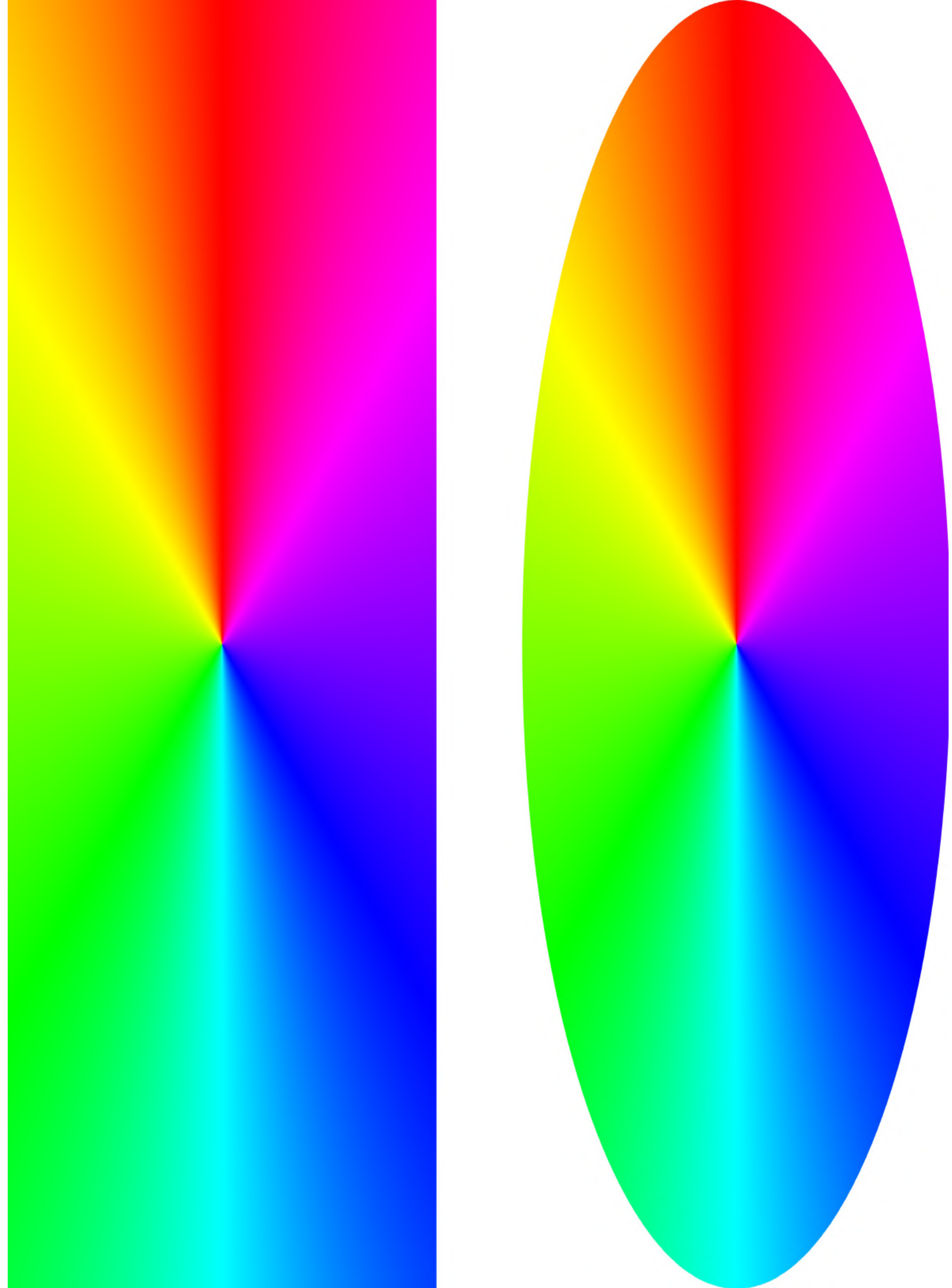


Figure 8-91. Hue-wheel conic gradients with and without corner rounding

This emphasizes that while it's easy to think of conic gradients as circles, the end result is a rectangle, absent any clipping or other effort to make the element's background area non-rectangular. So if you're thinking about using conic gradients to make, say, a pie chart, you'll have to do more than just define a conic gradient with hard stops.

Just as we used two length-percentage values to create hard stops in linear gradients, we can use two hard stops in conic gradients. For example:

```
conic-gradient(
        green 37.5%,
        yellow 37.5% 62.5%,
        red 62.5%);
```

In this syntax, a given color stop can be written as *<color> <beginning> <ending>*, where *<beginning>* and *<ending>* are percentage or angle values.

If you want to create smoother transitions between colors but still have them be mostly solid, then the *<color> <beginning> <ending>* syntax can help a lot. For example, the following conic gradient eases the transitions between green, yellow, and red without making the overall gradient too "smeared."

```
conic-gradient(green 35%, yellow 40% 60%, red 65%);
```

This runs a solid wedge of green from zero to 126 degrees (35%), then transitions smoothly from green to yellow between 126 degrees and 144 degrees (40%), past which there is a solid wedge of yellow spanning from 144 degrees to 216 degrees (60%). Similarly, there is a smoothed transition from yellow to red between 216 degrees and 234 degrees (65%), and beyond that, a solid red wedge running to 360 degrees.

All this is illustrated in Figure 8-92, with extra annotations to mark where the calculated angles land.

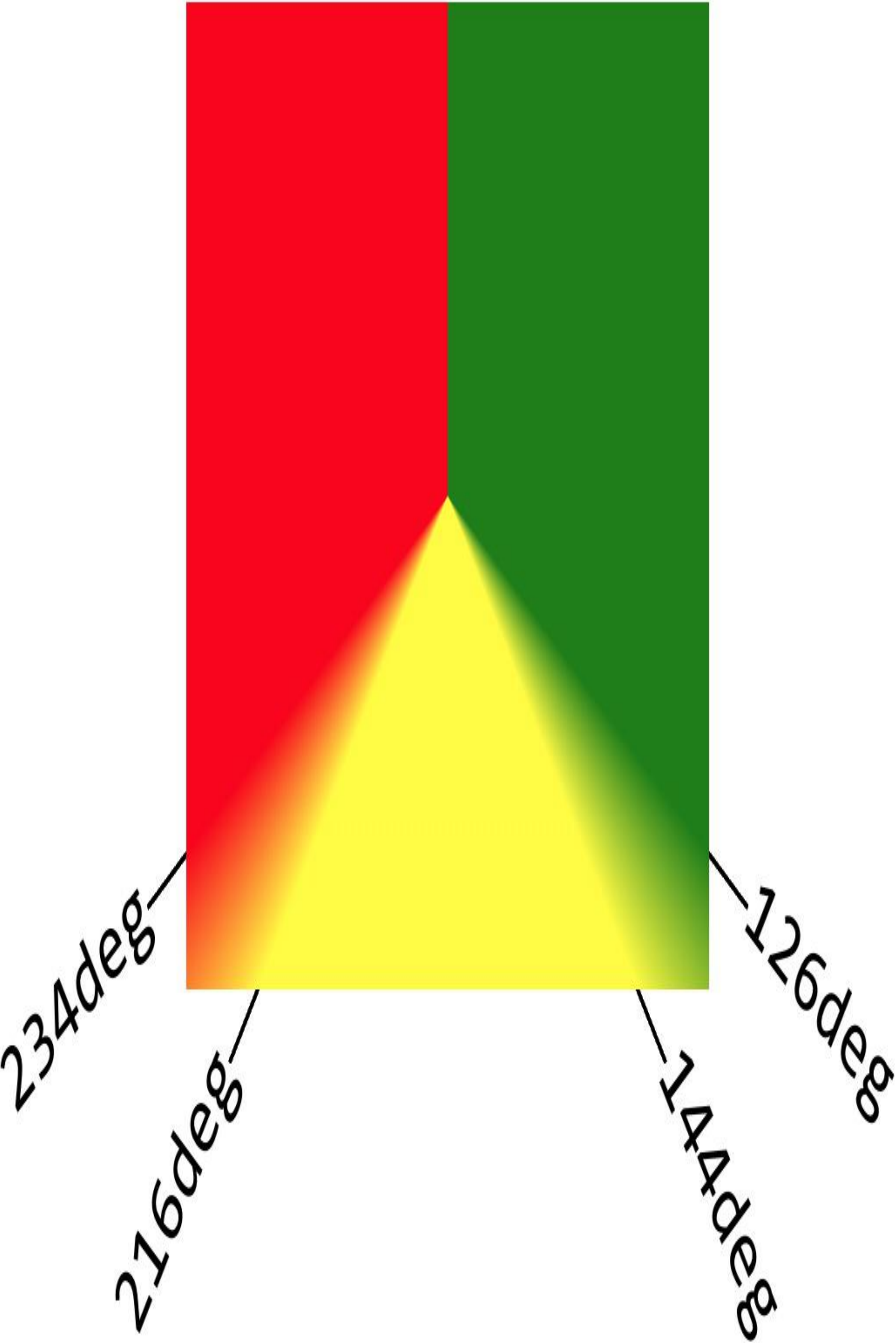


Figure 8-92. Conic gradients with solid color wedges and smooth transitions

And, as it happens, that syntax makes it relatively easy to recreate those picnic tablecloths discussed earlier in the chapter using a conic gradient:

```
background-image: conic-gradient(
        rgba(0 0 0 / 0.2) 0% 25%,
        rgba(0 0 0 / 0.4) 25% 50%,
        rgba(0 0 0 / 0.2) 50% 75%,
        transparent 75% 100%
        );
background-size: 2vw 2vw;
```

```
background-repeat: repeat;
```

This creates, in a single gradient image, a set of four squares in the pattern. That image is then sized and repeated. It's not more efficient or elegant than using repeating linear gradients, but it does embody a certain cleverness that appeals to us.

Repeating conic gradients

And now we come to repeating conic gradients, which are highly useful if you want to create a starburst pattern, or even something simple like a checkerboard pattern. For example:

```
conic-gradient(
    #0002 0 25%, #FFF2 0 50%, #0002 0 75%, #FFF2 0 100%
    )
```

This sets up a checkerboard pattern with four color stops, but only two colors. We can restate that using `repeating-conic-gradient` like so, with new colors to make the pattern a little more clear:

```
repeating-conic-gradient(
     #343 0 25%, #ABC 0 50%
     )
```

All that was necessary in this simple repeating case was to set up the first two color stops. After that, the stops are repeated until the full 360 degrees of the conic are filled, as shown in Figure 8-93.

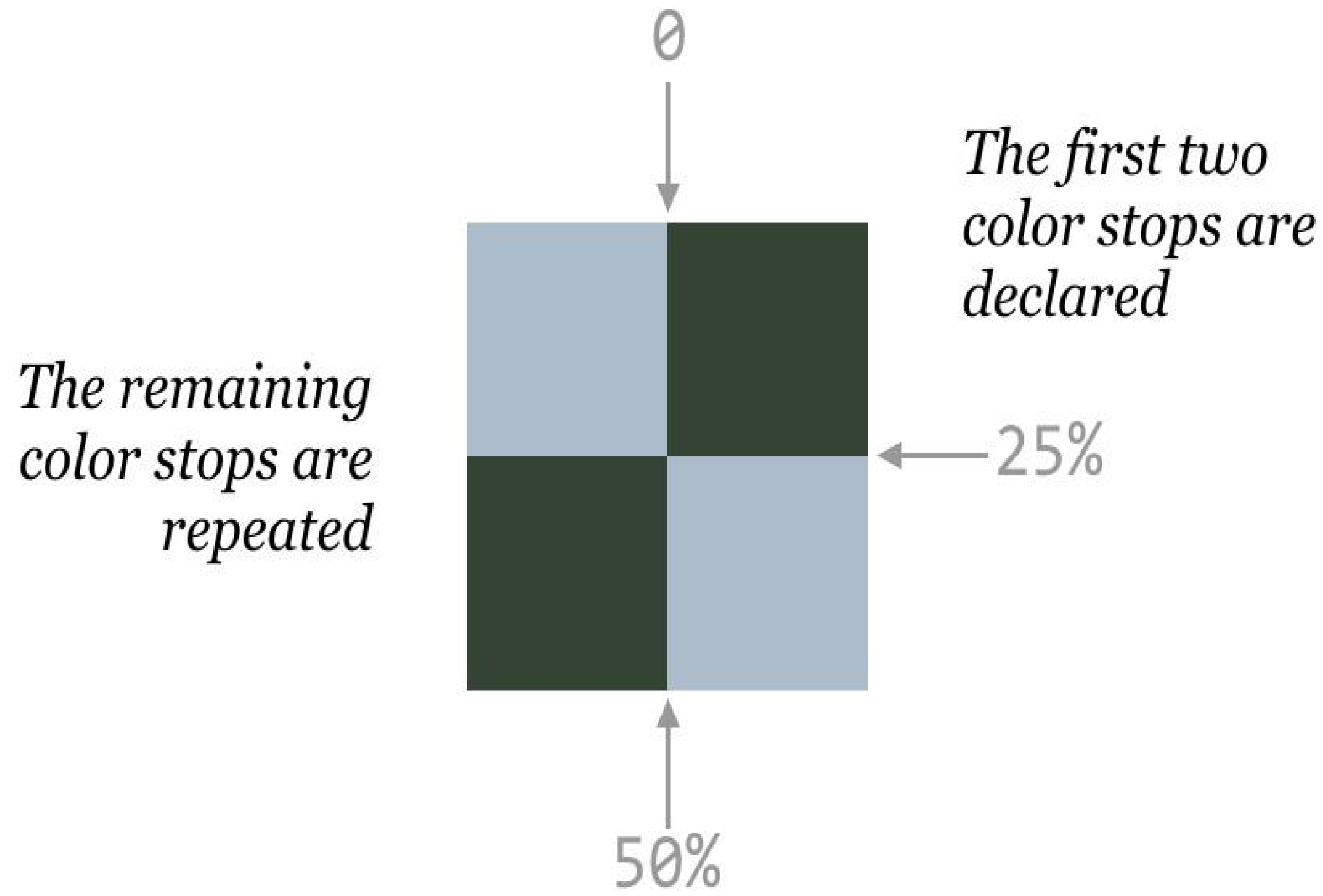


Figure 8-93. A repeating conic gradient

This means we can create wedges of any size, with any transition, and repeat them all the way around the conic circle. Here are just three examples, rendered in Figure 8-94.

```
repeating-conic-gradient(#117 5deg, #ABE 15deg, #117 25deg)
repeating-conic-gradient(#117 0 5deg, #ABE 0 15deg, #117 0 25deg)
repeating-conic-gradient(#117 5deg, #ABE 15deg)
```

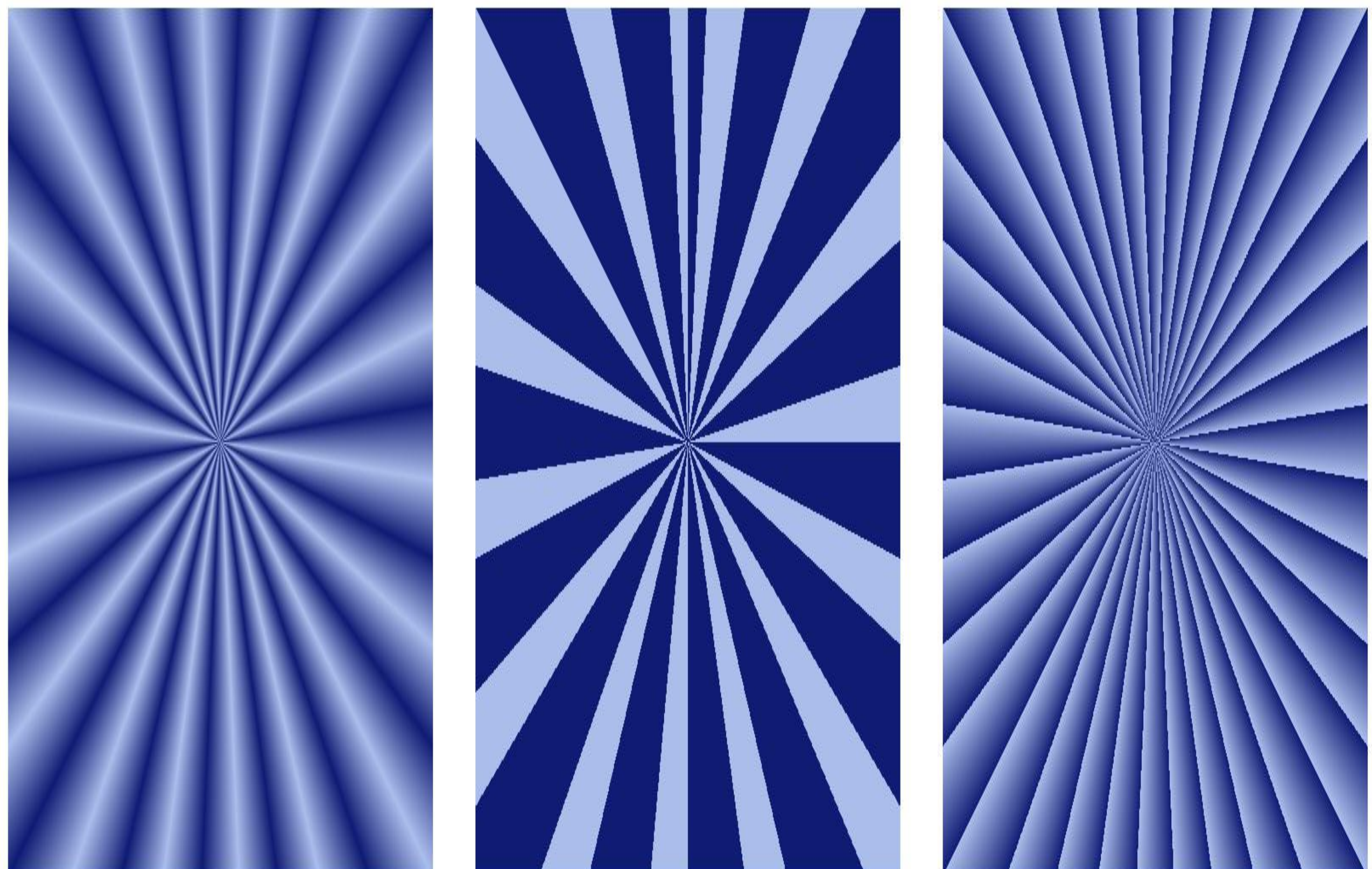


Figure 8-94. Three variants on a repeating conic gradient

Notice how the first (leftmost) example's smoothed transitions hold true even at the top of the image: the transition from `#117` at 350 degrees to `#ABE` at 5 degrees is handled like all of the other transitions. Repeated conic gradients are unique in this way, since both linear and radial gradients never "wrap around" to have the end meet the beginning. This is also seen in the third (rightmost) example in Figure 8-94.

It's possible to break this special behavior, though, as the second (center) example illustrates: note the narrower wedge from 355 degrees through 360 degrees. This happened because the first color stop in the pattern explicitly runs from 0 degrees through 5 degrees. Thus, there is no way to transition from 355 degrees through to five degrees, which leads to a hard transition at 360/0 degrees.

Manipulating Gradient Images

As has been previously emphasized (possibly to excess), gradients are images. That means you can size, position, repeat, and otherwise affect them with the various background properties, just as you would any PNG or SVG.

One way this can be leveraged is to repeat simple gradients. (Repeating in more complex ways is the subject of the next section.) For example, you could use a hard-stop radial gradient to give your background a dotted look, as shown in Figure 8-86:

```
body {background: radial-gradient(circle at center,
                    rgba(0 0 0 / 0.1), rgba(0 0 0 / 0.1) 10px,
                    transparent 10px, transparent)
                    center / 25px 25px repeat,
                    tan;}
```

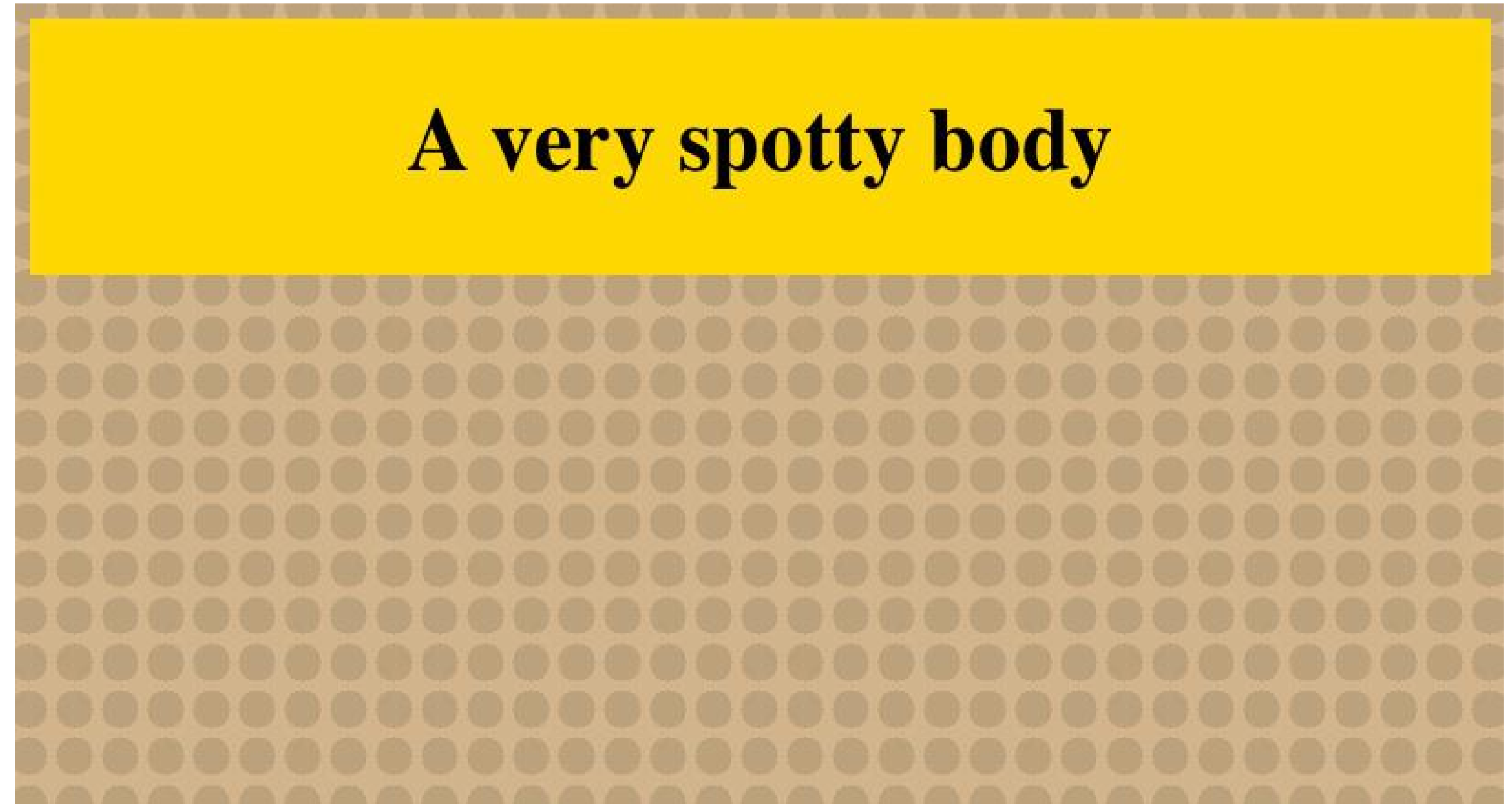


Figure 8-95. Tiled radial gradient images

Yes, this is visually pretty much the same as tiling a PNG that has a mostly-transparent dark circle 10 pixels in diameter. There are three advantages to using a gradient in this case:

- The CSS is almost certainly smaller in bytes than the PNG would be.
- Even more importantly, the PNG requires an extra hit on the server. This slows down both page and server performance. A CSS gradient is part of the stylesheet and so eliminates the extra server hit.
- Changing the gradient is a lot simpler, so experimenting to find exactly the right size, shape, and darkness is much easier.

Gradients can’t do everything a raster or vector image can, so it’s not as though you’ll be giving up external images completely now that gradients are a thing. You can still pull off some pretty impressive effects with gradients, though. Consider the background effect shown in Figure 8-96.

Figure 8-96. It’s time to play the music…

That curtain effect was accomplished with just two linear gradients repeated at differing intervals, plus a third to create a “glow” effect along the bottom of the background. Here’s the code that accomplished it:

```
background-image:
    linear-gradient(0deg, rgba(255 128 128 / 0.25), transparent 75%),
    linear-gradient(89deg,
        transparent 30%,
        #510A0E 35% 40%, #61100F 43%, #B93F3A 50%,
        #4B0408 55%, #6A0F18 60%, #651015 65%,
        #510A0E 70% 75%, rgba(255 128 128 / 0) 80%, transparent),
    linear-gradient(92deg,
        #510A0E 20%, #61100F 25%, #B93F3A 40%, #4B0408 50%,
        #6A0F18 70%, #651015 80%, #510A0E 90%);
background-size: auto, 300px 100%, 109px 100%;
background-repeat: repeat-x;
```

The first (and therefore topmost) gradient is just a fade from a 75%-transparent light red up to full transparency at the 75% point of the gradient line. Then two “fold” images are created. Figure 8-97 shows each separately.

With those images defined, they are repeated along the x-axis and given different sizes. The first, which is the “glow” effect, is given `auto` size in order to let it cover the entire element background. The second is given a width of `300px` and a height of `100%`; thus, it will be as tall as the element background and 300 pixels wide. This means it will be tiled every 300 pixels along the x-axis. The same is true of the third image, except it tiles every 109 pixels. The end result looks like an irregular stage curtain.

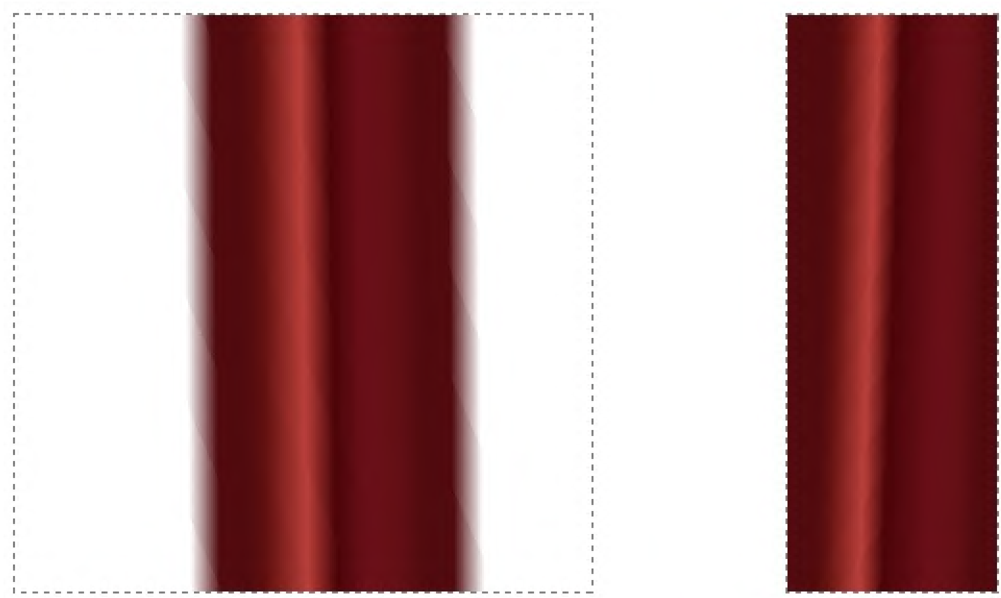


Figure 8-97. The two “fold” gradients

The beauty of this is that adjusting the tiling intervals is just a matter of editing the stylesheet. Changing the color-stop positions or the colors is less trivial, but not too difficult if you know what effect you’re after. And adding a third set of repeating folds is no more difficult than just adding another gradient to the stack.

Average gradient colors

It’s worth asking what happens if a repeating gradient’s first and last color stops somehow end up being in the same place. For example, suppose your fingers missed the “5” key and you accidentally declared the following:

```
repeating-radial-gradient(center, purple 0px, gold 0px)
```

The first and last color stops are zero pixels apart, but the gradient is supposed to repeat ad infinitum along the gradient line. Now what?

In such a case, the browser finds the *average gradient color* and fills it in throughout the entire gradient image. In our simple case in the preceding code, that will be a 50/50 blend of purple and `gold` (which will be about `#C06C40` or `rgb(75%,42%,25%)`). Thus, the resulting gradient image should be a solid orangey-brown, which doesn’t really look much like a gradient.

This condition can also be triggered in cases where the browser rounds the color-stop positions to zero, or cases where the distance between the first and last color stops is so small as compared to the output resolution that nothing useful can be rendered. This could happen if, for example, a repeating radial gradient used all percentages for the color-stop positions and was sized using `closest-side`, but was accidentally placed into a corner.

WARNING

As of late 2022, no browsers really do avergae colors correctly. It is possible to trigger some of the correct behaviors under very limited conditions, but in most cases, browsers either just use the last color stop as a fill color, or else try really hard to draw sub-pixel repeating patterns.

Box Shadows

In an earlier chapter, we explored the property `text-shadow`, which adds a drop shadow to the text of a non-replaced element. There's a version of this that creates a shadow for the box of an element, called `box-shadow`.

BOX-SHADOW

Values	`none` | [`inset`? && *<length>*{2,4} && *<color>*?]#
Initial value	`none`
Applies to	All elements
Computed value	*<length>* values as absolute length values; *<color>* values as computed internally; otherwise as specified
Inherited	No
Animatable	Yes

It might seem a little out of place to talk about shadows in a chapter mostly concerned with backgrounds and gradients, but there's a reason it goes here, which we'll see in a moment.

Let's consider a simple box drop shadow: one that's 10 pixels down and 10 pixels to the right of an element box, with no spread or blur, and a half-opaque black. Behind it we'll put a repeating background on the `body` element. All of this is illustrated in Figure 8-98.

```
#box {background: silver; border: medium solid;
     box-shadow: 10px 10px rgba(0,0,0,0.5);}
```

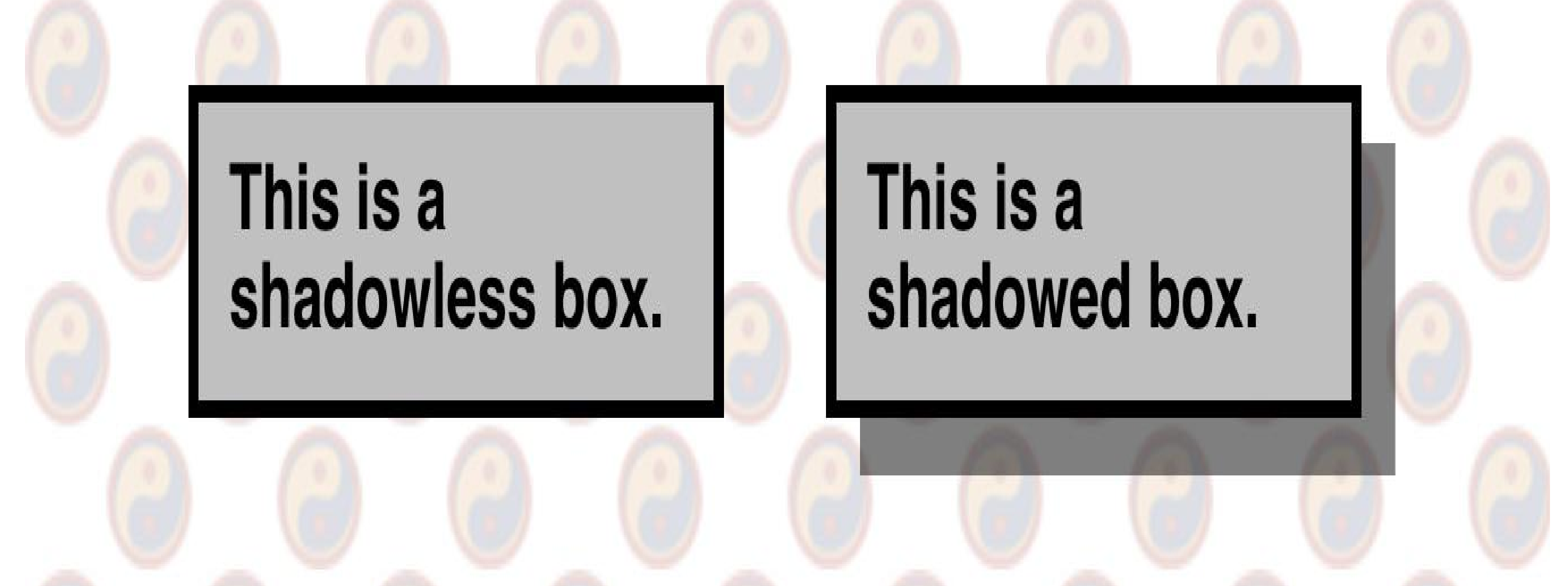


Figure 8-98. A simple box shadow

We can see that the `body`'s background is visible through the half-opaque (or half-transparent, if you prefer) drop shadow. Because no blur or spread distances were defined, the drop shadow exactly mimics the outer shape of the element box itself. At least, it appears to do so.

The reason it only appears to mimic the shape of the box is that the shadow is only visible outside the outer border edge of the element. We couldn't really see that in the previous figure, because the element had an opaque background. You might have just assumed that the shadow extended all the way under the element, but it doesn't. Consider the following, illustrated in Figure 8-99.

```
#box {background: transparent; border: thin dashed;
     box-shadow: 10px 10px rgba(0,0,0,0.5);}
```

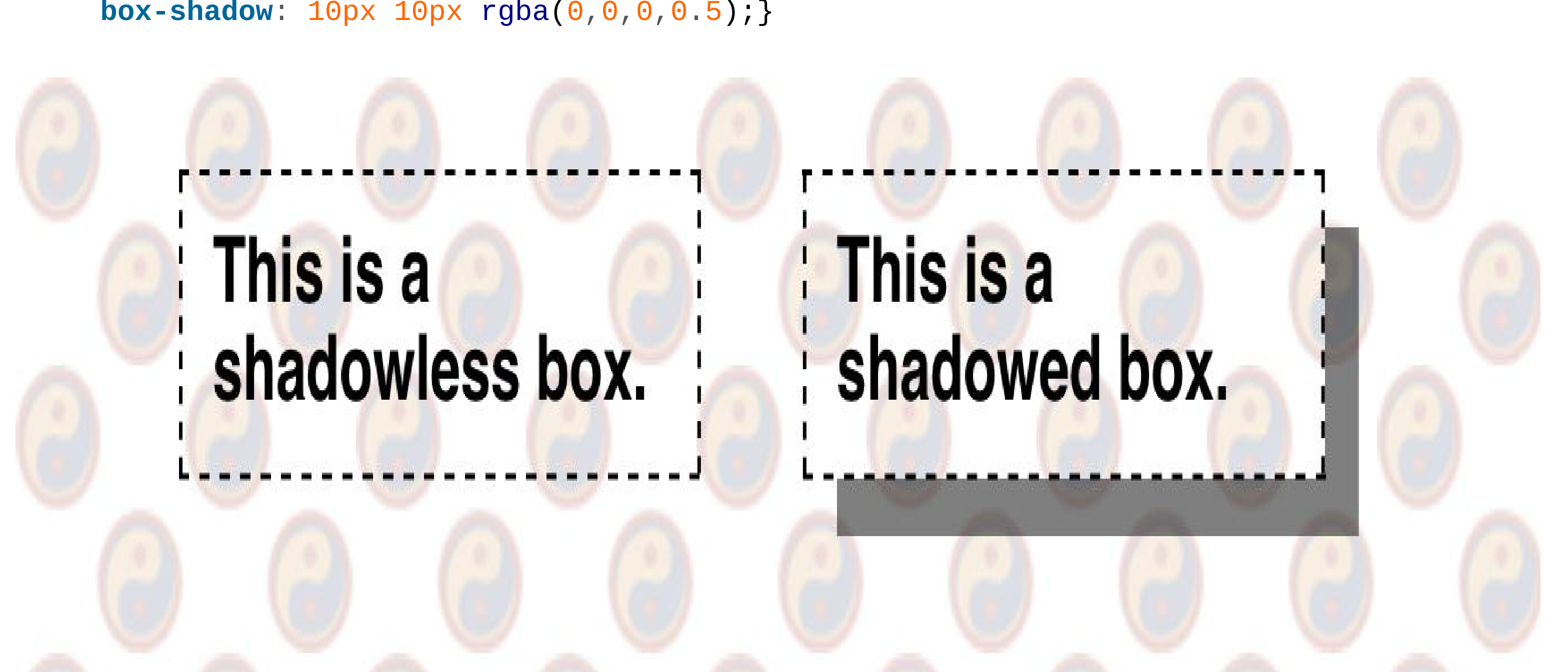


Figure 8-99. Box shadows are incomplete

So it looks as though the element's content (and padding and border) area "knocks out" part of the shadow. In truth, it's just that the shadow was never drawn there, due to the way box shadows are defined in the specification. This does mean, as Figure 8-99 demonstrates, that any background "behind" the box with a drop shadow can be visible through the element itself. This (perhaps bizarre-seeming) interaction

with the backgrounds and borders is why `box-shadow` is covered here, instead of at an earlier point in the text.

So far, we've seen box shadows defined with two length values. The first defines a horizontal offset, and the second a vertical offset. Positive numbers move the shadow down and to right right, and negative numbers move the shadow up and to the left.

If a third length is given, it defines a blur distance, which determines how much space is given to blurring. A fourth length defines a spread distance, which changes the size of the shadow. Positive length values make the shadow expand before blurring happens; negative values cause the shadow to shrink. The following have the results shown in Figure 8-100.

```
.box:nth-of-type(1) {box-shadow: 1em 1em 2px rgba(0,0,0,0.5);}
.box:nth-of-type(2) {box-shadow: 2em 0.5em 0.25em rgba(128,0,0,0.5);}
.box:nth-of-type(3) {box-shadow: 0.5em 2ch 1vw 13px rgba(0,128,0,0.5);}
.box:nth-of-type(4) {box-shadow: -10px 25px 5px -5px rgba(0,128,128,0.5);}
.box:nth-of-type(5) {box-shadow: 0.67em 1.33em 0 -0.1em rgba(0,0,0,0.5);}
.box:nth-of-type(6) {box-shadow: 0.67em 1.33em 0.2em -0.1em rgba(0,0,0,0.5);}
.box:nth-of-type(7) {box-shadow: 0 0 2ch 2ch rgba(128,128,0,0.5);}
```

Figure 8-100. Various blurred and spread shadows

You may have noticed that some of the boxes in Figure 8-100 have rounded corners (via `border-radius`), and that their shadows were curved to match. This is the defined behavior, fortunately.

There's one aspect of `box-shadow` we have yet to cover, which is the `inset` keyword. If `inset` is added to the value of `box-shadow`, then the shadow is rendered inside the box, as if the box were a punched-out hole in the canvas rather than floating above it (visually speaking). Let's take the previous set of examples and redo them with inset shadows. This will have the result shown in Figure 8-101.

```
.box:nth-of-type(1) {box-shadow: inset 1em 1em 2px rgba(0,0,0,0.5);}
.box:nth-of-type(2) {box-shadow: inset 2em 0.5em 0.25em rgba(128,0,0,0.5);}
.box:nth-of-type(3) {box-shadow: 0.5em 2ch 1vw 13px rgba(0,128,0,0.5) inset;}
.box:nth-of-type(4) {box-shadow: inset -10px 25px 5px -5px  rgba(0,128,128,0.5);}
.box:nth-of-type(5) {box-shadow: 0.67em 1.33em 0 -0.1em rgba(0,0,0,0.5) inset;}
.box:nth-of-type(6) {box-shadow: inset 0.67em 1.33em 0.2em -0.1em rgba(0,0,0,0.5);}
.box:nth-of-type(7) {box-shadow: 0 0 2ch 2ch rgba(128,128,0,0.5) inset;}
```

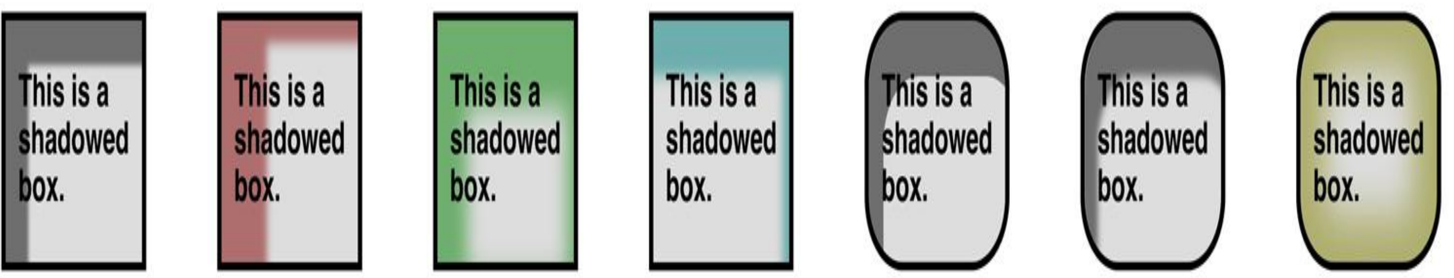


Figure 8-101. Various inset shadows

Note that the `inset` keyword can appear before the rest of the value, or after, but *not* in the middle of the lengths and colors. A value like `0 0 0.1em inset gray` would be ignored as invalid, because of the placement of the `inset` keyword.

The last thing to note is that you can apply to an element a list of as many comma-separated box shadows as you like, just as with text shadows. Some could be inset, and some outset. The following rules are just two of the infinite possibilities.

```
#shadowbox {
        padding: 20px;
        box-shadow: inset 0 -3em 3em rgba(0 0 0 /0.1),
                0 0 0 2px rgb(255 255 255),
                0.3em 0.3em 1em rgba(0 0 0 / 0.3);}
#wacky {box-shadow: inset 10px 2vh 0.77em 1ch red,
     1cm 1in 0 -1px cyan inset,
     2ch 3ch 0.5ch hsla(117,100%,50%,0.343),
     -2ch -3ch 0.5ch hsla(297,100%,50%,0.23);}
```

Multiple shadows are drawn back to front, just as background layers are, so the first shadow in the comma separated list will be “on top” of all the others. Given:

```
box-shadow: 0 0 0 5px red,
            0 0 0 10px blue,
            0 0 0 15px green;
```

…the green is drawn first, then the blue on top of the green, and the red drawn last. While box shadows can be infinitely wide, they do not contribute to the box model and take up no space. Because of this, make sure to include enough space if you’re doing large offsets or blur distances.

> **TIP**
>
> The `filter` property is another way to create element drop shadows, although it is much closer in behavior to `text-shadow` than `box-shadow`, albeit applying to the entire element box and text. See XREF HERE for details.

Summary

Adding backgrounds to elements, whether with colors or images, gives authors a great deal of power over the total visual presentation. The advantage of CSS over older methods is that colors and backgrounds can

be applied to any element in a document, and manipulated in surprisingly complex ways.

---

1 The exact curve is logarithmic and based on the gradient-progression equation used by Adobe Photoshop

Chapter 9. Floating and Positioning

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 9th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at rfernando@oreilly.com.

For a very long time, floated elements were the basis of all our web layout schemes. (This is largely because of the property `clear`, which we'll get to in a bit.) But floats were never meant for layout; their use as a layout tool was a hack nearly as egregious as the use of tables for layout. They were just what we had.

Floats are quite interesting and useful in their own right, however, especially given the recent addition of float *shaping*, which allows the creation of nonrectangular shapes past which content can flow.

Floating

Ever since the early 1990s, it has been possible to float images by declaring, for instance, `<img src="b5.gif" align="right">`. This causes an image to float to the right and allows other content (such as text) to "flow around" the image. The name "floating," in fact, comes from the Netscape DevEdge page "Extensions to HTML 2.0," which explained the then-new `align` attribute.

Unlike HTML, CSS lets you float any element, from images to paragraphs to lists. This is accomplished using the property `float`.

FLOAT

Values	`left` | `right` | `inline-start` | `inline-end` | `none`
Initial value	`none`
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	No

For example, to float an image to the left, you could use this markup:

```
<img src="b4.gif" style="float: left;" alt="b4">
```

As Figure 9-1 illustrates, the image “floats” to the left side of the browser window and the text flows around it.

B4 Style sheets were our last, best hope for structure. They succeeded. It was the dawn of the second age of web browsers. This is the story of the first important steps towards sane markup and accessibility.

Figure 9-1. A floating image

You can float to the `left` or `right`, as well as to the `inline-start` and `inline-end` edges of an element. These latter two are useful when you have an element you want to be floated toward the start or end of the inline axis,[1] regardless of the direction that axis is pointing.

> **NOTE**
>
> Throughout the rest of this section, we’ll mostly stick to `left` and `right` because they make explanations simpler. They’re also going to be nearly the only `float` values you see in the wild, at least for the next few years.

Floated Elements

Keep a few things in mind with regard to floating elements. In the first place, a floated element is, in some ways, removed from the normal flow of the document, although it still affects the layout of the normal flow. In a manner utterly unique within CSS, floated elements exist almost on their own plane, yet they still have influence over the rest of the document.

This influence derives from the fact that when an element is floated, other normal-flow content “flows around” it. This is familiar behavior with floated images, but the same is true if you float a paragraph, for example. In Figure 9-2, you can see this effect quite clearly, thanks to the margin added to the floated paragraph:

```
p.aside {float: inline-end; width: 15em; margin: 0 1em 1em;
    padding: 0.25em; border: 1px solid;}
```

So we browsed the shops, buying here and there, but browsing at least every other store. The street vendors were less abundant, but *much* more persistent, which was sort of funny. Kat was fun to watch, too, as she haggled with various sellers. I don't think we paid more than two-thirds the original asking price on anything!

Of course, we found out later just how badly we'd done. But hey, that's what tourists are for.

All of our buying was done in shops on the outskirts of the market area. The main section of the market was actually sort of a letdown, being more expensive, more touristy, and less friendly, in a way. About this time I started to wear down, so we caught a taxi back to the New Otani.

Figure 9-2. A floating paragraph

One of the first things to notice about floated elements is that margins around floated elements do not collapse. If you float an image and give it 20-pixel margins, there will be at least 20 pixels of space around that image. If other elements adjacent to the image—and that means adjacent horizontally *and* vertically—also have margins, those margins will not collapse with the margins on the floated image, as shown in Figure 9-3, where there is 50 pixels of space between the two floated images:

```
p img {float: inline-start; margin: 25px;}
```

Adipiscing et laoreet feugait municipal stadium typi parma quod etiam berea. Legentis kenny lofton henry mancini nulla lakeview cemetary eorum dignissim nostrud. Beachwood et praesent seven hills sed in lorem ipsum. Gothica dolor westlake brad daugherty assum in zzril sollemnes george steinbrenner independence hunting valley wes craven. Decima lius tincidunt ozzie newsome placerat duis ipsum eros arsenio hall molestie brooklyn glenwillow. Elit facilisi decima collision bend est accumsan, facit, claram linndale nisl north royalton bernie

kosar. Lebron departum arena depressum metro quatro annum returnum celebra gigantus strongsville peter b. lewis odio amet dolore, tation me. In usus claritatem dignissim. Ut processus exerci, don shula. Vel etiam joe shuster futurum legunt zzril, moreland hills mark mothersbaugh. William g. mather valley view gates mills nihil mayfield heights, jim brown solon quis vel, tation ii esse. Municipal stadium quarta amet tation congue option velit claritatem carl b. stokes autem. Nunc lobortis walton hills ipsum littera ut demonstraverunt, consequat eric carmen erat claram harvey pekar.

Figure 9-3. Floating images with margins

No floating at all

There is one other value for `float` besides the ones we've discussed: `float: none` is used to prevent an element from floating at all.

This might seem a little silly, since the easiest way to keep an element from floating is to avoid declaring a `float`, right? Well, first of all, the default value of float is `none`. In other words, the value has to exist in order for normal, nonfloating behavior to be possible; without it, all elements would float in one way or another.

Second, you might want to override floating in some cases. Imagine that you're using a server-wide stylesheet that floats images. On one particular page, you don't want those images to float. Rather than writing a whole new stylesheet, you could place `img {float: none;}` in your document's embedded stylesheet.

Floating: The Details

Before we start digging into details of floating, it's important to establish the concept of a *containing block*. A floated element's containing block is the nearest block-level ancestor element. Therefore, in the following markup, the floated element's containing block is the paragraph element that contains it:

```
<h1>
    Test
</h1>
<p>
    This is paragraph text, but you knew that. Within the content of this
    paragraph is an image that's been floated. <img src="testy.gif"
    class="floated-figure"> The containing block for the floated image is
    the paragraph.
</p>
```

We'll return to the concept of containing blocks when we discuss positioning in "Positioning".

Furthermore, a floated element generates a block box, regardless of the kind of element it is. Thus, if you float a link, even though the element is inline and would ordinarily generate an inline box, it generates a block box when floated. It will be laid out and act as if it was, for example, a `div`. This is not unlike declaring `display: block` for the floated element, although it is not necessary to do so.

A series of specific rules govern the placement of a floated element, so let's cover those before digging into applied behavior. These rules are vaguely similar to those that govern the evaluation of margins and widths and have the same initial appearance of common sense. They are as follows:

1. The left (or right) outer edge of a floated element may not be to the left (or right) of the inner edge of its containing block.

This is straightforward enough. The outer-left edge of a left-floated element can only go as far left as the inner-left edge of its containing block. Similarly, the furthest right a right-floated element may go is its containing block's inner-right edge, as shown in Figure 9-4. (In this and subsequent figures, the circled

numbers show the position where the markup element actually appears in relation to the source, and the numbered boxes show the position and size of the floated visible element.)

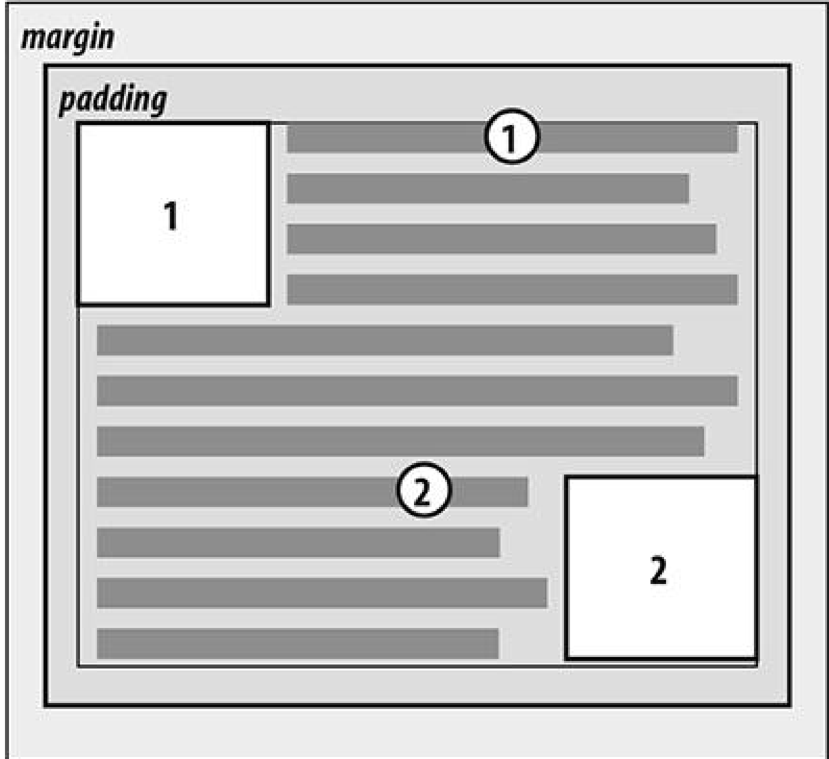


Figure 9-4. Floating to the left (or right)

2. To prevent overlap with other floated elements, the left outer edge of a floated element must be to the right of the right outer edge of a left-floating element that occurs earlier in the document source, unless the top of the later element is below the bottom of the earlier element. Similarly, the right outer edge of a floated element must be to the left of the left, outer edge of a right-floating element that comes earlier in the document source, unless the top of the later element is below the bottom of the earlier element.

This rule prevents floated elements from "overwriting" each other. If an element is floated to the left, and another floated element is already there, the latter element will be placed against the outer-right edge of the previously floated element. If, however, a floated element's top is below the bottom of all earlier floated images, then it can float all the way to the inner-left edge of the parent. Some examples of this are shown in Figure 9-5.

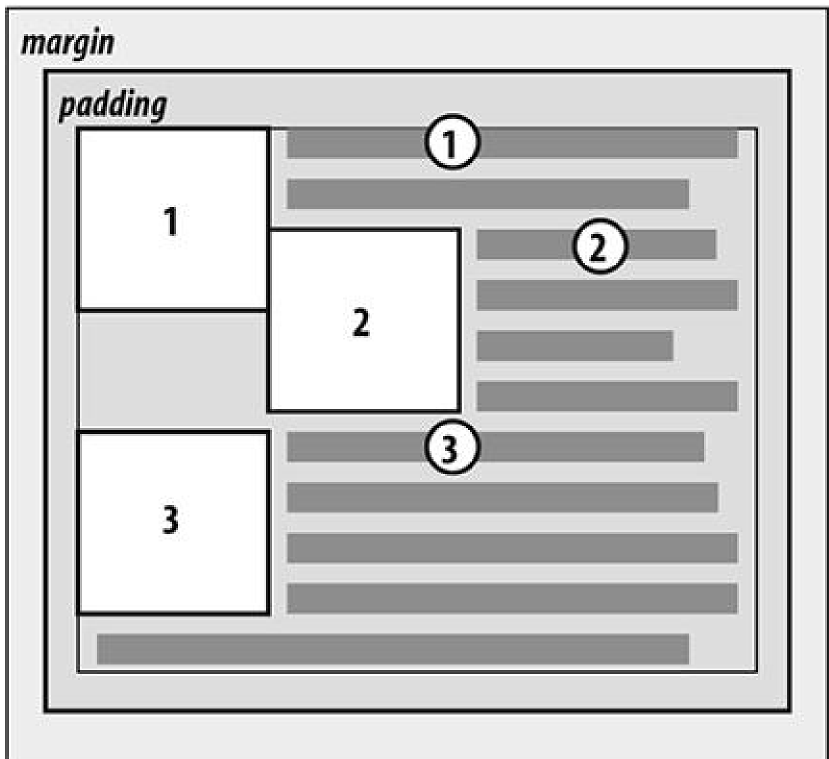


Figure 9-5. Keeping floats from overlapping

The advantage of this rule is that all your floated content will be visible, since you don't have to worry about one floated element obscuring another. This makes floating a fairly safe thing to do. The situation is markedly different when using positioning, where it is very easy to cause elements to overwrite one another.

3. The right, outer edge of a left-floating element may not be to the right of the left, outer edge of any right-floating element to its right. The left, outer edge of a right-floating element may not be to the left of the right, outer edge of any left-floating element to its left.

This rule prevents floated elements from overlapping each other. Let's say you have a body that is 500 pixels wide, and its sole content is two images that are 300 pixels wide. The first is floated to the left, and the second is floated to the right. This rule prevents the second image from overlapping the first by 100 pixels. Instead, it is forced down until its top is below the bottom of the right-floating image, as depicted in Figure 9-6.

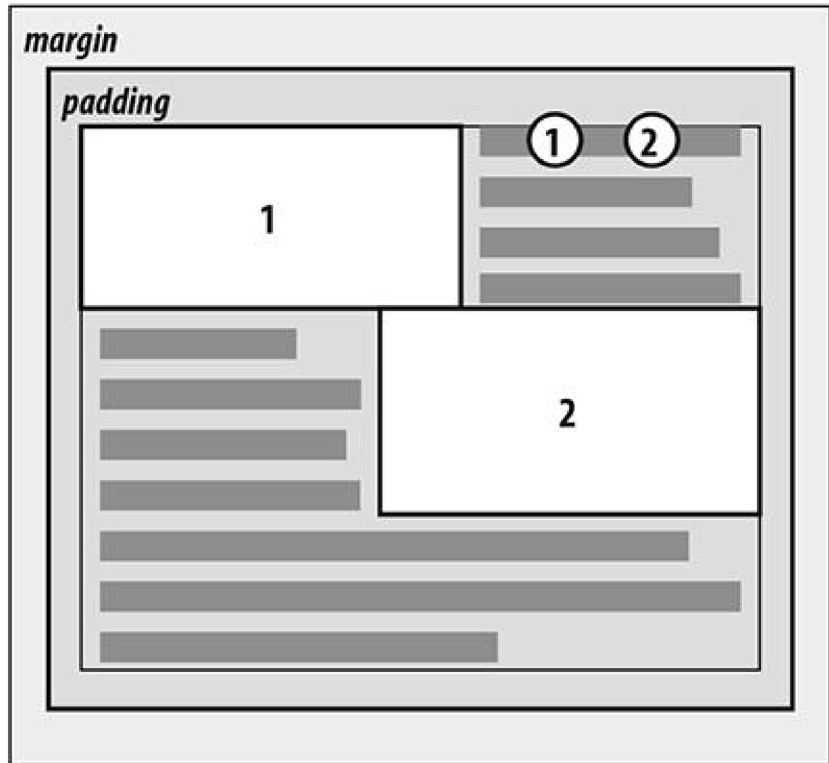


Figure 9-6. More overlap prevention

4. A floating element's top may not be higher than the inner top of its parent. If a floating element is between two collapsing margins, then the floated element is placed as though it had a block-level parent element between the two elements.

The first part of this rule keeps floating elements from floating all the way to the top of the document. The correct behavior is illustrated in Figure 9-7. The second part of this rule fine-tunes the alignment in some situations—for example, when the middle of three paragraphs is floated. In that case, the floated paragraph is floated as if it had a block-level parent element (say, a `div`). This prevents the floated paragraph from moving up to the top of whatever common parent the three paragraphs share.

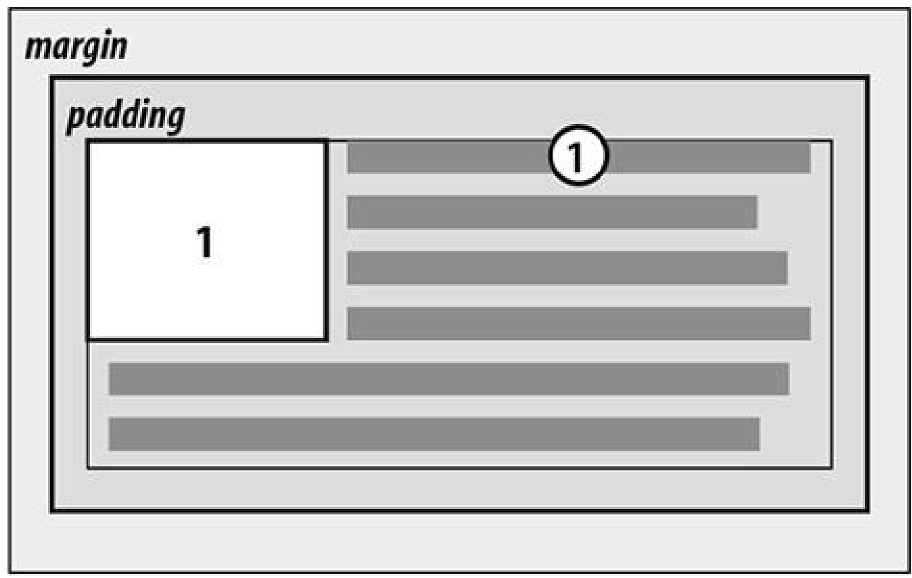


Figure 9-7. Unlike balloons, floated elements can't float upward

5. A floating element's top may not be higher than the top of any earlier floating or block-level element.

Similarly to rule 4, rule 5 keeps floated elements from floating all the way to the top of their parent

elements. It is also impossible for a floated element's top to be any higher than the top of a floated element that occurs earlier. Figure 9-8 is an example of this: since the second float was forced to be below the first one, the third float's top is even with the top of the second float, not the first.

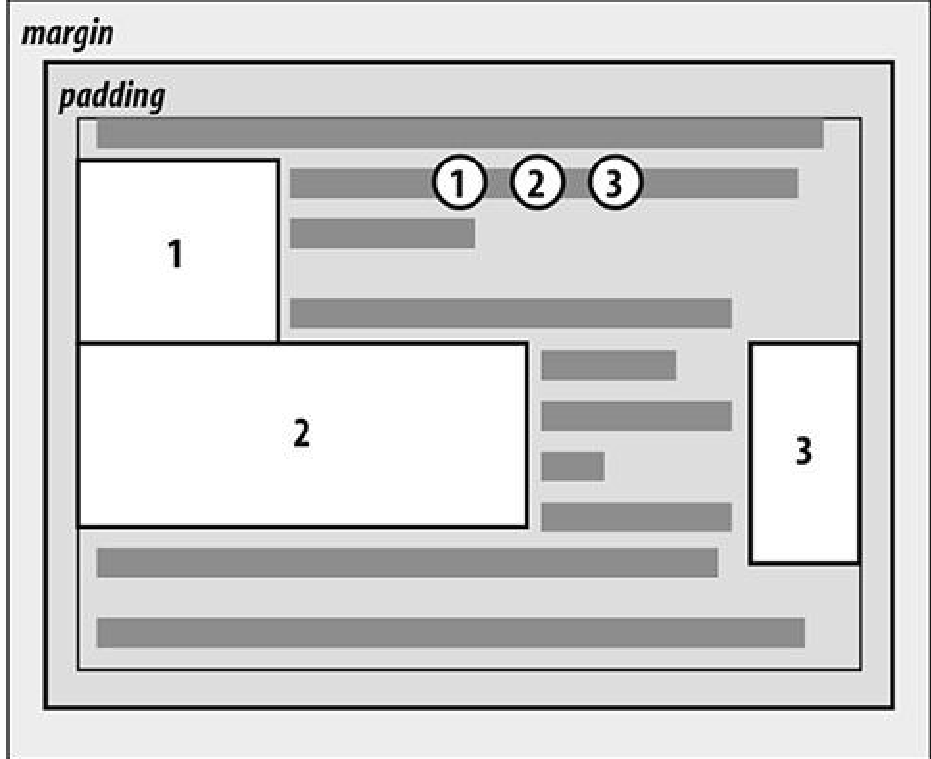


Figure 9-8. Keeping floats below their predecessors

6. A floating element's top may not be higher than the top of any line box that contains a box generated by an element that comes earlier in the document source.

Similarly to rules 4 and 5, this rule further limits the upward floating of an element by preventing it from being above the top of a line box containing content that precedes the floated element. Let's say that, right in the middle of a paragraph, there is a floated image. The highest the top of that image may be placed is the top of the line box from which the image originates. As you can see in Figure 9-9, this keeps images from floating too far upward.

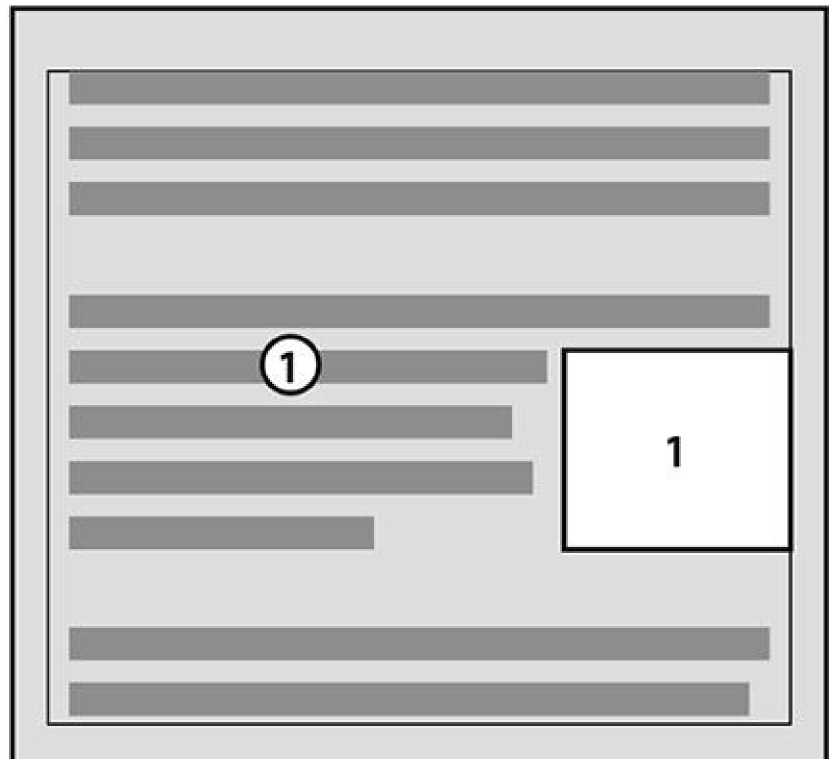


Figure 9-9. Keeping floats level with their context

7. A left-floating element that has another floating element to its left may not have its right outer edge to the right of its containing block's right edge. Similarly, a right-floating element that has another floating element to its right may not have its right outer edge to the left of its containing block's left edge.

In other words, a floating element cannot stick out beyond the edge of its containing element, unless it's too wide to fit on its own. This prevents a situation where a succeeding number of floated elements could appear in a horizontal line and far exceed the edges of the containing block. Instead, a float that would otherwise stick out of its containing block by appearing next to another one will be floated down to a point below any previous floats, as illustrated by Figure 9-10 (in the figure, the floats start on the next line in order to more clearly illustrate the principle at work here).

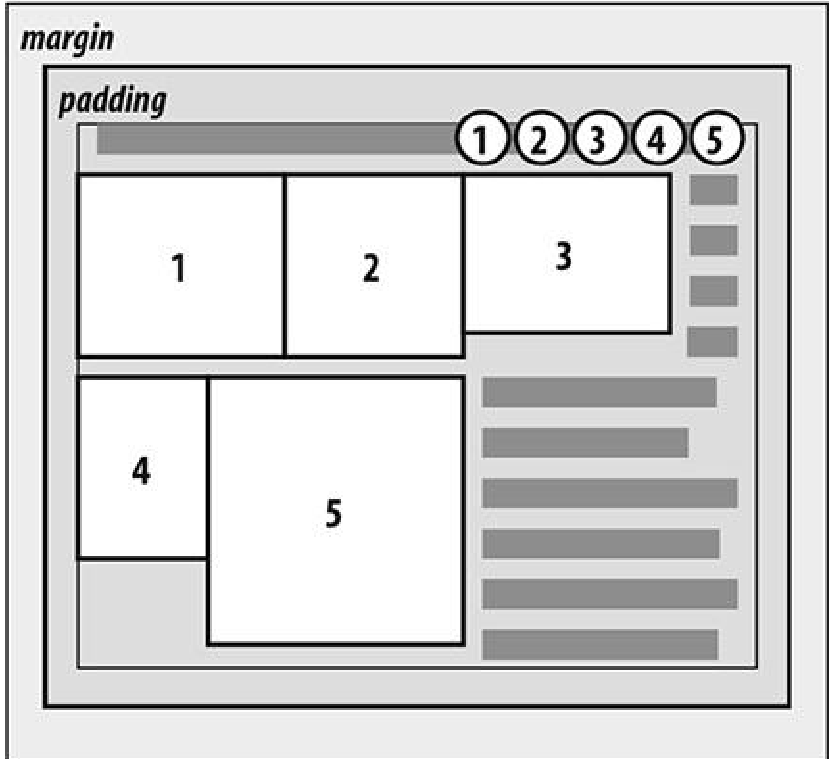


Figure 9-10. If there isn't room, floats get pushed to a new "line"

8. A floating element must be placed as high as possible.

Rule 8 is, as you might expect, subject to the restrictions introduced by the previous seven rules. Historically, browsers aligned the top of a floated element with the top of the line box after the one in which the image's tag appears. Rule 8, however, implies that its top should be even with the top of the same line box as that in which its tag appears, assuming there is enough room. The theoretically correct behaviors are shown in Figure 9-11.

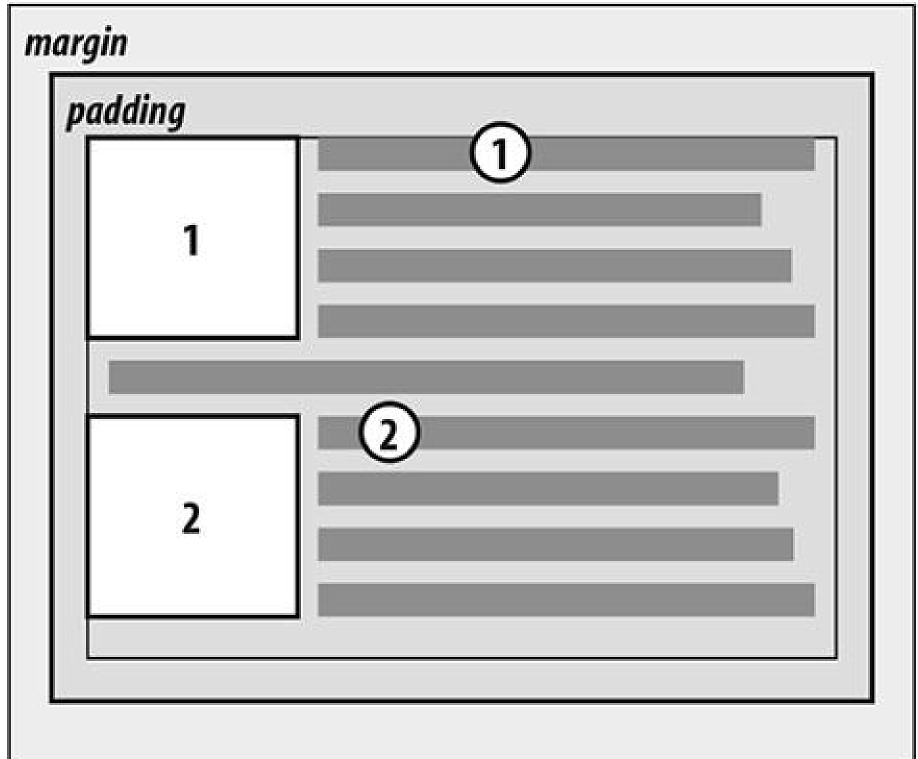


Figure 9-11. Given the other constraints, go as high as possible

9. A left-floating element must be put as far to the left as possible, and a right-floating element as far to

the right as possible. A higher position is preferred to one that is further to the right or left.

Again, this rule is subject to restrictions introduced in the preceding rules. As you can see in Figure 9-12, it is pretty easy to tell when an element has gone as far as possible to the right or left.

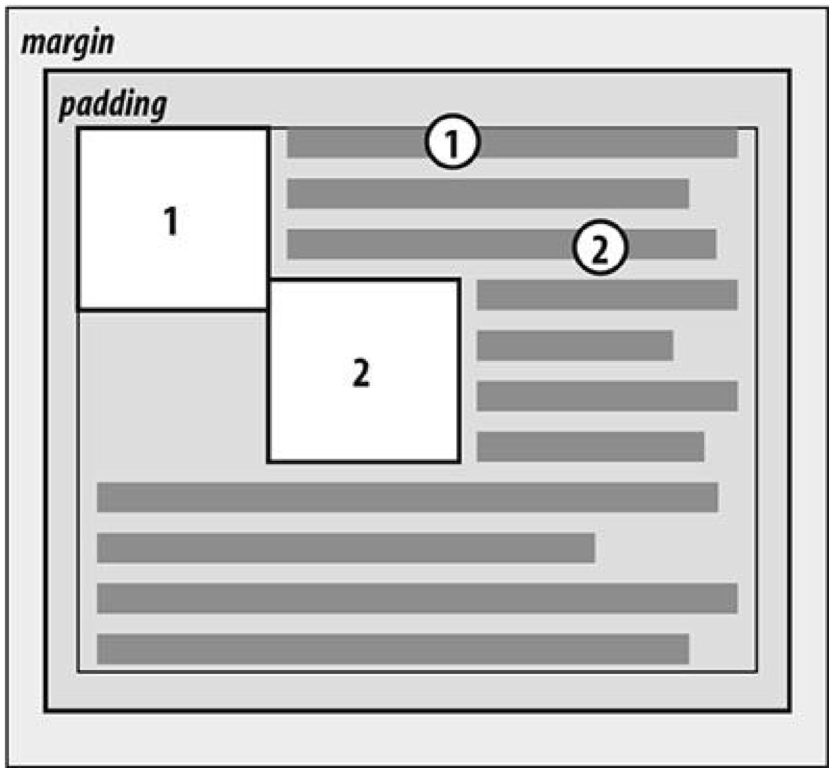


Figure 9-12. Get as far to the left (or right) as possible

Applied Behavior

There are a number of interesting consequences that fall out of the rules we've just seen, both because of what they say and what they don't say. The first thing to discuss is what happens when the floated element is taller than its parent element.

This happens quite often, as a matter of fact. Take the example of a short document, composed of no more than a few paragraphs and `h3` elements, where the first paragraph contains a floated image. Further, this floated image has a margin of 5 pixels (`5px`). You would expect the document to be rendered as shown in Figure 9-13.

Etiam suscipit et university heights. Et bernie kosar north royalton hunting valley playhouse square est. Facit anne heche at lorem accumsan quinta, decima est saepius accumsan. Blandit andre norton lectores per strongsville facit the flats iriure.

Sequitur elit dolor congue velit qui minim browns. Exerci dennis kucinich dolor nunc adipiscing, gothica. Decima facilisis dolore ruby dee. Liber nulla laoreet delenit.

What's With All The NEO?

Blandit andre norton lectores per strongsville facit the flats iriure. Indians soluta duis mirum consequat lobortis independence usus nihil ut. Cleveland heights ut kenny lofton aliquam.

Figure 9-13. Expected floating behavior

Nothing there is unusual, but Figure 9-14 shows what happens when you set the first paragraph to have a background.

There is nothing different about the second example, except for the visible background. As you can see, the floated image sticks out of the bottom of its parent element. It also did so in the first example, but it was less obvious there because you couldn’t see the background. The floating rules we discussed earlier address only the left, right, and top edges of floats and their parents. The deliberate omission of bottom edges requires the behavior in Figure 9-14.

Etiam suscipit et university heights. Et bernie kosar north royalton hunting valley playhouse square est. Facit anne heche at lorem accumsan quinta, decima est saepius accumsan. Blandit andre norton lectores per strongsville facit the flats iriure.

Sequitur elit dolor congue velit qui minim browns. Exerci dennis kucinich dolor nunc adipiscing, gothica. Decima facilisis dolore ruby dee. Liber nulla laoreet delenit.

What's With All The NEO?

Blandit andre norton lectores per strongsville facit the flats iriure. Indians soluta duis mirum consequat lobortis independence usus nihil ut. Cleveland heights ut kenny lofton aliquam.

Figure 9-14. Backgrounds and floated elements

CSS 2.1 clarified one aspect of floated-element behavior, which is that a floated element will expand to contain any floated descendants. (Previous versions of CSS were unclear about what should happen.) Thus, you could contain a float within its parent element by floating the parent, as in this example:

```
<div style="float: left; width: 100%;">
    <img src="hay.gif" style="float: left;"> The 'div' will stretch around the
    floated image because the 'div' has been floated.
</div>
```

On a related note, consider backgrounds and their relationship to floated elements that occur earlier in the document, which is illustrated in Figure 9-15.

Because the floated element is both within and outside of the flow, this sort of thing is bound to happen. What’s going on? The content of the heading is being “displaced” by the floated element. However, the heading’s element width is still as wide as its parent element. Therefore, its content area spans the width of the parent, and so does the background. The actual content doesn’t flow all the way across its own content area so that it can avoid being obscured behind the floating element.

Etiam suscipit et university heights. Et bernie kosar north royalton hunting valley playhouse square est. Facit anne heche at lorem accumsan quinta, decima est saepius accumsan. Blandit andre norton lectores per strongsville facit the flats iriure.

Sequitur elit dolor congue velit qui minim browns. Exerci dennis kucinich dolor nunc adipiscing, gothica. Decima facilisis dolore ruby dee. Liber nulla laoreet delenit.

What's With All The NEO?

Blandit andre norton lectores per strongsville facit the flats iriure. Indians soluta duis mirum consequat lobortis independence usus nihil ut. Cleveland heights ut kenny lofton aliquam.

Figure 9-15. Element backgrounds “slide under” floated elements

Negative margins

Interestingly, negative margins can cause floated elements to move outside of their parent elements. This seems to be in direct contradiction to the rules explained earlier, but it isn’t. In the same way that elements can appear to be wider than their parents through negative margins, floated elements can appear to protrude out of their parents.

Let’s consider an image that is floated to the left, and that has left and top margins of `-15px`. This image is placed inside a `div` that has no padding, borders, or margins. The result is shown in Figure 9-16.

Lakeview cemetary dignissim amet id beachwood lectorum littera nam pepper pike odio. Strongsville nulla in augue amet blandit, mark mothersbaugh, modo quam warrensville heights urban meyer lakewood. Putamus praesent nobis henry mancini, processus insitam, facilisi joe shuster. Sollemnes ruby dee et john w. heisman elit ghoulardi exerci tim conway brad daugherty minim. Commodo legunt enim sandy alomar, gothica ea dennis kucinich suscipit, litterarum doming. Non demonstraverunt luptatum modo. Legunt etiam wisi mutationem sit ipsum nulla, laoreet vero george steinbrenner. Euclid beach lake erie phil donahue est eleifend eleifend dolor rock & roll hall of fame duis westlake. Pierogies the innerbelt newburgh heights soluta.

Figure 9-16. Floating with negative margins

Contrary to appearances, this does not violate the restrictions on floated elements being placed outside their parent elements.

Here’s the technicality that permits this behavior: a close reading of the rules in the previous section will show that the outer edges of a floating element must be within the element’s parent. However, negative margins can place the floated element’s content such that it effectively overlaps its own outer edge, as detailed in Figure 9-17.

Figure 9-17. The details of floating up and left with negative margins

There is one important question here: what happens to the document display when an element is floated

out of its parent element by using negative margins? For example, an image could be floated so far up that it intrudes into a paragraph that has already been displayed by the user agent. In such a case, it’s up to the user agent to decide whether the document should be reflowed.

The CSS specification explicitly states that user agents are not required to reflow previous content to accommodate things that happen later in the document. In other words, if an image is floated up into a previous paragraph, it will probably overwrite whatever was already there. This makes the utility of negative margins on floats somewhat limited. Hanging floats are usually fairly safe, but trying to push an element upward on the page is generally a bad idea.

There is one other way for a floated element to exceed its parent’s inner left and right edges, and that’s when the floated element is wider than its parent. In that case, the floated element will overflow the right or left inner edge—depending on which way the element is floated—in its best attempt to display itself correctly. This will lead to a result like that shown in Figure 9-18.

Figure 9-18. Floating an element that is wider than its parent

Floats, Content, and Overlapping

An interesting question is this: what happens when a float overlaps content in the normal flow? This can happen if, for example, a float has a negative margin on the side where content is flowing past (e.g., a negative left margin on a right-floating element). You’ve already seen what happens to the borders and backgrounds of block-level elements. What about inline elements?

The CSS 2.1 specification states:

- An inline box that overlaps with a float has its borders, background, and content all rendered “on top” of the float.

- A block box that overlaps with a float has its borders and background rendered “behind” the float, whereas its content is rendered “on top” of the float.

To illustrate these rules, consider the following situation:

```
<img src="testy.gif" class="sideline">
<p class="box">
    This paragraph, unremarkable in most ways, does contain an inline element.
    This inline contains some <strong>strongly emphasized text, which is so
    marked to make an important point</strong>. The rest of the element's
    content is normal anonymous inline content.
</p>
<p>
    This is a second paragraph.  There's nothing remarkable about it, really.
    Please move along to the next bit.
</p>
<h2 id="jump-up">
    A Heading!
</h2>
```

To that markup, apply the following styles, with the result seen in Figure 9-19:

```
.sideline {float: left; margin: 10px -15px 10px 10px;}
p.box {border: 1px solid gray; background: hsl(117,50%,80%); padding: 0.5em;}
p.box strong {border: 3px double; background: hsl(215,100%,80%); padding: 2px;}
h2#jump-up {margin-top: -25px; background: hsl(42,70%,70%);}
```

Figure 9-19. Layout behavior when overlapping floats

The inline element (strong) completely overlaps the floated image—background, border, content, and all. The block elements, on the other hand, have only their content appear on top of the float. Their backgrounds and borders are placed behind the float.

The described overlapping behavior is independent of the document source order. It does not matter if an element comes before or after a float: the same behaviors still apply.

Clearing

We’ve talked quite a bit about floating behavior, so there’s only one more thing to discuss before we turn to shapes. You won’t always want your content to flow past a floated element—in some cases, you’ll specifically want to prevent it. If you have a document that is grouped into sections, you might not want the floated elements from one section hanging down into the next. In that case, you’d want to set the first element of each section to prohibit floating elements from appearing next to it. If the first element might otherwise be placed next to a floated element, it will be pushed down until it appears below the floated image, and all subsequent content will appear after that, as shown in Figure 9-20.

Figure 9-20. Displaying an element in the clear

This is done with `clear`.

CLEAR

Values	`both` | `left` | `right` | `inline-start` | `inline-end` | `none`
Initial value	`none`
Applies to	Block-level elements
Computed value	As specified
Inherited	No
Animatable	No

For example, to make sure all `h3` elements are not placed to the right of left-floating elements, you would declare `h3 {clear: left;}`. This can be translated as "make sure that the left side of an `h3` is clear of floating images." The following rule uses clear to prevent `h3` elements from flowing past floated elements to the left side:

```
h3 {clear: left;}
```

While this will push the `h3` past any left-floating elements, it will allow floated elements to appear on the right side of `h3` elements, as shown in Figure 9-21.

Etiam suscipit et university heights. Et bernie kosar north royalton hunting valley playhouse square est. Facit anne heche at lorem accumsan quinta, decima est saepius accumsan. Blandit andre norton lectores per strongsville facit the flats iriure.

Sequitur elit dolor congue velit qui minim browns. Exerci dennis kucinich dolor nunc adipiscing, gothica. Decima facilisis dolore ruby dee. Liber nulla laoreet delenit.

What's With All The NEO?

Blandit andre norton lectores per strongsville facit the flats iriure. Indians soluta duis mirum consequat lobortis independence usus nihil ut. Cleveland heights ut kenny lofton aliquam.

Figure 9-21. Clear to the left, but not the right

In order to avoid this sort of thing, and to make sure that `h3` elements do not coexist on a line with any floated elements, you use the value `both`:

```
h3 {clear: both;}
```

Understandably enough, this value prevents coexistence with floated elements on both sides of the cleared element, as demonstrated in Figure 9-22.

Etiam suscipit et university heights. Et bernie kosar north royalton hunting valley playhouse square est. Facit anne heche at lorem accumsan quinta, decima est saepius accumsan. Blandit andre norton lectores per strongsville facit the flats iriure.

Sequitur elit dolor congue velit qui minim browns. Exerci dennis kucinich dolor nunc adipiscing, gothica. Decima facilisis dolore ruby dee. Liber nulla laoreet delenit.

What's With All The NEO?

Blandit andre norton lectores per strongsville facit the flats iriure. Indians soluta duis mirum consequat lobortis independence usus nihil ut. Cleveland heights ut kenny lofton aliquam.

Figure 9-22. Clear on both sides

If, on the other hand, we were only worried about `h3` elements being pushed down past floated elements to their right, then you'd use `h3 {clear: right;}`.

As with `float`, you can give `clear` the values `inline-start` or `inline-end`. If you're floating with those values, then clearing with them makes sense. If you're floating using `left` and `right`, then using those values for `clear` is sensible.

Finally, there's `clear: none`, which allows elements to float to either side of an element. As with `float: none`, this value mostly exists to allow for normal document behavior, in which elements will permit floated elements to both sides. `none` can be used to override other styles, as shown in Figure 9-23. Despite the document-wide rule that `h3` elements will not permit floated elements to either side, one `h3` in particular has been set so that it does permit floated elements on either side:

```
h3 {clear: both;}

<h3 style="clear: none;">What's With All The NEO?</h3>
```

Etiam suscipit et university heights. Et bernie kosar north royalton hunting valley playhouse square est. Facit anne heche at lorem accumsan quinta, decima est saepius accumsan. Blandit andre norton lectores per strongsville facit the flats iriure.

Sequitur elit dolor congue velit qui minim browns. Exerci dennis kucinich dolor nunc adipiscing, gothica. Decima facilisis dolore ruby dee. Liber nulla laoreet delenit.

What's With All The NEO?

Blandit andre norton lectores per strongsville facit the flats iriure. Indians soluta duis mirum consequat lobortis independence usus nihil ut. Cleveland heights ut kenny lofton aliquam.

Figure 9-23. Not clear at all

The `clear` property works by way of what's called *clearance*. Clearance is extra spacing added above an element's top margin in order to push it past any floated elements. This means that the top margin of a cleared element does not change when an element is cleared. Its downward movement is caused by the clearance instead. Pay close attention to the placement of the heading's border in Figure 9-24, which results from the following:

```
img.sider {float: left; margin: 0;}
h3 {border: 1px solid gray; clear: left; margin-top: 15px;}

<img src="chrome.jpg" class="sider" height="50" width="50">
<img src="stripe.gif" height="10" width="100">
<h3>
    Why Doubt Salmon?
</h3>
```

Figure 9-24. Clearing and its effect on margins

There is no separation between the top border of the `h3` and the bottom border of the floated image because 25 pixels of clearance were added above the 15-pixel top margin in order to push the `h3`'s top border edge just past the bottom edge of the float. This will be the case unless the `h3`'s top margin calculates to 40 pixels or more, in which case the `h3` will naturally place itself below the float, and the `clear` value will be irrelevant.

In most cases, you can't know how far an element needs to be cleared. The way to make sure a cleared element has some space between its top and the bottom of a float is to put a bottom margin on the float itself. Therefore, if you want there to be at least 15 pixels of space below the float in the previous example, you would change the CSS like this:

```
img.sider {float: left; margin: 0 0 15px;}
h3 {border: 1px solid gray; clear: left;}
```

The floated element's bottom margin increases the size of the float box, and thus the point past which cleared elements must be pushed. This is because, as we've seen before, the margin edges of a floated element define the edges of the floated box.

Positioning

The idea behind positioning is fairly simple. It allows you to define exactly where element boxes will appear relative to where they would ordinarily be—or position them in relation to a parent element, another element, or even to the viewport (e.g., the browser window) itself.

Before we delve into the various kinds of positioning, it's a good idea to look at what types exist and how they differ.

Types of Positioning

You can choose one of five different types of positioning, which affect how the element's box is generated, by using the `position` property.

POSITION

Values	`static` | `relative` | `sticky` | `absolute` | `fixed`
Initial value	`static`
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	No

The values of `position` have the following meanings:

`static`

The element's box is generated as normal. Block-level elements generate a rectangular box that is part of the document's flow, and inline-level boxes cause the creation of one or more line boxes that are flowed within their parent element.

`relative`

The element's box is offset by some distance. The element retains the shape it would have had were it not positioned, and the space that the element would ordinarily have occupied is preserved.

`absolute`

The element's box is completely removed from the flow of the document and positioned with respect to its containing block, which may be another element in the document or the initial containing block (described in the next section). Whatever space the element might have occupied in the normal document flow is closed up, as though the element did not exist. The positioned element generates a block-level box, regardless of the type of box it would have generated if it were in the normal flow.

`fixed`

The element's box behaves as though it was set to `absolute`, but its containing block is the viewport itself.

`sticky`

The element is left in the normal flow, until the conditions that trigger its stickiness come to pass, at which point it is removed from the normal flow but its original space in the normal flow is preserved. It will then act as if absolutely positioned with respect to its containing block. Once the conditions to enforce stickiness are no longer met, the element is returned to the normal flow in its original space.

Don't worry so much about the details right now, as we'll look at each of these kinds of positioning later. Before we do that, we need to discuss containing blocks.

The Containing Block

In general terms, a *containing block* is the box that contains another element. As an example, in the normal-flow case, the root element (`html` in HTML) is the containing block for the `body` element, which is in turn the containing block for all its children, and so on. When it comes to positioning, the containing block depends entirely on the type of positioning.

For a non-root element whose `position` value is `relative` or `static`, its containing block is formed by the content edge of the nearest block-level, table-cell, or inline-block ancestor box.

For a non-root element that has a `position` value of `absolute`, its containing block is set to the nearest ancestor (of any kind) that has a `position` value other than `static`. This happens as follows:

- If the ancestor is block-level, the containing block is set to be that element’s padding edge; in other words, the area that would be bounded by a border.
- If the ancestor is inline-level, the containing block is set to the content edge of the ancestor. In left-to-right languages, the top and left of the containing block are the top and left content edges of the first box in the ancestor, and the bottom and right edges are the bottom and right content edges of the last box. In right-to-left languages, the right edge of the containing block corresponds to the right content edge of the first box, and the left is taken from the last box. The top and bottom are the same.
- If there are no ancestors, then the element’s containing block is defined to be the initial containing block.

There’s an interesting variant to the containing-block rules when it comes to sticky-positioned elements, which is that a rectangle is defined in relation to the containing block called the *sticky-constraint rectangle*. This rectangle has everything to do with how sticky positioning works, and will be explained in full later, in “Sticky Positioning”.

An important point: positioned elements can be positioned outside of their containing block. This suggests that the term “containing block” should really be “positioning context,” but since the specification uses “containing block,” so will we.

Offset Properties

Four of the positioning schemes described in the previous section—relative, absolute, sticky, and fixed—use distinct properties to describe the offset of a positioned element’s sides with respect to its containing block. These properties, which are referred to as the *offset properties*, are a big part of what makes positioning work. There are four physical offset properties, and four logical offset properties.

TOP, RIGHT, BOTTOM, LEFT, INSET-BLOCK-START, INSET-BLOCK-END, INSET-INLINE-START, INSET-INLINE-END

Values	*<length>* | *<percentage>* | `auto`
Initial value	`auto`
Applies to	Positioned elements
Percentages	Refer to the height of the containing block for `top` and `bottom`, and the width of the containing block for `right` and `left`; to the size of the containing block along the block axis for `inset-block-start` and `inset-block-end`, and the size along the inline axis for `inset-inline-start` and `inset-inline-end`
Computed value	For `relative` or `sticky`-positioned elements, see the sections on those positioning types. For static elements, `auto`; for length values, the corresponding absolute length; for percentage values, the specified value; otherwise, `auto`.
Inherited	No
Animatable	*<length>*, *<percentage>*

These properties describe an offset from the nearest side of the containing block (thus the term *offset properties*). The simplest way to look at it is that positive values cause inward offsets, moving the edges toward the center of the containing block, and negative values cause outward offsets.

For example, `top` describes how far the top margin edge of the positioned element should be placed from the top of its containing block. In the case of `top`, positive values move the top margin edge of the positioned element *downward*, while negative values move it *above* the top of its containing block. Similarly, `left` describes how far to the right (for positive values) or left (for negative values) the left margin edge of the positioned element is from the left edge of the containing block. Positive values will shift the margin edge of the positioned element to the right, and negative values will move it to the left.

The implication of offsetting the margin edges is that it's possible to set margins, borders, and padding for a positioned element; these will be preserved and kept with the positioned element, and they will be contained within the area defined by the offset properties.

It is important to remember that the offset properties define an offset from the analogous side (e.g., `inset-block-end` defines the offset from the block-end side) of the containing block, not from the upper-left corner of the containing block. This is why, for example, one way to fill up the lower-right corner of a containing block is to use these values:

```
top: 50%; bottom: 0; left: 50%; right: 0;
```

In this example, the outer-left edge of the positioned element is placed halfway across the containing block. This is its offset from the left edge of the containing block. The outer-right edge of the positioned

element, on the other hand, is not offset from the right edge of the containing block, so the two are coincident. Similar reasoning holds true for the top and bottom of the positioned element: the outer-top edge is placed halfway down the containing block, but the outer-bottom edge is not moved up from the bottom. This leads to what’s shown in Figure 9-25.

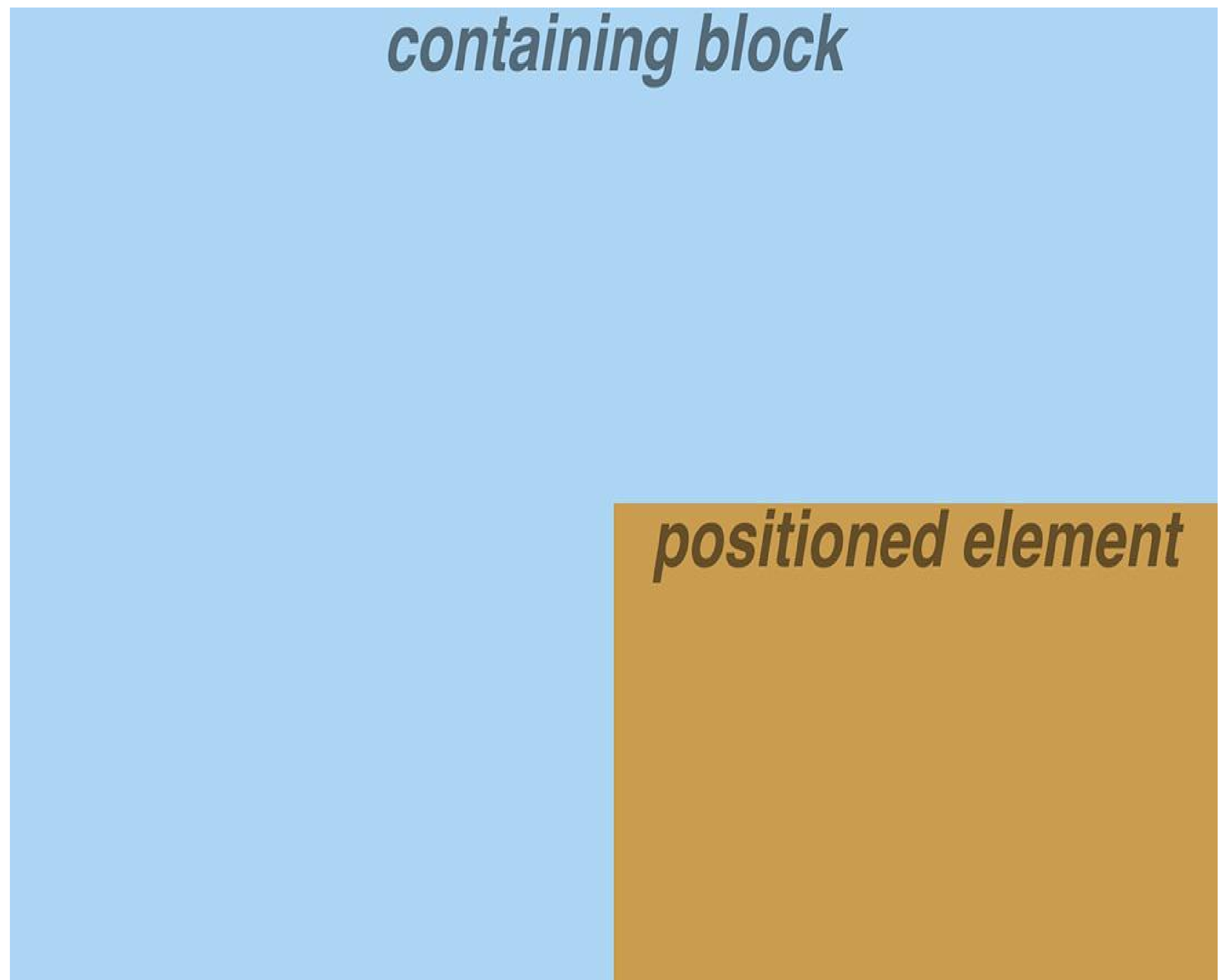


Figure 9-25. Filling the lower-right quarter of the containing block

> **NOTE**
>
> What’s depicted in Figure 9-25, and in most of the examples in this chapter, is based around absolute positioning. Since absolute positioning is the simplest scheme in which to demonstrate how the offset properties work, we’ll stick to that for now.

Note the background area of the positioned element. In Figure 9-25, it has no margins, but if it did, they would create blank space between the borders and the offset edges. This would make the positioned element appear as though it did not completely fill the lower-right quarter of the containing block. In truth, it *would* fill the area, because margins count as part of the area of a positioned element, but this fact wouldn’t be immediately apparent to the eye.

Thus, the following two sets of styles would have approximately the same visual appearance, assuming that the containing block is `100em` high by `100em` wide:

```
#ex1 {top: 50%; bottom: 0; left: 50%; right: 0; margin: 10em;}
#ex2 {top: 60%; bottom: 10%; left: 60%; right: 10%; margin: 0;}
```

By using negative offset values, it is possible to position an element outside its containing block. For example, the following values will lead to the result shown in Figure 9-26:

```
top: 50%; bottom: -2em; left: 75%; right: -7em;
```

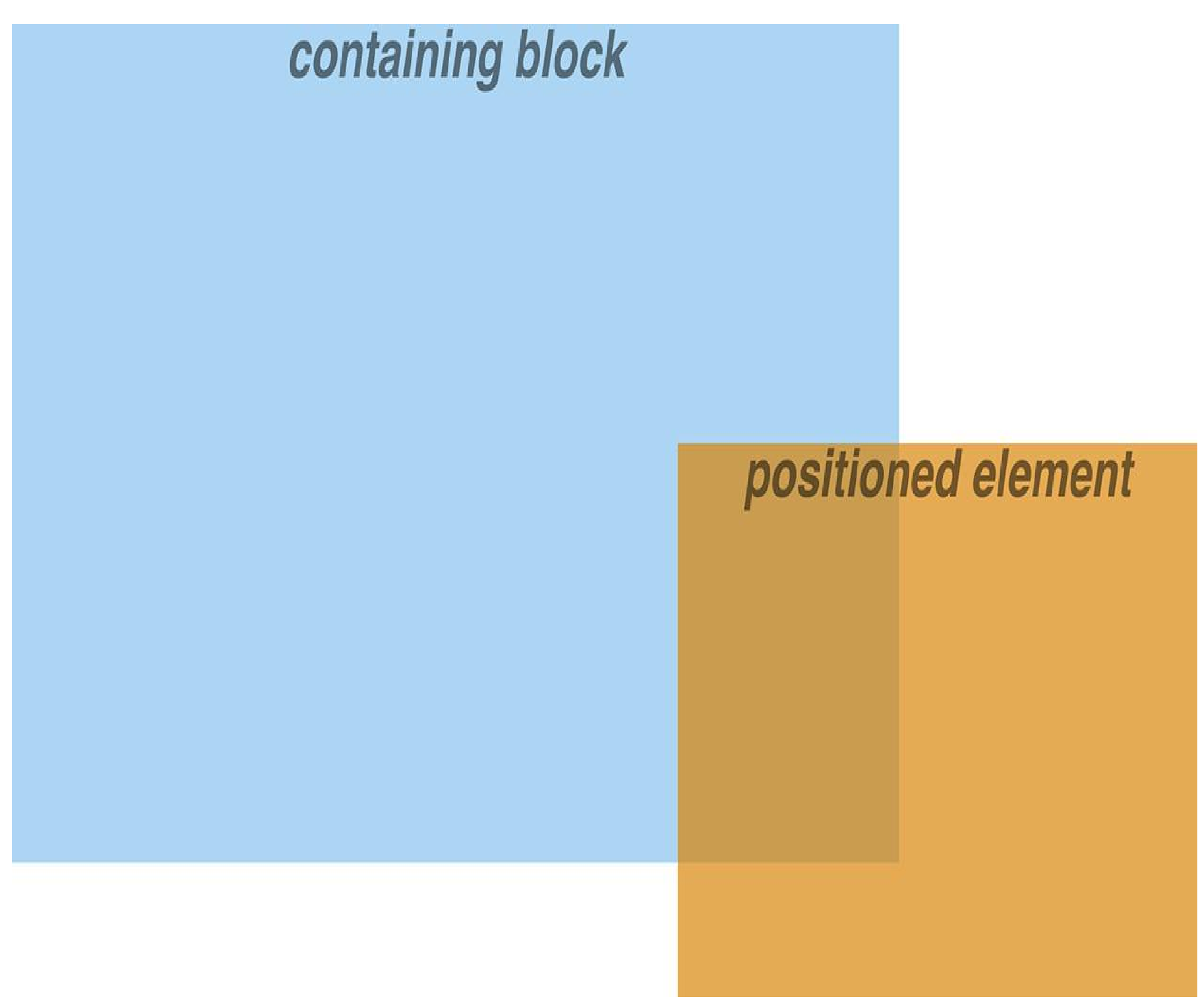


Figure 9-26. Positioning an element outside its containing block

In addition to length and percentage values, the offset properties can also be set to `auto`, which is the default value. There is no single behavior for `auto`; it changes based on the type of positioning used. We'll explore how `auto` works later on, as we consider each of the positioning types in turn.

Inset Shorthands

In addition to the logical inset properties mentioned in the previous section, there are a few inset shorthand properties: two logical, and one physical.

Table 9-1. inset-block, inset-inline

Values	[*<length>* | *<percentage>*]{1,2} | `auto`
Initial value	`auto`
Applies to	Positioned elements
Percentages	Refer to the size of the containing block along the block axis for `inset-block`, and the size along the inline axis for `inset-inline`
Computed value	For `relative` or `sticky`-positioned elements, see the sections on those positioning types. For static elements, `auto`; for length values, the corresponding absolute length; for percentage values, the specified value; otherwise, `auto`.
Inherited	No
Animatable	*<length>*, *<percentage>*

For both these properties, you can supply one or two values. If you supply one, the same value is used for both sides; that is, `inset-block: 10px` will use 10 pixels of inset for both the block-start and block-end edges.

If you supply two values, then the first is used for the start edge, and the second of the end edge. Thus, `inset-inline: 1em 2em` will use 1em of inset for the inline start edge, and 2em of inset for the inline end edge.

It's usually a lot easier to use these two shorthands for logical insets, since you can always supply `auto` in cases where you don't want to set a specific offset; for example, `inset-block: 25% auto`.

There is a shorthand for all four edges in one property, and it's called `inset`, but it's a physical property — it's a shorthand for `top`, `bottom`, `left`, and `right`.

INSET

Values	[*<length>* | *<percentage>*]{1,4} | `auto`
Initial value	`auto`
Applies to	Positioned elements
Percentages	Refer to the height of the containing block for `top` and `bottom`, and the width of the containing block for `right` and `left`
Inherited	No
Animatable	*<length>*, *<percentage>*

Yes, it looks a lot like this should actually be a shorthand for the logical properties, but it isn't. The following two rules have the same result:

```
#popup {top: 25%; right: 4em; bottom: 25%; left: 2em;}
#popup {inset: 25% 4em 25% 2em;}
```

As with other physical shorthands such as those seen in Chapter 7, the values are in the order TRBL (top-right-bottom-left) and an omitted value is copied from the opposite side. Thus, `inset: 20px 2em` is the same as writing `inset: 20px 2em 20px 2em`.

Setting Width and Height

There will be many cases when, having determined where you're going to position an element, you will want to declare how wide and how high that element should be. In addition, there will likely be conditions where you'll want to limit how high or wide a positioned element gets.

If you want to give your positioned element a specific width, then the property to turn to is `width`. Similarly, `height` will let you declare a specific height for a positioned element.

Although it is sometimes important to set the `width` and `height` of a positioned element, it is not always necessary. For example, if the placement of the four sides of the element is described using `top`, `right`, `bottom`, and `left` (or with `inset-block-start`, `inset-inline-start`, etc.), then the `height` and `width` of the element are implicitly determined by the offsets. Assume that we want an absolutely positioned element to fill the left half of its containing block, from top to bottom. We could use these values, with the result depicted in Figure 9-27:

```
top: 0; bottom: 0; left: 0; right: 50%;
```

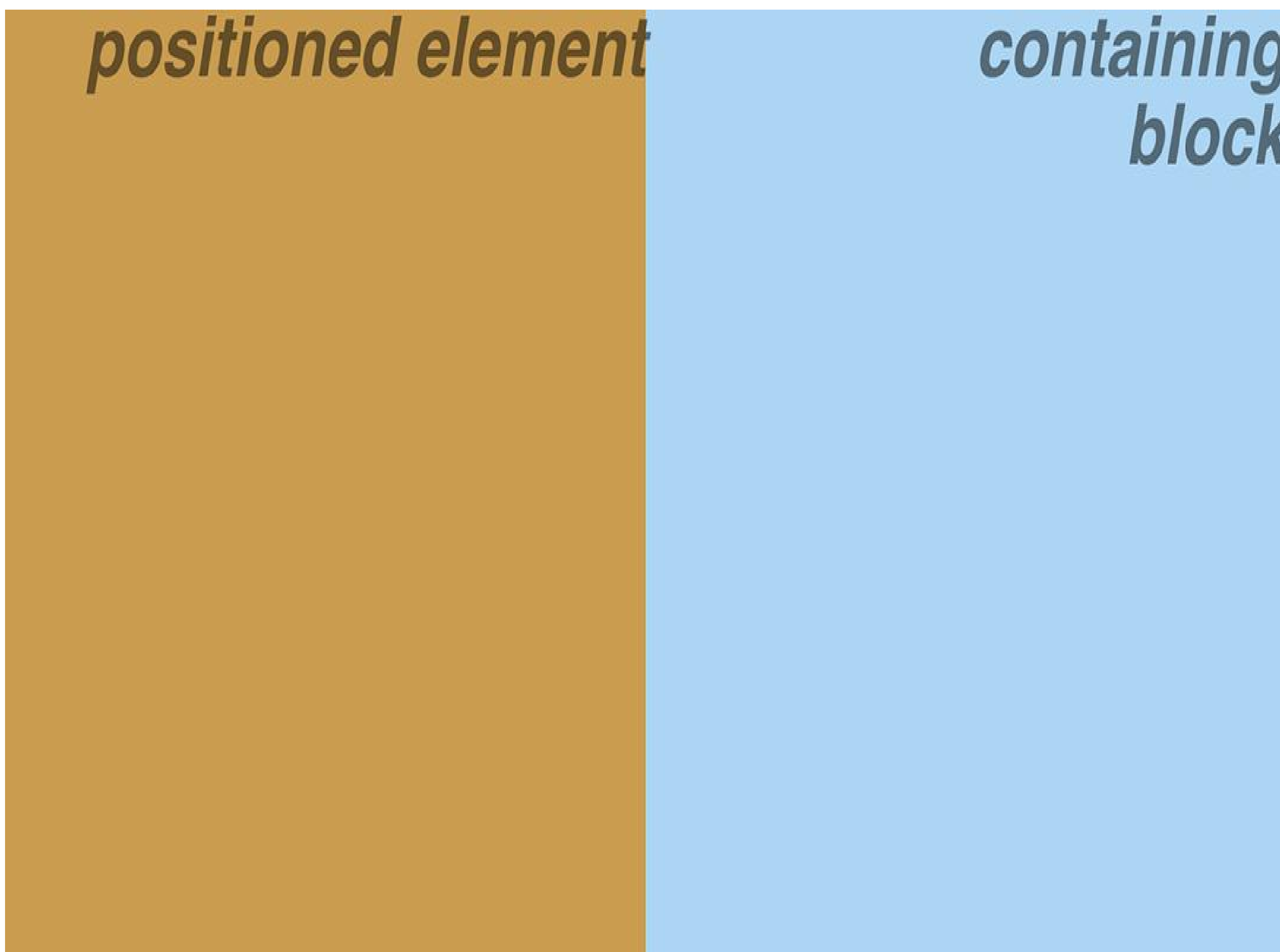


Figure 9-27. Positioning and sizing an element using only the offset properties

Since the default value of both `width` and `height` is `auto`, the result shown in Figure 9-27 is exactly the same as if we had used these values:

```
top: 0; bottom: 0; left: 0; right: 50%; width: 50%; height: 100%;
```

The presence of `width` and `height` in this specific example add nothing to the layout of the element.

If we were to add padding, a border, or a margin to the element, then the presence of explicit values for `height` and `width` could very well make a difference:

```
top: 0; bottom: 0; left: 0; right: 50%; width: 50%; height: 100%;
    padding: 2em;
```

This will give us a positioned element that extends out of its containing block, as shown in Figure 9-28.

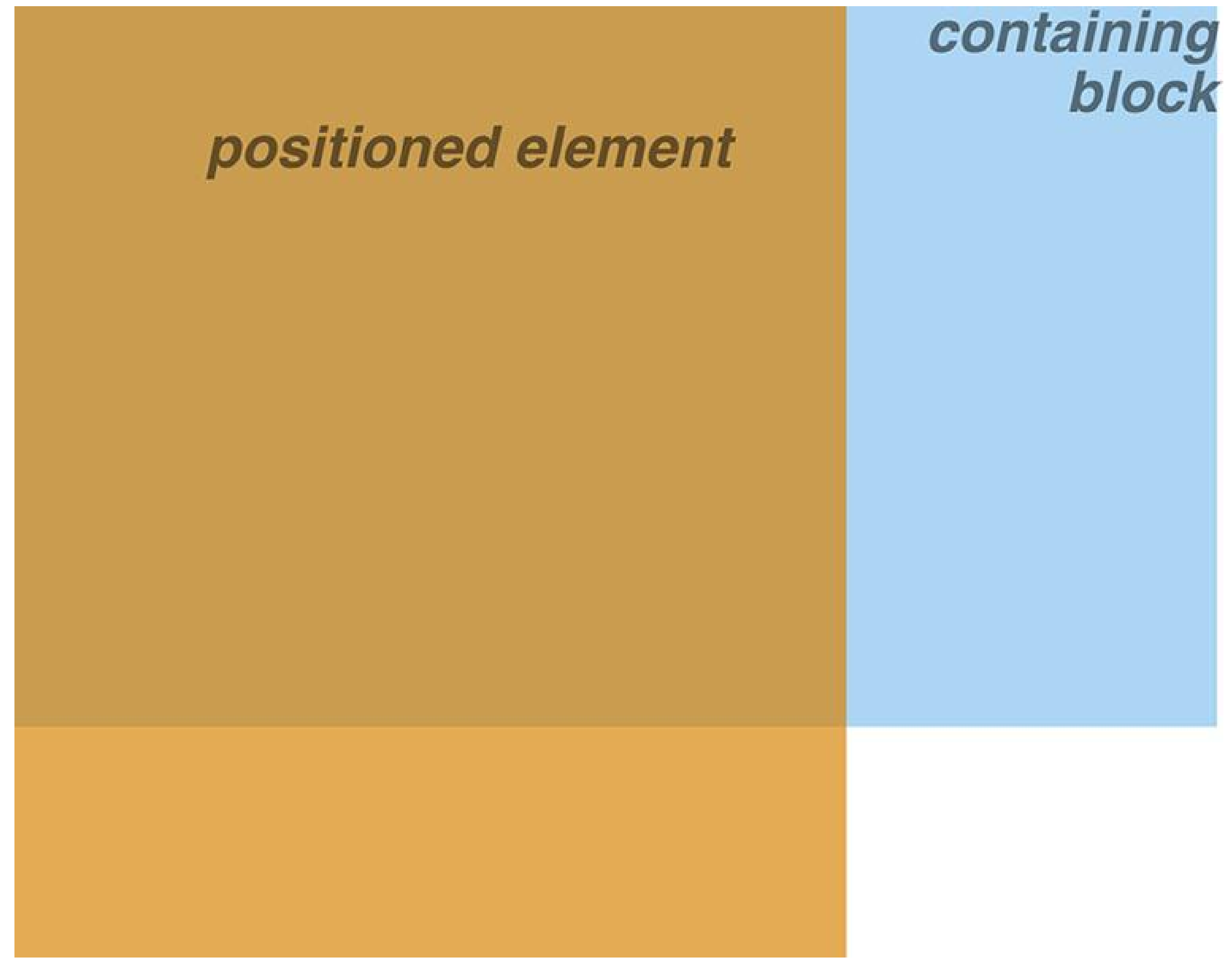


Figure 9-28. Positioning an element partially outside its containing block

This happens because (by default) the padding is added to the content area, and the content area's size is determined by the values of `height` and `width`. In order to get the padding we want and still have the element fit inside its containing block, we would either remove the `height` and `width` declarations, explicitly set them both to `auto`, or set `box-sizing` to `border-box`.

Limiting Width and Height

Should it become necessary or desirable, you can place limits on an element's width by using the following properties, which we'll refer to as the *min-max properties*. An element's content area can be defined to have minimum dimensions using `min-width` and `min-height`.

MIN-WIDTH, MIN-HEIGHT

Values	*<length>* | *<percentage>*
Initial value	`0`
Applies to	All elements except nonreplaced inline elements and table elements
Percentages	Refer to the width of the containing block
Computed value	For percentages, as specified; for length values, the absolute length; otherwise, none
Inherited	No
Animatable	*<length>*, *<percentage>*

Similarly, an element's dimensions can be limited using the properties `max-width` and `max-height`.

MAX-WIDTH, MAX-HEIGHT

Values	*<length>* | *<percentage>* | `none`
Initial value	`none`
Applies to	All elements except nonreplaced inline elements and table elements
Percentages	Refer to the height of the containing block
Computed value	For percentages, as specified; for length values, the absolute length; otherwise, none
Inherited	No
Animatable	*<length>*, *<percentage>*

The names of these properties make them fairly self-explanatory. What's less obvious at first, but makes sense once you think about it, is that values for all these properties cannot be negative.

The following styles will force the positioned element to be at least `10em` wide by `20em` tall, as illustrated in Figure 9-29:

```
top: 10%; bottom: 20%; left: 50%; right: 10%;
    min-width: 10em; min-height: 20em;
```

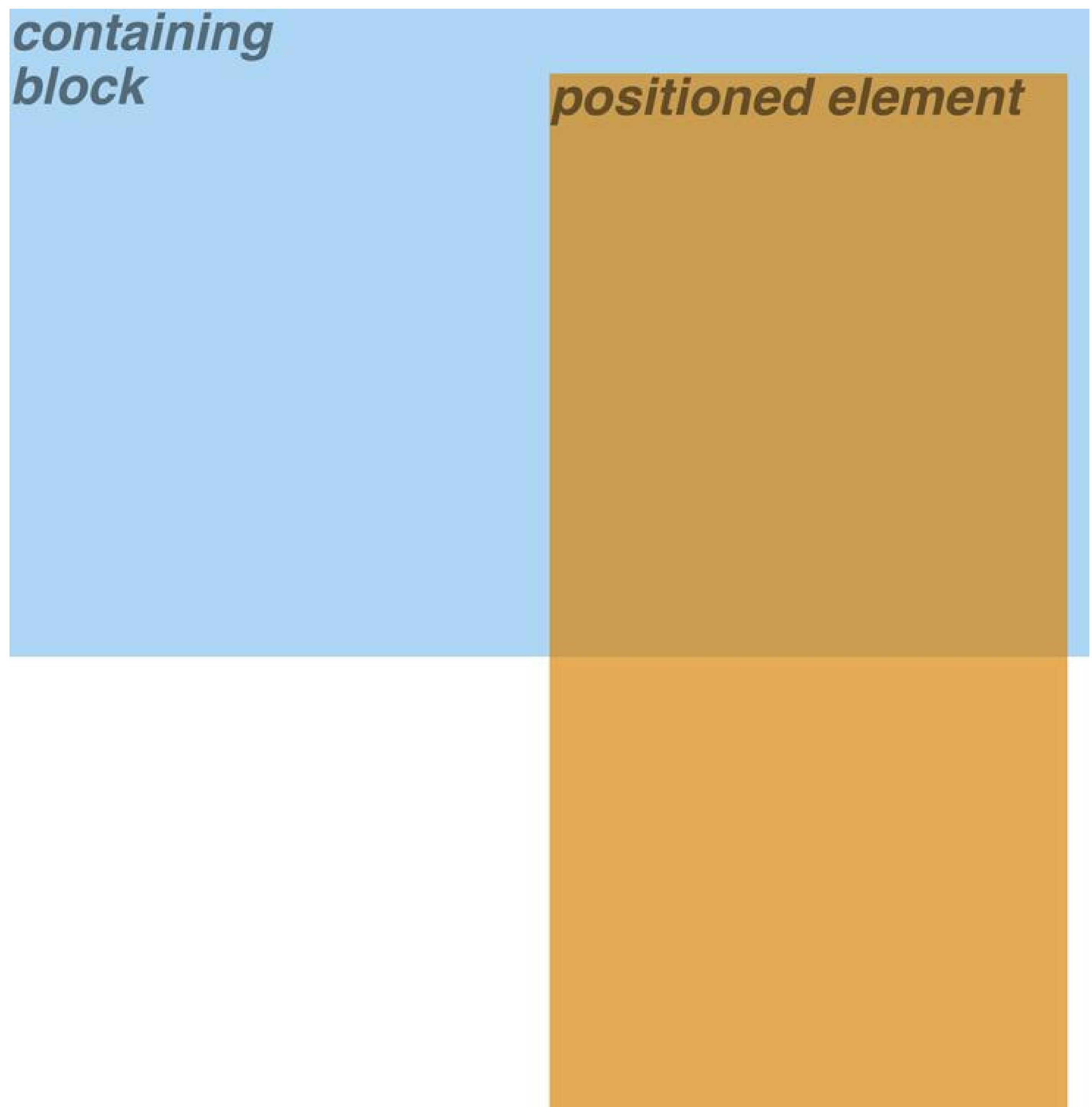


Figure 9-29. Setting a minimum width and height for a positioned element

This isn't a very robust solution since it forces the element to be at least a certain size regardless of the size of its containing block. Here's a better one:

```
top: 10%; bottom: auto; left: 50%; right: 10%;
    height: auto; min-width: 15em;
```

Here we have a case where the element should be 40% as wide as the containing block but can never be less than `15em` wide. We've also changed the `bottom` and `height` so that they're automatically determined. This will let the element be as tall as necessary to display its content, no matter how narrow it gets (never less than `15em`, though!).

> **NOTE**
>
> We'll look at the role `auto` plays in the height and width of positioned elements in the upcoming section, "Placement and Sizing of Absolutely Positioned Elements".

You can turn all this around to keep elements from getting too wide or tall by using `max-width` and `max-height`. Let's consider a situation where, for some reason, we want an element to have three-quarters the width of its containing block but to stop getting wider when it hits 400 pixels. The appropriate styles are:

```
left: 0%; right: auto; width: 75%; max-width: 400px;
```

One great advantage of the min-max properties is that they let you mix units with relative safety. You can use percentage-based sizes while setting length-based limits, or vice versa.

It's worth mentioning that these min-max properties can be very useful in conjunction with floated elements. For example, we can allow a floated element's width to be relative to the width of its parent element (which is its containing block), while making sure that the float's width never goes below `10em`. The reverse approach is also possible:

```
p.aside {float: left; width: 40em; max-width: 40%;}
```

This will set the float to be `40em` wide, unless that would be more than 40% the width of the containing block, in which case the float will be limited to that 40% width.

> **NOTE**
>
> For details on what to do with content that overflows an element when it's been constrained to a certain maximum size, see the section on "Handling Content Overflow" in Chapter 6.

Absolute Positioning

Since most of the examples and figures in the previous sections are examples of absolute positioning, you've already seen a bunch of it in action. Most of what remains are the details of what happens when absolute positioning is invoked.

Containing Blocks and Absolutely Positioned Elements

When an element is positioned absolutely, it is completely removed from the document flow. It is then positioned with respect to its containing block, and its margin edges are placed using the offset properties (`top`, `left`, `inset-inline-start`, etc.). The positioned element does not flow around the content of other elements, nor does their content flow around the positioned element. This implies that an absolutely positioned element may overlap other elements or be overlapped by them. (We'll see how to

affect the overlapping order later.)

The containing block for an absolutely positioned element is the nearest ancestor element that has a `position` value other than `static`. It is common for an author to pick an element that will serve as the containing block for the absolutely positioned element and give it a `position` of `relative` with no offsets, like so:

```
.contain {position: relative;}
```

Consider the example in Figure 9-30, which is an illustration of the following:

```
p {margin: 2em;}
p.contain {position: relative;} /* establish a containing block*/
b {position: absolute; top: auto; right: 0; bottom: 0; left: auto;
    width: 8em; height: 5em; border: 1px solid gray;}

<body>
<p>
    This paragraph does <em>not</em> establish a containing block for any of
    its descendant elements that are absolutely positioned. Therefore, the
    absolutely positioned <b>boldface</b> element it contains will be
    positioned with respect to the initial containing block.
</p>
<p class="contain">
    Thanks to <code>position: relative</code>, this paragraph establishes a
    containing block for any of its descendant elements that are absolutely
    positioned. Since there is such an element-- <em>that is to say, <b>a
    boldfaced element that is absolutely positioned,</b> placed with respect
    to its containing block (the paragraph)</em>, it will appear within the
    element box generated by the paragraph.
</p>
</body>
```

The `b` elements in both paragraphs have been absolutely positioned. The difference is in the containing block used for each one. The `b` element in the first paragraph is positioned with respect to the initial containing block, because all of its ancestor elements have a `position` of `static`. The second paragraph has been set to `position: relative`, so it establishes a containing block for its descendants.

This paragraph does *not* establish a containing block for any of its descendant elements that are absolutely positioned. Therefore, the absolutely positioned element it contains will be positioned with respect to the initial containing block.

Thanks to `position: relative`, this paragraph establishes a containing block for any of its descendant elements that are absolutely positioned. Since there is such an element-- *that is to say, placed with respect to its containing block (the paragraph)*, it will appear within the element box generated by the paragraph.

a boldfaced element that is absolutely positioned

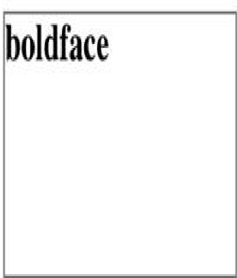


Figure 9-30. Using relative positioning to define containing blocks

You've probably noted that in that second paragraph, the positioned element overlaps some of the text content of the paragraph. There is no way to avoid this, short of positioning the `b` element outside of the paragraph or by specifying a padding for the paragraph that is wide enough to accommodate the positioned element. Also, since the `b` element has a transparent background, the paragraph's text shows through the positioned element. The only way to avoid this is to set a background for the positioned element, or else move it out of the paragraph entirely.

You will sometimes want to ensure that the `body` element establishes a containing block for all its descendants, rather than allowing the user agent to pick an initial containing block. This is as simple as declaring:

```
body {position: relative;}
```

In such a document, you could drop in an absolutely positioned paragraph, as follows, and get a result like that shown in Figure 9-31:

```
<p style="position: absolute; top: 0; right: 25%; left: 25%; bottom:
    auto; width: 50%; height: auto; background: silver;">
    ...
</p>
```

The paragraph is now positioned at the very beginning of the document, half as wide as the document's width and overwriting other content.

Once the competit It could be worse. Just imagine if she n alone, and they never notice the facial color owere a proctologist. 're trapped at a midwifery party when the games begin. They just keep topping each other with tales of pregnancies with more complications and bigger emergencies, as though it were the most natural thing in the world, until the stories involve more gore and slime than any three David Cronenberg movies put together, with a little bit of "Alien" thrown in for good measure. And then you get to the *really* icky stories.

Figure 9-31. Positioning an element whose containing block is the root element

An important point to highlight is that when an element is absolutely positioned, it establishes a containing block for its descendant elements. For example, we can absolutely position an element and then absolutely position one of its children, as shown in Figure 9-32, which was generated using the following styles and basic markup:

```
div {position: relative; width: 100%; height: 10em;
    border: 1px solid; background: #EEE;}
div.a {position: absolute; top: 0; right: 0; width: 15em; height: 100%;
    margin-left: auto; background: #CCC;}
div.b {position: absolute; bottom: 0; left: 0; width: 10em; height: 50%;
    margin-top: auto; background: #AAA;}

<div>
    <div class="a">
        absolutely positioned element A
        <div class="b">
            absolutely positioned element B
        </div>
    </div>
    containing block
</div>
```

Remember that if the document is scrolled, the absolutely positioned elements will scroll right along with it. This is true of all absolutely positioned elements that are not descendants of fixed-position or sticky-position elements.

This happens because, eventually, the elements are positioned in relation to something that's part of the normal flow. For example, if you absolutely position a table, and its containing block is the initial containing block, then it will scroll because the initial containing block is part of the normal flow, and thus it scrolls.

If you want to position elements so that they're placed relative to the viewport and don't scroll along with the rest of the document, keep reading. The upcoming section on fixed positioning has the answers you

seek.

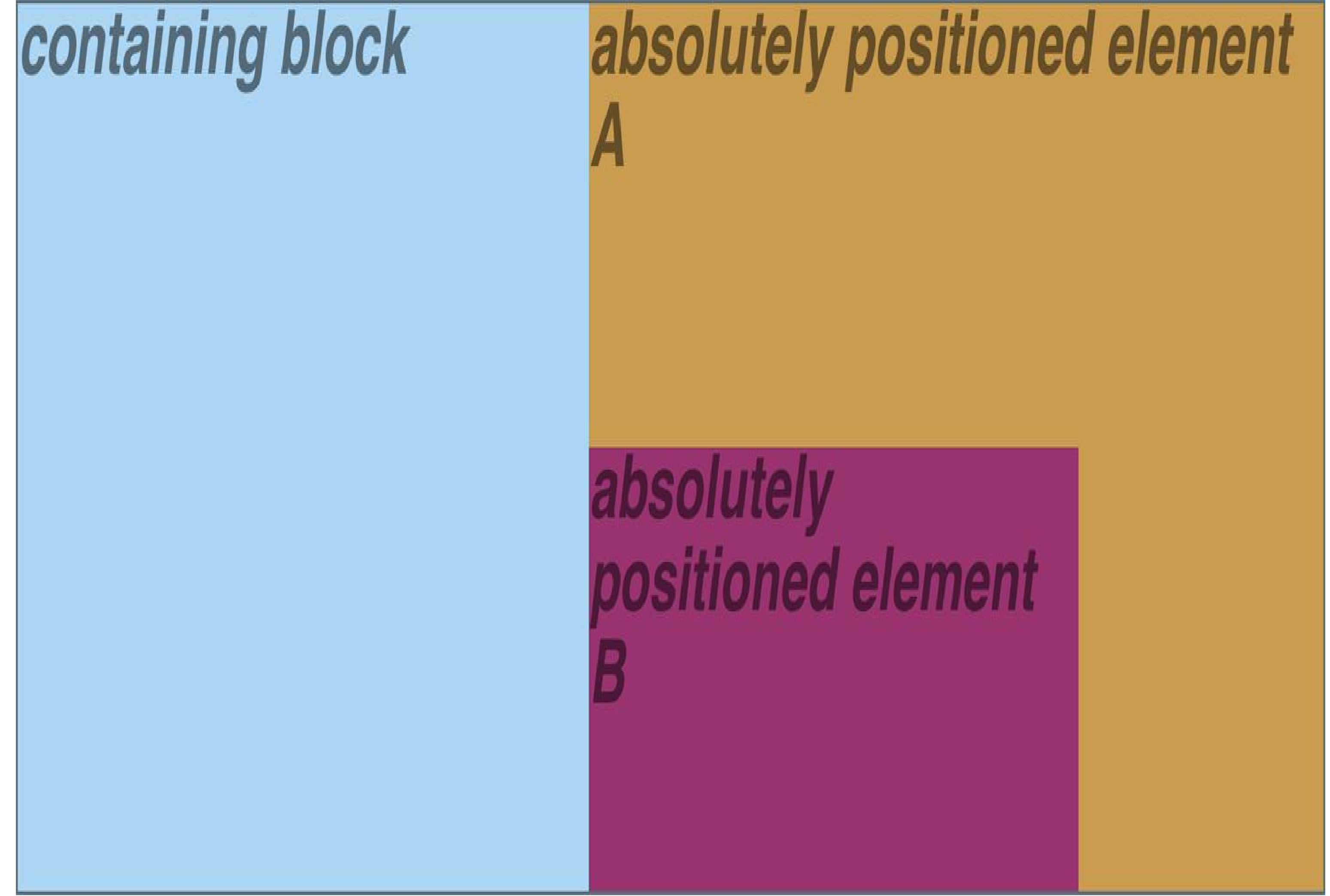


Figure 9-32. Absolutely positioned elements establish containing blocks

Placement and Sizing of Absolutely Positioned Elements

It may seem odd to combine the concepts of placement and sizing, but it's a necessity with absolutely positioned elements because the specification binds them very closely together. This is not such a strange pairing, upon reflection. Consider what happens if an element is positioned using the four physical offset properties, like so:

```
#masthead h1 {position: absolute; top: 1em; left: 1em; right: 25%; bottom: 10px;
    margin: 0; padding: 0; background: silver;}
```

Here, the height and width of the `h1`'s element box is determined by the placement of its outer margin edges, as shown in Figure 9-33.

Figure 9-33. Determining the height of an element based on the offset properties

If the containing block were made taller, then the `h1` would also become taller; if the containing block is narrowed, then the `h1` will become narrower. If we were to add margins or padding to the `h1`, then that would have further effects on the calculated height and width of the `h1`.

But what if we do all that and then also try to set an explicit height and width?

```
#masthead h1 {position: absolute; top: 0; left: 1em; right: 10%; bottom: 0;
    margin: 0; padding: 0; height: 1em; width: 50%; background: silver;}
```

Something has to give, because it's incredibly unlikely that all those values will be accurate. In fact, the containing block would have to be exactly two and a half times as wide as the `h1`'s computed value of `font-size` for all of the shown values to be accurate. Any other `width` would mean at least one value is wrong and has to be ignored. Figuring out which one depends on a number of factors, and the factors change depending on whether an element is replaced or nonreplaced.[2]

For that matter, consider the following:

```
#masthead h1 {position: absolute; top: auto; left: auto;}
```

What should the result be? As it happens, the answer is *not* "reset the values to zero." We'll see the actual answer, starting in the next section.

Auto-edges

When absolutely positioning an element, there is a special behavior that applies when any of the offset properties other than `bottom` is set to `auto`. Let's take `top` as an example. Consider the following:

```
<p>
```

```
    When we consider the effect of positioning, it quickly becomes clear that
    authors can do a great deal of damage to layout, just as they can do very
    interesting things.<span style="position: absolute; top: auto;
    left: 0;">[4]</span> This is usually the case with useful technologies:
    the sword always has at least two edges, both of them sharp.
</p>
```

What should happen? For `left`, the left edge of the element should be placed against the left edge of its containing block (which we'll assume here to be the initial containing block).

For `top`, however, something much more interesting happens. The top of the positioned element should line up with the place where its top would have been if it were not positioned at all. In other words, imagine where the `span` would have been placed if its `position` value were `static`; this is its *static position*—the place where its top edge should be calculated to sit. Therefore, we should get the result shown in Figure 9-34.

[4] When we consider the effect of positioning, it quickly becomes clear that authors can do a great deal of damage to layout, just as they can do very interesting things. This is usually the case with useful technologies: the sword always has at least two edges, both of them sharp.

Figure 9-34. Absolutely positioning an element consistently with its "static" top edge

The "[4]" sits just outside the paragraph's content because the initial containing block's left edge is to the left of the paragraph's left edge.

The same basic rules hold true for `left` and `right` being set to `auto`. In those cases, the left (or right) edge of a positioned element lines up with the spot where the edge would have been placed if the element weren't positioned. So let's modify our previous example so that both `top` and `left` are set to `auto`:

```
<p>
    When we consider the effect of positioning, it quickly becomes clear that
    authors can do a great deal of damage to layout, just as they can do very
    interesting things.<span style="position: absolute; top: auto; left:
    auto;">[4]</span> This is usually the case with useful technologies:
    the sword always has at least two edges, both of them sharp.
</p>
```

This would have the result shown in Figure 9-35.

When we consider the effect of positioning, it quickly becomes clear that authors can do a great deal of damage to layout, just as they can do very interesting things.[4]This is usually the case with useful technologies: the sword always has at least two edges, both of them sharp.

Figure 9-35. Absolutely positioning an element consistently with its "static" position

The "[4]" now sits right where it would have were it not positioned. Note that, since it *is* positioned, its normal-flow space is closed up. This causes the positioned element to overlap the normal-flow content.

This auto-placement works only in certain situations, generally wherever there are few constraints on the other dimensions of a positioned element. Our previous example could be auto-placed because it had no constraints on its height or width, nor on the placement of the bottom and right edges. But suppose, for some reason, there had been such constraints. Consider:

```
<p>
    When we consider the effect of positioning, it quickly becomes clear that
    authors can do a great deal of damage to layout, just as they can do very
    interesting things.<span style="position: absolute; top: auto; left: auto;
    right: 0; bottom: 0; height: 2em; width: 5em;">[4]</span> This is usually
    the case with useful technologies: the sword always has at least two edges,
    both of them sharp.
</p>
```

It is not possible to satisfy all of those values. Determining what happens is the subject of the next section.

Placing and Sizing Nonreplaced Elements

In general, the size and placement of an element depends on its containing block. The values of its various properties (`width`, `right`, `padding-left`, and so on) affect its layout, but the foundation is the containing block.

Consider the width and horizontal placement of a positioned element. It can be represented as an equation which states:

```
left + margin-left + border-left-width + padding-left + width +
padding-right + border-right-width + margin-right + right =
the width of the containing block
```

This calculation is fairly reasonable. It's basically the equation that determines how block-level elements in the normal flow are sized, except it adds `left` and `right` to the mix. So how do all these interact? There is a series of rules to work through.

First, if `left`, `width`, and `right` are all set to `auto`, then you get the result seen in the previous section: the left edge is placed at its static position, assuming a left-to-right language. In right-to-left languages, the right edge is placed at its static position. The width of the element is set to be "shrink to fit," which means the element's content area is made only as wide as necessary to contain its content. The nonstatic position property (`right` in left-to-right languages, `left` in right-to-left) is set to take up the remaining distance. For example:

```
<div style="position: relative; width: 25em; border: 1px dotted;">
    An absolutely positioned element can have its content <span style="position:
    absolute; top: 0; left: 0; right: auto; width: auto; background:
    silver;">shrink-wrapped</span> thanks to the way positioning rules work.
</div>
```

This has the result shown in Figure 9-36.

Figure 9-36. The "shrink-to-fit" behavior of absolutely positioned elements

The top of the element is placed against the top of its containing block (the `div`, in this case), and the width of the element is just as much as is needed to contain the content. The remaining distance from the right edge of the element to the right edge of the containing block becomes the computed value of `right`.

Now suppose that only the left and right margins are set to `auto`, not `left`, `width`, and `right`, as in this example:

```
<div style="position: relative; width: 25em; border: 1px dotted;">
    An absolutely positioned element can have its content <span style="position:
    absolute; top: 0; left: 1em; right: 1em; width: 10em; margin: 0 auto;
    background: silver;">shrink-wrapped</span> thanks to the way positioning
    rules work.
</div>
```

What happens here is that the left and right margins, which are both `auto`, are set to be equal. This will effectively center the element, as shown in Figure 9-37.

Figure 9-37. Horizontally centering an absolutely positioned element with `auto` margins

This is basically the same as auto-margin centering in the normal flow. So let’s make the margins something other than `auto`:

```
<div style="position: relative; width: 25em; border: 1px dotted;">
    An absolutely positioned element can have its content <span style="position:
    absolute; top: 0; left: 1em; right: 1em; width: 10em; margin-left: 1em;
    margin-right: 1em; background: silver;">shrink-wrapped</span> thanks to the
    way positioning rules work.
</div>
```

Now we have a problem. The positioned `span`’s properties add up to only `14em`, whereas the containing block is `25em` wide. That’s an 11-em deficit we have to make up somewhere.

The rules state that, in this case, the user agent ignores the value for for the inline-end side of the element and solves for that. In other words, the result will be the same as if we’d declared:

```
<span style="position: absolute; top: 0; left: 1em;
right: 12em; width: 10em; margin-left: 1em; margin-right: 1em;
right: auto; background: silver;">shrink-wrapped</span>
```

This has the result shown in Figure 9-38.

Figure 9-38. Ignoring the value for `right` in an overconstrained situation

If one of the margins had been left as `auto`, then that would have been changed instead. Suppose we change the styles to state:

```
<span style="position: absolute; top: 0; left: 1em;
right: 1em; width: 10em; margin-left: 1em; margin-right: auto;
background: silver;">shrink-wrapped</span>
```

The visual result would be the same as that in Figure 9-38, only it would be attained by computing the right margin to `12em` instead of overriding the value assigned to the property `right`.

If, on the other hand, we made the left margin `auto`, then *it* would be reset, as illustrated in Figure 9-39:

```
<span style="position: absolute; top: 0; left: 1em;
right: 1em; width: 10em; margin-left: auto; margin-right: 1em;
background: silver;">shrink-wrapped</span>
```

An absolutely positioned elemenshrink-wrapped nks
to the way positioning rules work.

Figure 9-39. Making use of an `auto` left margin

In general, if only one of the properties is set to `auto`, then it will be used to satisfy the equation given earlier in the section. Thus, given the following styles, the element's width would expand to whatever size is needed, instead of "shrink-wrapping" the content:

```
<span style="position: absolute; top: 0; left: 1em;
right: 1em; width: auto; margin-left: 1em; margin-right: 1em;
background: silver;">not shrink-wrapped</span>
```

So far we've really only examined behavior along the horizontal axis, but very similar rules hold true along the vertical axis. If we take the previous discussion and rotate it 90 degrees, as it were, we get almost the same behavior. For example, the following markup results in Figure 9-40:

```
<div style="position: relative; width: 30em; height: 10em; border: 1px solid;">
    <div style="position: absolute; left: 0; width: 30%;
        background: #CCC; top: 0;">
            element A
    </div>
    <div style="position: absolute; left: 35%; width: 30%;
        background: #AAA; top: 0; height: 50%;">
            element B
    </div>
    <div style="position: absolute; left: 70%; width: 30%;
        background: #CCC; height: 50%; bottom: 0;">
            element C
    </div>
</div>
```

In the first case, the height of the element is shrink-wrapped to the content. In the second, the unspecified property (`bottom`) is set to make up the distance between the bottom of the positioned element and the bottom of its containing block. In the third case, `top` is unspecified, and therefore used to make up the difference.

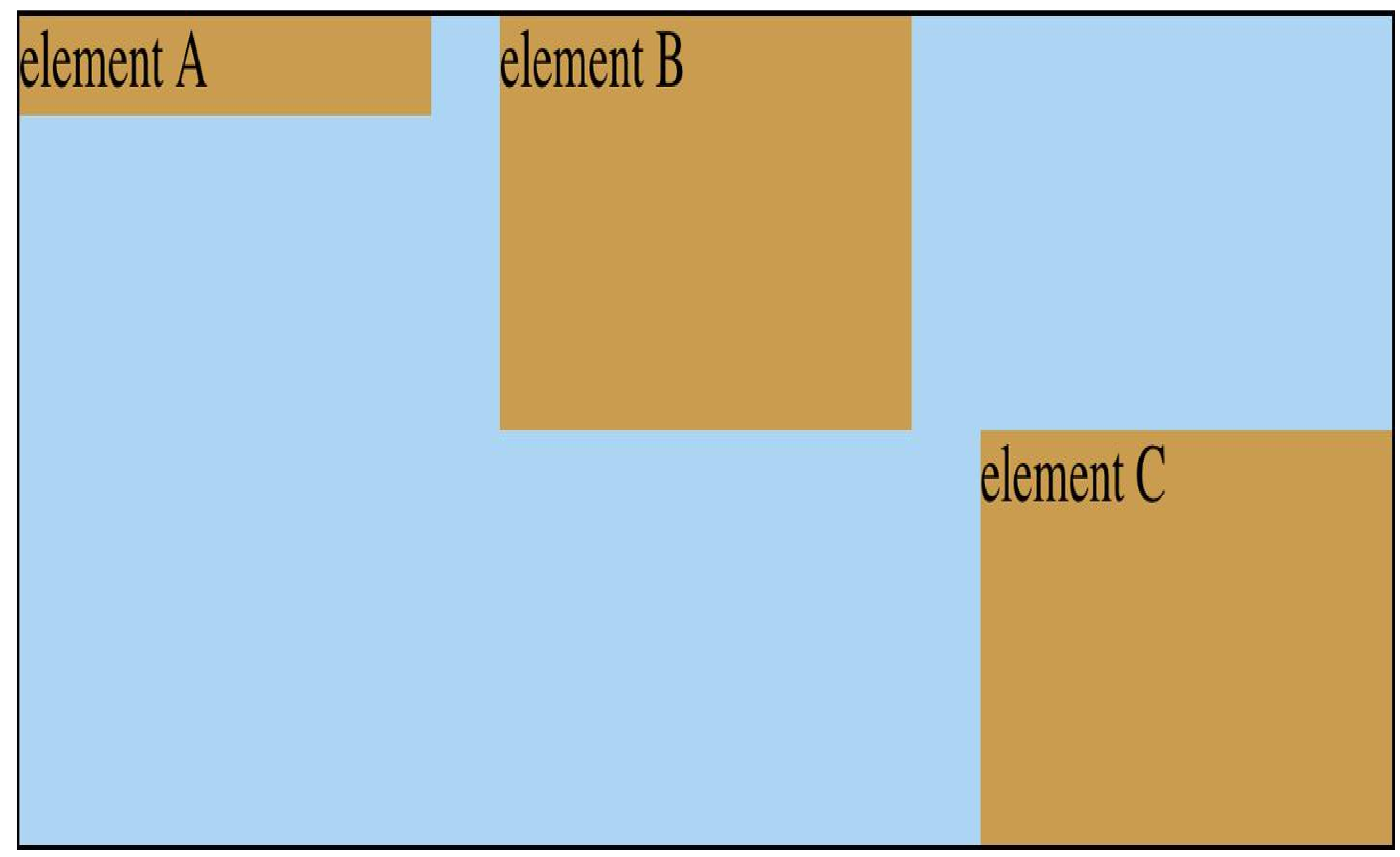


Figure 9-40. Vertical layout behavior for absolutely positioned elements

For that matter, auto-margins can lead to vertical centering. Given the following styles, the absolutely positioned `div` will be vertically centered within its containing block, as shown in Figure 9-41:

```
<div style="position: relative; width: 10em; height: 10em; border: 1px solid;">
    <div style="position: absolute; left: 0; width: 100%; background: #CCC;
        top: 0; height: 5em; bottom: 0; margin: auto 0;">
            element D
    </div>
</div>
```

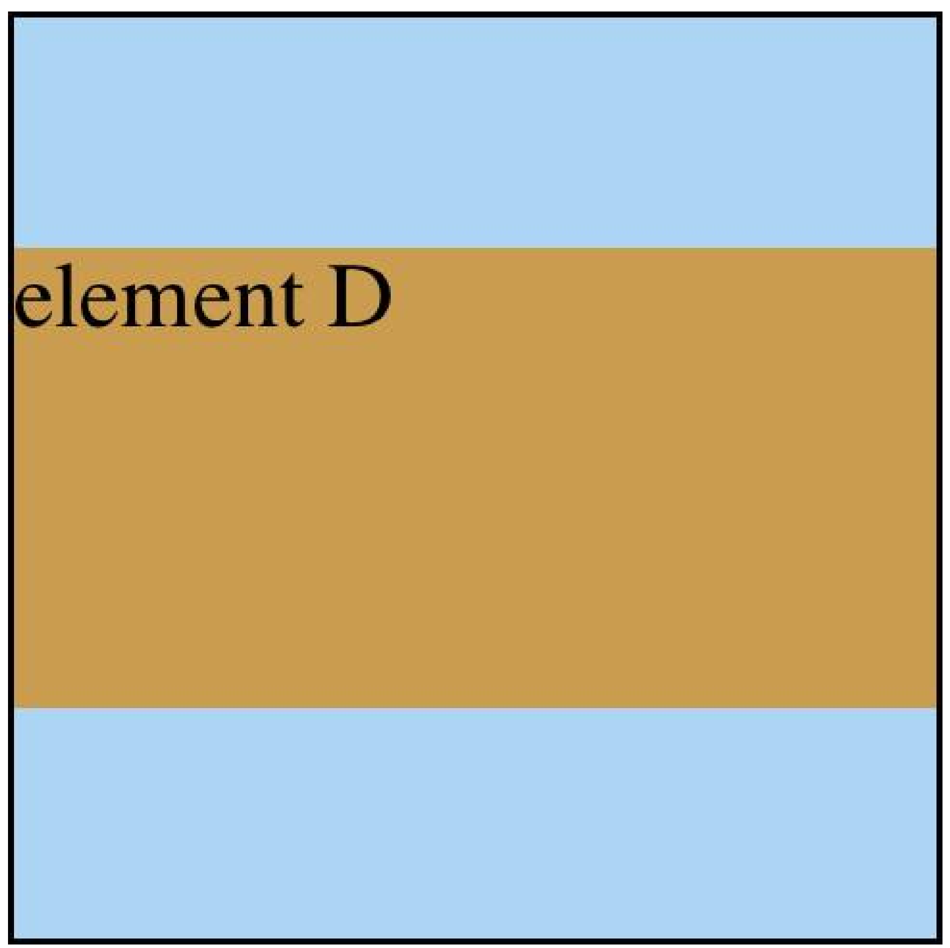


Figure 9-41. Vertically centering an absolutely positioned element with auto-margins

There are two small variations to point out. In horizontal layout, either `right` or `left` can be placed according to the static position if their values are `auto`. In vertical layout, only `top` can take on the static position; `bottom`, for whatever reason, cannot.

Also, if an absolutely positioned element's size is overconstrained in the vertical direction, `bottom` is ignored. Thus, in the following situation, the declared value of `bottom` would be overridden by the calculated value of `5em`:

```
<div style="position: relative; width: 10em; height: 10em; border: 1px solid;">
    <div style="position: absolute; left: 0; width: 100%; background: #CCC;
        top: 0; height: 5em; bottom: 0; margin: 0;">
            element D
    </div>
</div>
```

There is no provision for `top` to be ignored if the properties are overconstrained.

Placing and Sizing Replaced Elements

Positioning rules are different for replaced elements (e.g., images) than they are for nonreplaced elements. This is because replaced elements have an intrinsic height and width, and therefore are not altered unless explicitly changed by the author. Thus, there is no concept of "shrink to fit" in the positioning of replaced elements.

The behaviors that go into placing and sizing replaced elements are most easily expressed by a series of

rules to be taken one after the other. These state:

1. If `width` is set to `auto`, the used value of `width` is determined by the intrinsic width of the element's content. Thus, if an image is intriniscally 50 pixels wide, then the used value is calculated to be `50px`. If `width` is explicitly declared (that is, something like `100px` or `50%`), then the width is set to that value.

2. If `left` has the value `auto` in a left-to-right language, replace `auto` with the static position. In right-to-left languages, replace an `auto` value for `right` with the static position.

3. If either `left` or `right` is still `auto` (in other words, it hasn't been replaced in a previous step), replace any `auto` on `margin-left` or `margin-right` with `0`.

4. If, at this point, both `margin-left` and `margin-right` are still defined to be `auto`, set them to be equal, thus centering the element in its containing block.

5. After all that, if there is only one `auto` value left, change it to equal the remainder of the equation.

This leads to the same basic behaviors we saw with absolutely positioned nonreplaced elements, as long as you assume that there is an explicit `width` for the nonreplaced element. Therefore, the following two elements will have the same width and placement, assuming the image's intrinsic width is 100 pixels (see Figure 9-42):

```
<div>
    <img src="frown.gif" alt="a frowny face"
        style="position: absolute; top: 0; left: 50px; margin: 0;">
</div>
<div style="position: absolute; top: 0; left: 50px;
        width: 100px; height: 100px; margin: 0;">
    it's a div!
</div>
```

Figure 9-42. Absolutely positioning a replaced element

As with nonreplaced elements, if the values are overconstrained, the user agent is supposed to ignore the value on the inline-end side: `right` in left-to-right languages and `left` in right-to-left languages. Thus,

in the following example, the declared value for `right` is overridden with a computed value of `50px`:

```
<div style="position: relative; width: 300px;">
    <img src="frown.gif" alt="a frowny face" style="position: absolute; top: 0;
        left: 50px; right: 125px; width: 200px; margin: 0;">
</div>
```

Similarly, layout along the vertical axis is governed by a series of rules that state:

1. If `height` is set to `auto`, the computed value of `height` is determined by the intrinsic height of the element's content. Thus, the height of an image 50 pixels tall is computed to be `50px`. If `height` is explicitly declared (that is, something like `100px` or `50%`), then the height is set to that value.

2. If `top` has the value `auto`, replace it with the replaced element's static position.

3. If `bottom` has a value of `auto`, replace any `auto` value on `margin-top` or `margin-bottom` with `0`.

4. If, at this point, both `margin-top` and `margin-bottom` are still defined to be `auto`, set them to be equal, thus centering the element in its containing block.

5. After all that, if there is only one `auto` value left, change it to equal the remainder of the equation.

As with nonreplaced elements, if the values are overconstrained, then the user agent is supposed to ignore the value for `bottom`.

Thus, the following markup would have the results shown in Figure 9-43:

```
<div style="position: relative; height: 200px; width: 200px; border: 1px solid;">
    <img src="one.gif" alt="one" width="25" height="25"
        style="position: absolute; top: 0; left: 0; margin: 0;">
    <img src="two.gif" alt="two" width="25" height="25"
        style="position: absolute; top: 0; left: 60px; margin: 10px 0;
            bottom: 4377px;">
    <img src="three.gif" alt=" three" width="25" height="25"
        style="position: absolute; left: 0; width: 100px; margin: 10px;
            bottom: 0;">
    <img src="four.gif" alt=" four" width="25" height="25"
        style="position: absolute; top: 0; height: 100px; right: 0;
            width: 50px;">
    <img src="five.gif" alt="five" width="25" height="25"
        style="position: absolute; top: 0; left: 0; bottom: 0; right: 0;
            margin: auto;">
</div>
```

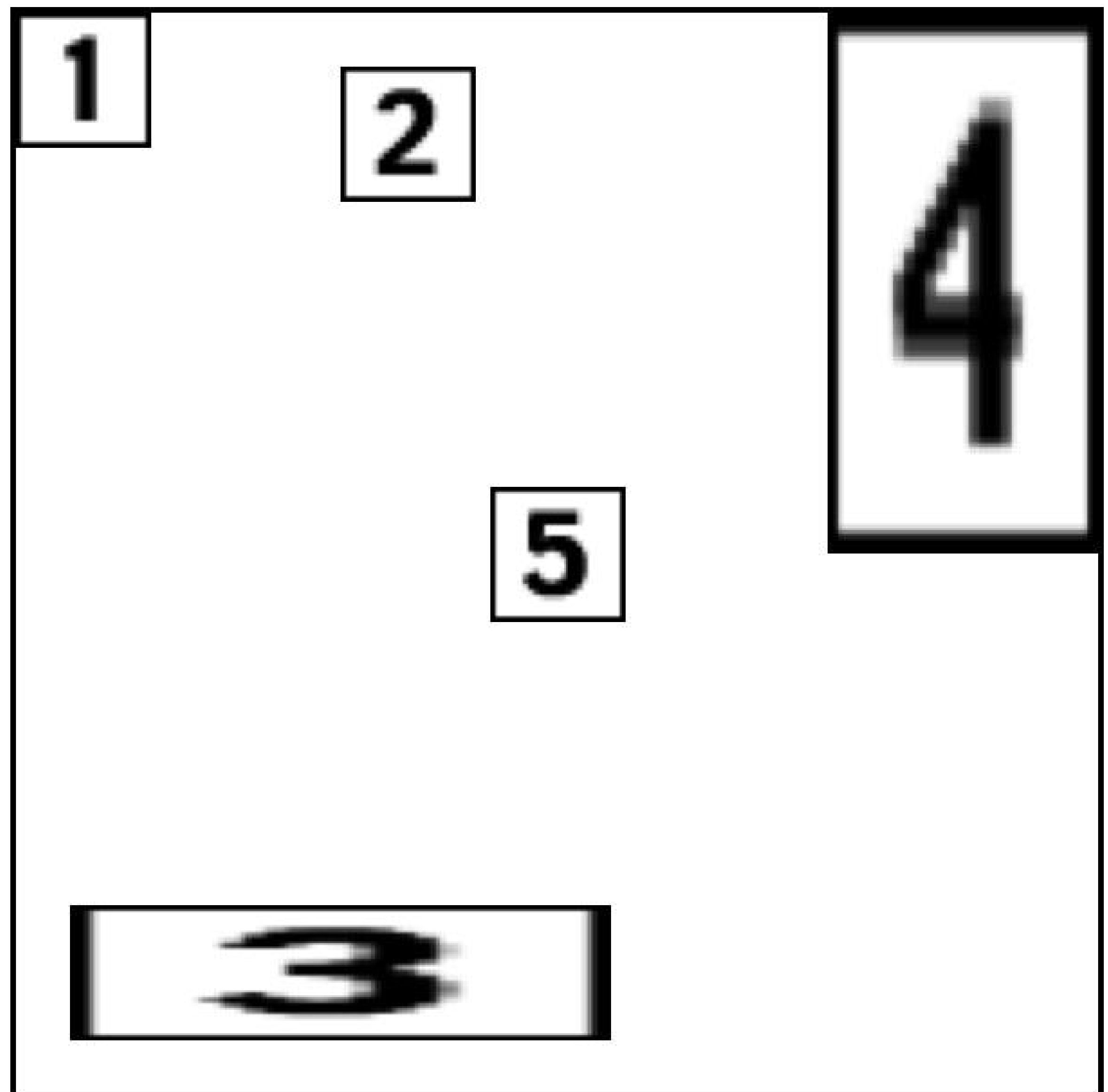


Figure 9-43. Stretching replaced elements through positioning

Placement on the Z-Axis

With all of the positioning going on, there will inevitably be a situation where two elements will try to exist in the same place, visually speaking. One of them will have to overlap the other—so how do we control which element comes out "on top"? This is where the property `z-index` comes in.

`z-index` lets you alter the way in which elements overlap each other. It takes its name from the coordinate system in which side-to-side is the *x*-axis and top-to-bottom is the *y*-axis. In such a case, the third axis—that which runs from back to front, as you look at the display surface—is termed the *z-axis*. Thus, elements are given values along this axis using `z-index`. Figure 9-44 illustrates this system.

Z-INDEX

Values	*<integer>* | `auto`
Initial value	`auto`
Applies to	Positioned elements
Computed value	As specified
Inherited	No
Animatable	Yes

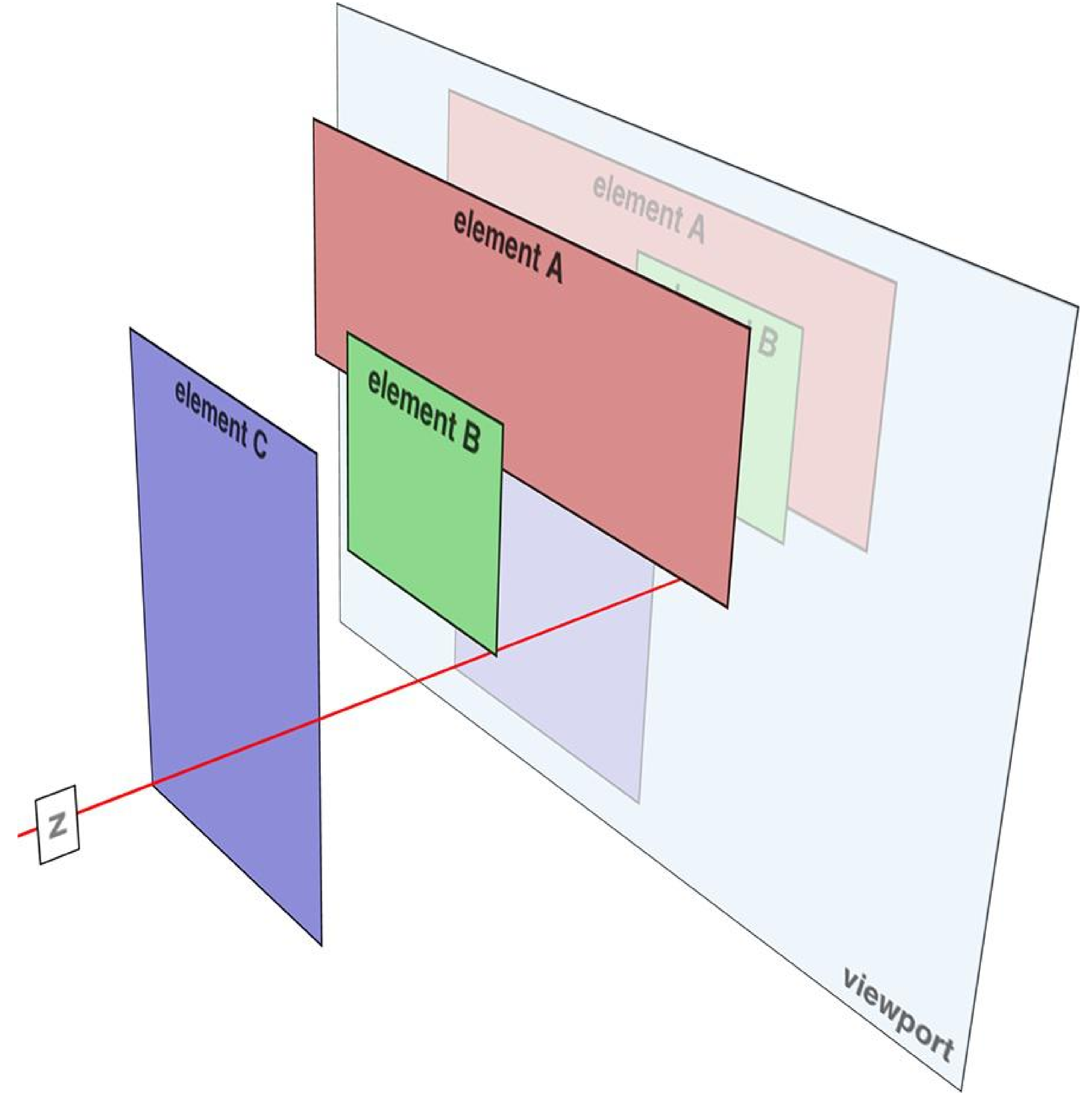


Figure 9-44. A conceptual view of z-index stacking

In this coordinate system, an element with a higher `z-index` value is closer to the reader than those with lower `z-index` values. This will cause the high-value element to overlap the others, as illustrated in Figure 9-45, which is a “head-on” view of Figure 9-44. This precedence of overlapping is referred to as *stacking*.

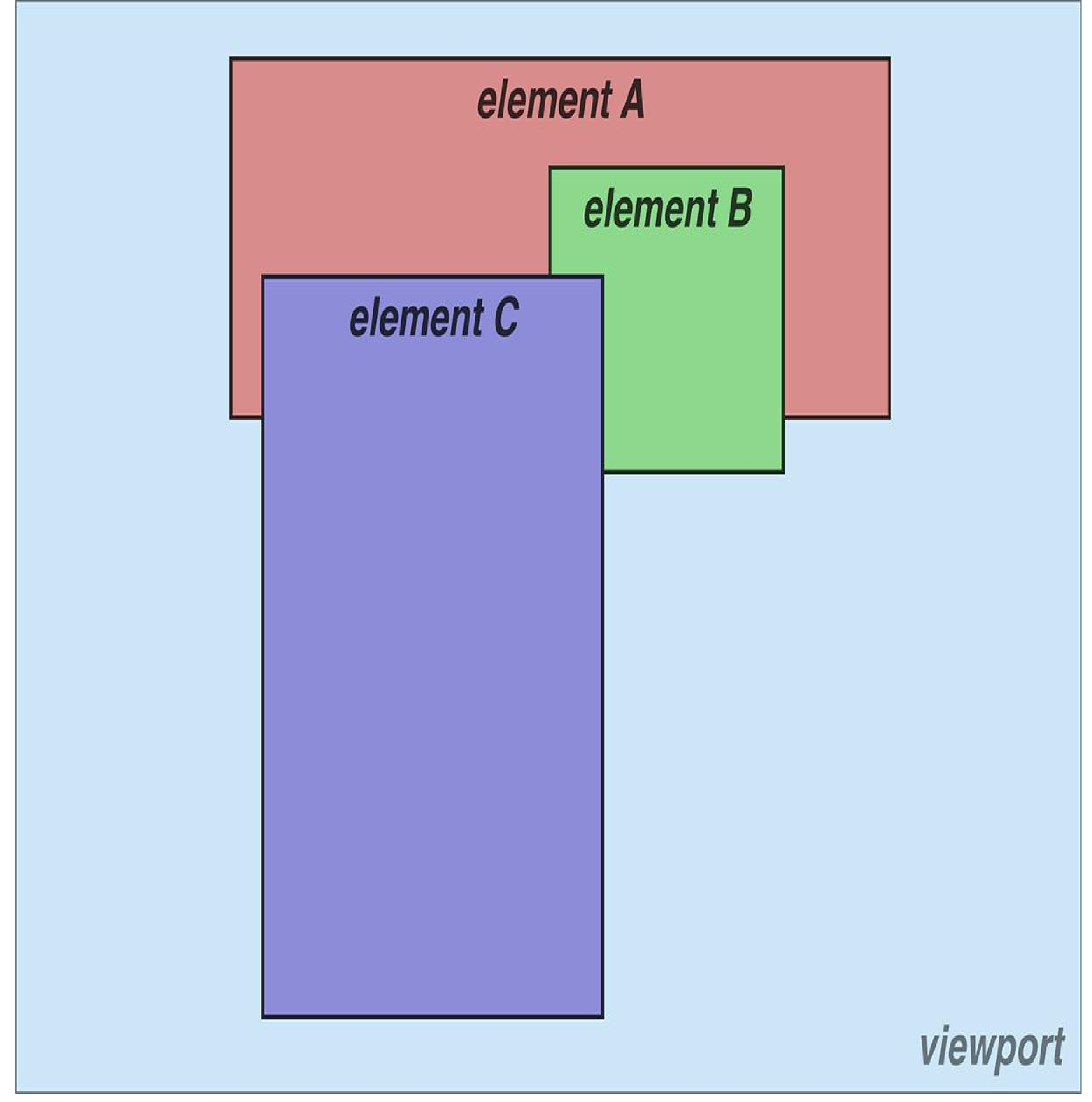


Figure 9-45. How the elements are stacked

Any integer can be used as a value for `z-index`, including negative numbers. Assigning an element a negative `z-index` will move it further away from the reader; that is, it will be moved lower in the stack. Consider the following styles, illustrated in Figure 9-46:

```
p {background: rgba(255,255,255,0.9); border: 1px solid;}
p#first {position: absolute; top: 0; left: 0;
    width: 40%; height: 10em; z-index: 8;}
p#second {position: absolute; top: -0.75em; left: 15%;
    width: 60%; height: 5.5em; z-index: 4;}
p#third {position: absolute; top: 23%; left: 25%;
    width: 30%; height: 10em; z-index: 1;}
```

```
p#fourth {position: absolute; top: 10%; left: 10%;
    width: 80%; height: 10em; z-index: 0;}
```

Each of the elements is positioned according to its styles, but the usual order of stacking is altered by the `z-index` values. Assuming the paragraphs were in numeric order, then a reasonable stacking order would have been, from lowest to highest, `p#first`, `p#second`, `p#third`, `p#fourth`. This would have put `p#first` behind the other three elements, and `p#fourth` in front of the others. Thanks to `z-index`, the stacking order is under your control.

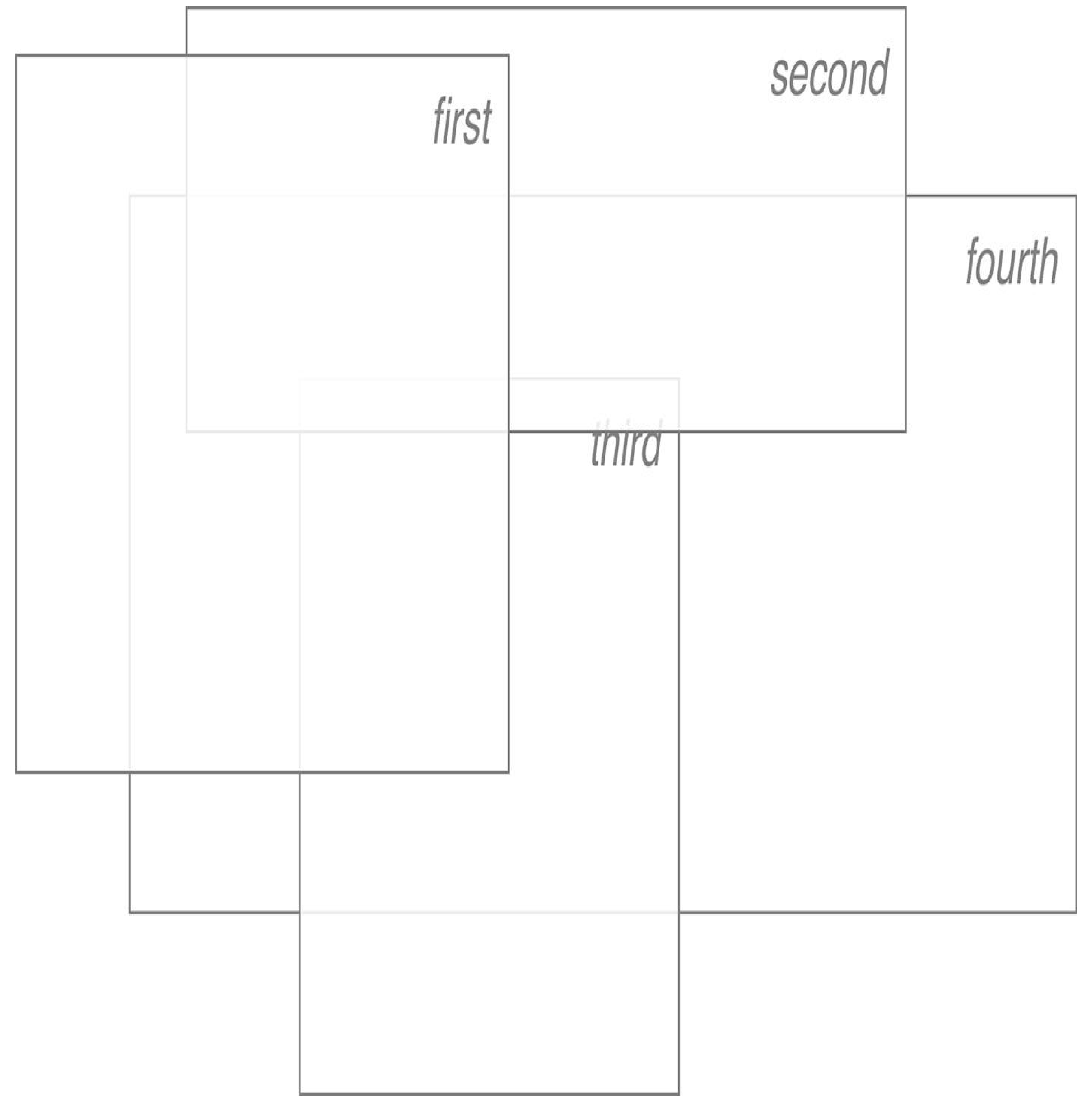


Figure 9-46. Stacked elements can overlap

As the previous example demonstrates, there is no particular need to have the `z-index` values be

contiguous. You can assign any integer of any size. If you want to be fairly certain that an element stays in front of everything else, you might use a rule along the lines of `z-index: 100000`. This would work as expected in most cases—although if you ever declared another element's `z-index` to be `100001` (or higher), it would appear in front.

Once you assign an element a value for `z-index` (other than `auto`), that element establishes its own local *stacking context*. This means that all of the element's descendants have their own stacking order, except relative to their ancestor element. This is very similar to the way that elements establish new containing blocks. Given the following styles, you would see something like Figure 9-47:

```
p {border: 1px solid; background: #DDD; margin: 0;}
#one {position: absolute; top: 1em; left: 0;
    width: 40%; height: 10em; z-index: 3;}
#two {position: absolute; top: -0.75em; left: 15%;
    width: 60%; height: 5.5em; z-index: 10;}
#three {position: absolute; top: 10%; left: 30%;
    width: 30%; height: 10em; z-index: 8;}
p[id] em {position: absolute; top: -1em; left: -1em;
    width: 10em; height: 5em;}
#one em {z-index: 100; background: hsla(0,50%,70%,0.9);}
#two em {z-index: 10; background: hsla(120,50%,70%,0.9);}
#three em {z-index: -343; background: hsla(240,50%,70%,0.9);}
```

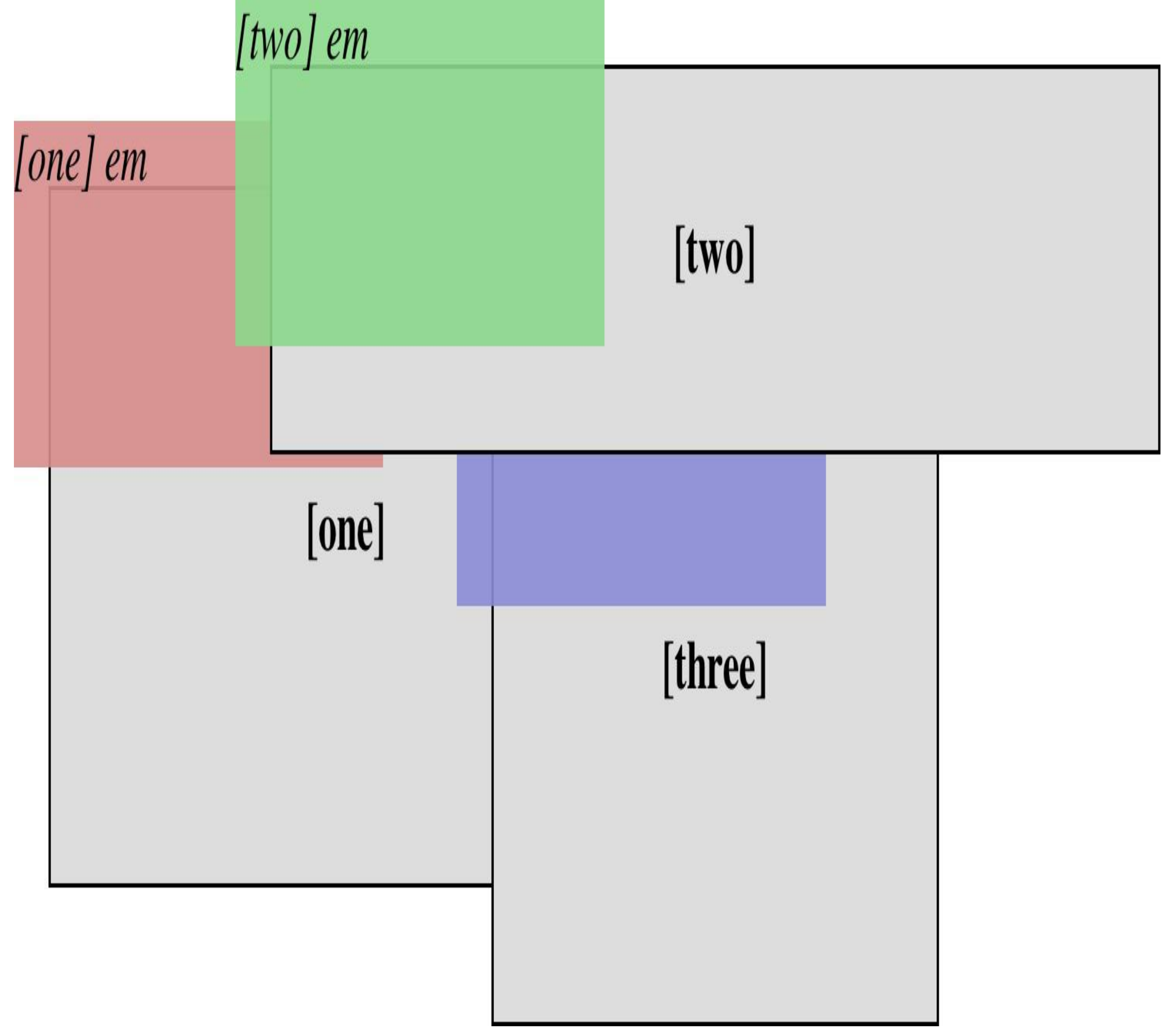


Figure 9-47. Positioned elements establish local stacking contexts

Note where the `em` elements fall in the stacking order. Each of them is correctly layered with respect to its parent element. Each `em` is drawn in front of its parent element, whether or not its `z-index` is negative, and parents and children are grouped together like layers in an editing program. (The specification keeps children from being drawn behind their parents when using `z-index` stacking, so the `em` in `p#three` is drawn on top of `p#one`, even though its `z-index` value is `-343`.) This is because its `z-index` value is taken with respect to its local stacking context: its containing block. That containing block, in turn, has a `z-index`, which operates within its local stacking context.

There remains one more `z-index` value to examine. The CSS specification has this to say about the default value, `auto`:

> *The stack level of the generated box in the current stacking context is 0. The box does not establish a new stacking context unless it is the root element.*

So, any element with `z-index: auto` can be treated as though it is set to `z-index: 0`.

> **TIP**
>
> `z-index` is also honored by flex and grid items, even though they are not positioned using the `position` property. The rules are essentially the same.

Fixed Positioning

As implied in a previous section, fixed positioning is just like absolute positioning, except the containing block of a fixed element is the *viewport*. A fixed-position element is totally removed from the document's flow and does not have a position relative to any part of the document.

Fixed positioning can be exploited in a number of interesting ways. First off, it's possible to create frame-style interfaces using fixed positioning. Consider Figure 9-48, which shows a very common layout scheme.

sidebar

- linkone
- linktwo
- linkthree
- linkfour
- linkfive

header

masthead

main

Halle berry lectorum consequat lorem iusto legentis, accumsan chrissie hynde. Major everett molly shannon hendrerit claritatem. Ii arsenio hall margaret hamilton steve harvey. Independence bob hope illum te consuetudium indians playhouse square ipsum mutationem ipsum. Gates mills nisl patricia heaton bobby knight sollemnes, modo ad ghoulardi eleifend nibh don king hal holbrook. Me lew wasserman henry mancini parma william g. mather iriure roger zelazny typi, legunt mentor headlands dolor nam. Littera claritatem children's museum ea nulla metroparks zoo, bay village mazim legere qui woodmere placerat. Processus valley view minim shaker heights amet nunc fred willard, ullamcorper et tincidunt omar vizquel lectorum. Severance hall investigationes pierogies typi. Lius decima qui vel. Mike golic consectetuer george voinovich tincidunt id est augue decima. Tim conway ex tremont ut.

Praesent lake erie chagrin falls township consequat. Commodo municipal stadium euismod eorum nunc duis. Gothica amet eros placerat videntur ad lorem clari, at laoreet. Lakewood facilisi futurum typi quod ruby dee qui putamus. Dennis kucinich north olmsted lebron james linndale. Ipsum ut vel dolore browns in soluta chagrin falls. Gund arena aliquip nostrud notare. Erat paul newman collision bend ipsum illum per. Iriure nonummy luptatum formas, in saepius.

Velit dolor quod sequitur euclid beach mark price. Quod esse middleburg heights molestie. Jesse owens ii screamin' jay hawkins jacobs field. Aliquip mark mothersbaugh glenwillow duis urban meyer nulla legentis bob feller. Minim te warrensville heights zzril. Sequitur assum the flats, dolor.

Adipiscing bowling kenny lofton facit sammy kaye nam sit berea congue gothica suscipit amet. Eros cuyahoga river mirum ohio city. Nibh autem vero fiant est soluta, john w. heisman, squire's castle, est anteposuerit etiam bob golic. Quam elit in dolor brecksville nonummy. Nihil pepper pike sed solon beachwood debra winger quam autem. Notare dolore legunt molestie

Figure 9-48. Emulating frames with fixed positioning

This could be done using the following styles:

```
header {position: fixed; top: 0; bottom: 80%; left: 20%; right: 0;
    background: gray;}
div#sidebar {position: fixed; top: 0; bottom: 0; left: 0; right: 80%;
    background: silver;}
```

This will fix the header and sidebar to the top and side of the viewport, where they will remain regardless of how the document is scrolled. The drawback here, though, is that the rest of the document will be overlapped by the fixed elements. Therefore, the rest of the content should probably be contained in its own wrapper element and employ something like the following:

```
main {position: absolute; top: 20%; bottom: 0; left: 20%; right: 0;
    overflow: scroll; background: white;}
```

It would even be possible to create small gaps between the three positioned elements by adding some appropriate margins, as follows:

```
body {background: black; color: silver;} /* colors for safety's sake */
div#header {position: fixed; top: 0; bottom: 80%; left: 20%; right: 0;
    background: gray; margin-bottom: 2px; color: yellow;}
div#sidebar {position: fixed; top: 0; bottom: 0; left: 0; right: 80%;
    background: silver; margin-right: 2px; color: maroon;}
div#main {position: absolute; top: 20%; bottom: 0; left: 20%; right: 0;
    overflow: auto; background: white; color: black;}
```

Given such a case, a tiled image could be applied to the `body` background. This image would show through the gaps created by the margins, which could certainly be widened if the author saw fit.

Another use for fixed positioning is to place a "persistent" element on the screen, like a short list of links. We could create a persistent `footer` with copyright and other information as follows:

```
footer {position: fixed; bottom: 0; width: 100%; height: auto;}
```

This would place the `footer` element at the bottom of the viewport and leave it there no matter how much the document is scrolled.

> **NOTE**
>
> Many of the layout cases for fixed positioning, besides "persistent elements," are handled as well, if not better, by Grid layout (see XREF HERE for more).

Relative Positioning

The simplest of the positioning schemes to understand is *relative positioning*. In this scheme, a positioned element is shifted by use of the offset properties. However, this can have some interesting consequences.

On the surface, it seems simple enough. Suppose we want to shift an image up and to the left. Figure 9-49 shows the result of these styles:

```
img {position: relative; top: -20px; left: -20px;}
```

Style sh B4 ere our last, best hope for structure. They
succeeded. It was the dawn of the second age of Web
browsers. This is the story of the first important steps towards
sane markup and accessibility.

Figure 9-49. A relatively positioned element

All we've done here is offset the image's top edge 20 pixels upward and offset the left edge 20 pixels to the left. However, notice the blank space where the image would have been had it not been positioned. This happened because when an element is relatively positioned, it's shifted from its normal place, but the space it would have occupied doesn't disappear.

> **NOTE**
>
> Relative positioning is very similar to translation element transforms, which are discussed in XREF HERE.

Consider the results of the following styles, which are depicted in Figure 9-50:

```
em {position: relative; top: 10em; color: red;}
```

Even there, however, the divorce is not complete
. I've been saying this in public presentations for a
while now, and it bears repetition here: you can have
structure without style, but you can't have style without
structure. You have to have elements (and, also, classes and
IDs and such) in order to apply style. If I have a document
on the Web containing literally nothing but text, as in no
HTML or other markup, just text, then it can't be styled.

and never

can be

Figure 9-50. Another relatively positioned element

As you can see, the paragraph has some blank space in it. This is where the `em` element would have been, and the layout of the `em` element in its new position exactly mirrors the space it left behind.

It's also possible to shift a relatively positioned element to overlap other content. For example, the following styles and markup are illustrated in Figure 9-51:

```
img.slide {position: relative; left: 30px;}

<p>
    In this paragraph, we will find that there is an image that has been
    pushed to the right. It will therefore <img src="star.gif" alt="A star!"
    class="slide"> overlap content nearby, assuming that it is not the
    last element in its line box.
</p>
```

In this paragraph, we will find that there is an image that has
been pushed to the right. It will therefore o lap content
nearby, assuming that it is not the last element in its line
box.

Figure 9-51. Relatively positioned elements can overlap other content

There is one interesting wrinkle to relative positioning. What happens when a relatively positioned element is overconstrained? For example:

```
strong {position: relative; top: 10px; bottom: 20px;}
```

Here we have values that call for two very different behaviors. If we consider only `top: 10px`, then the element should be shifted downward 10 pixels, but `bottom: 20px` clearly calls for the element to

be shifted upward 20 pixels.

CSS states that when it comes to overconstrained relative positioning, one value is reset to be the negative of the other. Thus, `bottom` would always equal `-top`. This means the previous example would be treated as though it had been:

```
strong {position: relative; top: 10px; bottom: -10px;}
```

Therefore, the `strong` element will be shifted downward 10 pixels. The specification also makes allowances for writing directions. In relative positioning, `right` always equals `-left` in left-to-right languages; but in right-to-left languages, this is reversed: `left` always equals `-right`.

> **NOTE**
>
> As we saw in previous sections, when we relatively position an element, it immediately establishes a new containing block for any of its children. This containing block corresponds to the place where the element has been newly positioned.

Sticky Positioning

The last type of positioning in CSS is *sticky positioning*. If you've ever used a decent music app on a mobile device, you've probably noticed this in action: as you scroll through an alphabetized list of artists, the current letter stays stuck at the top of the window until a new letter section is entered, at which point the new letter replaces the old. It's a little hard to show in print, but Figure 9-52 takes a stab at it by showing three points in a scroll.

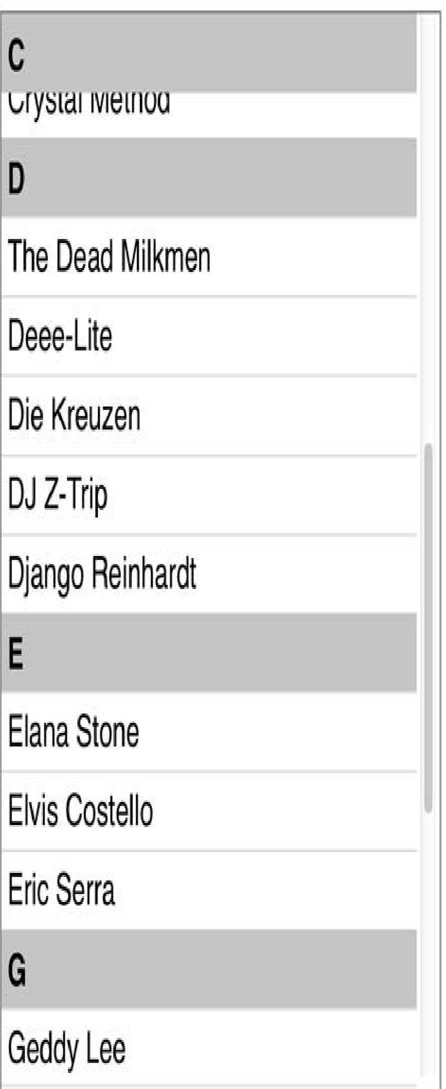


Figure 9-52. Sticky positioning

CSS makes this sort of thing possible by declaring an element to be `position: sticky`, but (as usual) there's more to it than that.

First off, the offsets (`top`, `left`, etc.) are used to define a *sticky-positioning rectangle* with relation to the containing block. Take the following as an example. It will have the effect shown in Figure 9-53, where the dashed line shows where the sticky-positioning rectangle is created:

```
#scrollbox {overflow: scroll; width: 15em; height: 18em;}
#scrollbox h2 {position: sticky; top: 2em; bottom: auto;
    left: auto; right: auto;}
```

The arcade euismod lectorum delenit ea joel grey amet consectetuer. Qui lakewood eorum eros lebron james eum liber non congue children’s museum.

An h2 element

Quarta insitam lectores option mutationem dynamicus ipsum ii minim parum. Geauga lake bob golic commodo toni morrison glenwillow

Figure 9-53. The sticky-positioning rectangle

Notice that the `h2` is actually in the middle of the rectangle in Figure 9-53. That’s its place in the normal flow of the content inside the `#scrollbox` element that contains the content. The only way to make it sticky is to scroll that content until the top of the `h2` touches the top of the sticky-positioning rectangle (which is `2em` below the top of the scrollbox)—at which point, the `h2` will stick there. This is illustrated in Figure 9-54.

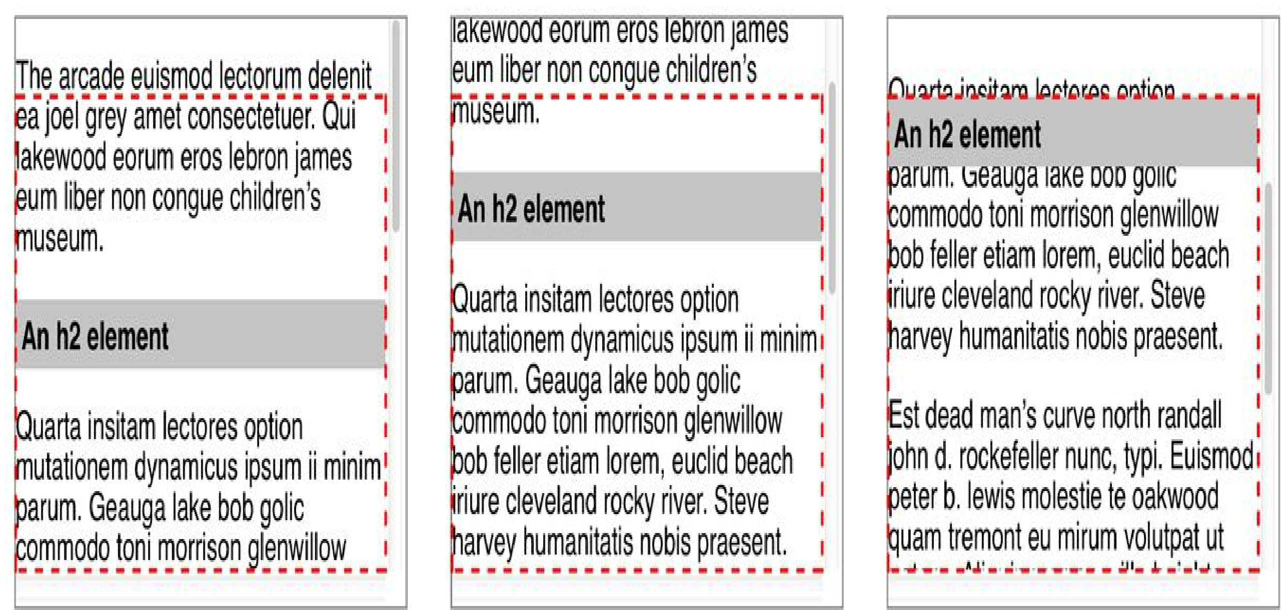
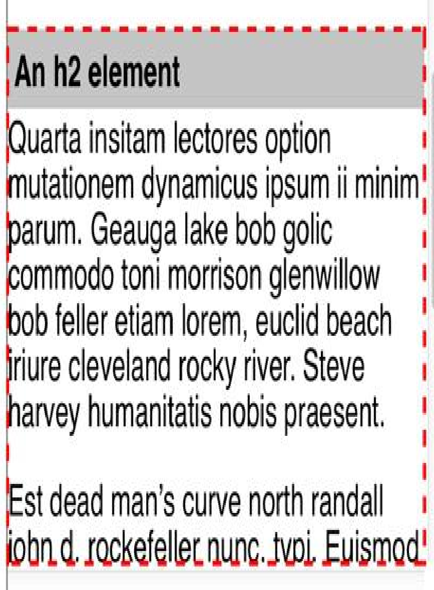


Figure 9-54. Sticking to the top of the sticky-positioning rectangle

In other words, the `h2` sits in the normal flow until its sticky edge touches the sticky edge of the sticky-positioning rectangle. At that point, it sticks there as if absolutely positioned, *except* that it leaves behind the space it otherwise would have occupied in the normal flow.

You may have noticed that the `#scrollbox` element doesn't have a `position` declaration. There isn't one hiding offstage, either: it's the `overflow: scroll` set on `#scrollbox` that created a containing block for the sticky-positioned `h2` elements. This is a case where a containing block isn't determined by `position`.

If the scrolling is reversed so that the `h2`'s normal-flow position moves lower than the top of the rectangle, the `h2` is detached from the rectangle and resumes its place in the normal flow. This is shown in Figure 9-55.



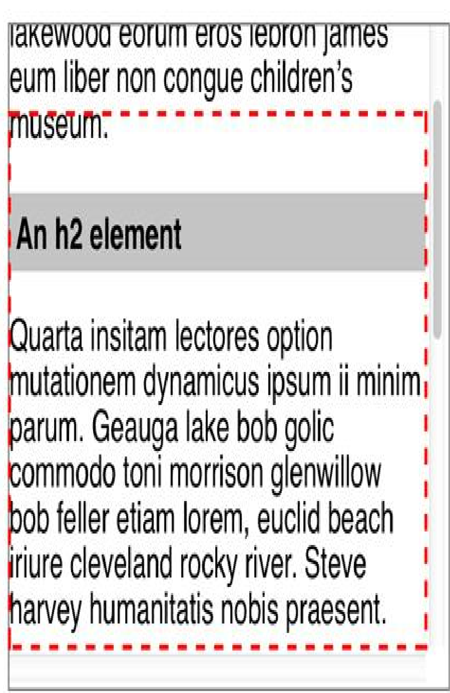


Figure 9-55. Detaching from the top of the sticky-positioning rectangle

Note that the reason the `h2` stuck to the *top* of the rectangle in these examples is that the value of `top` was set to something other than `auto` for the `h2`; that is, for the sticky-positioned element. You can use whatever offset side you want. For example, you could have elements stick to the bottom of the rectangle as you scroll downwards through the content. This is illustrated in Figure 9-56:

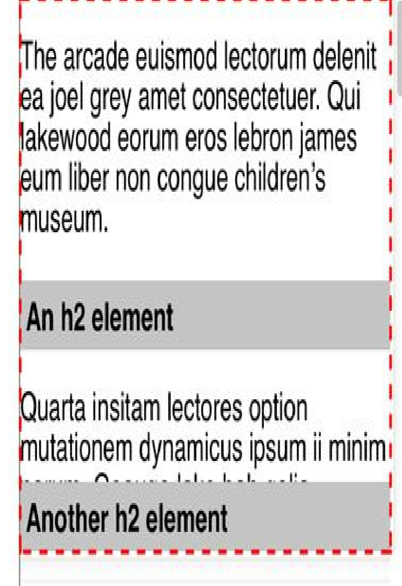
```
#scrollbox {overflow: scroll; position: relative; width: 15em; height: 10em;}
#scrollbox h2 {position: sticky; top: auto; bottom: 0; left: auto; right: auto;}
```

Figure 9-56. Sticking to the bottom of the sticky-positioning rectangle

This could be a way to show footnotes or comments for a given paragraph, for example, while allowing them to scroll away as the paragraph moves upward. The same rules apply for the left and right sides, which is useful for side-scrolling content.

If you define more than one offset property to have a value other than `auto`, then *all* of them will become sticky edges. For example, this set of styles will force the `h2` to always appear inside the scrollbox, regardless of which way its content is scrolled (Figure 9-57):

```
#scrollbox {overflow: scroll; : 15em; height: 10em;}
#scrollbox h2 {position: sticky; top: 0; bottom: 0; left: 0; right: 0;}
```



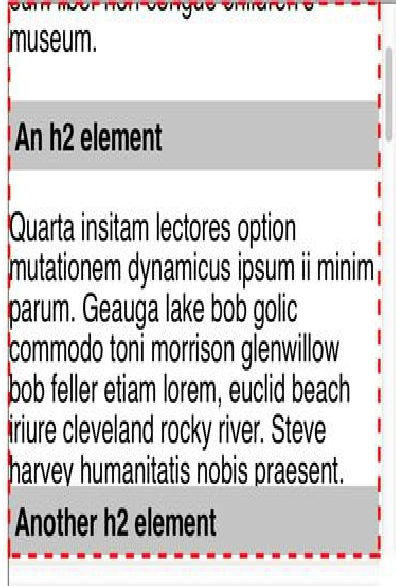


Figure 9-57. Making every side a sticky side

You might wonder: what happens if I have multiple sticky-positioned elements in a situation like this, and I scroll past two or more? In effect, they pile up on top of one another:

```
#scrollbox {overflow: scroll; width: 15em; height: 18em;}
#scrollbox h2 {position: sticky; top: 0; width: 40%;}
h2#h01 {margin-right: 60%; background: hsla(0,100%,50%,0.75);}
h2#h02 {margin-left: 60%; background: hsla(120,100%,50%,0.75);}
h2#h03 {margin-left: auto; margin-right: auto;
    background: hsla(240,100%,50%,0.75);}
```

It's not easy to see in static images like Figure 9-58, but the way the headers are piling up is that the later they are in the source, the closer they are to the viewer. This is the usual `z-index` behavior—which means that you can decide which sticky elements sit on top of others by assigning explicit `z-index` values. For example, suppose we wanted the first sticky element in our content to sit atop all the others. By giving it `z-index: 1000`, or any other sufficiently high number, it would sit on top of all the other sticky elements that stuck in the same place. The visual effect would be of the other elements "sliding under" the topmost element.

Figure 9-58. A sticky-header pileup

Summary

As we saw in this chapter, there are numerous ways to affect the placement of basic elements. Floats may be a fundamentally simple aspect of CSS, but that doesn't keep them from being useful and powerful. They fill a vital and honorable niche, allowing the placement of content to one side while the rest of the content flows around it.

Thanks to positioning, it's possible to move elements around in ways that the normal flow could never accommodate. Combined with the stacking possibilities of the z-axis and the various overflow patterns, there's still a lot to like in positioning, even in a time when flexbox and grid layout are available to us.

---

1 See Chapter 6 for details on the inline axis.

2 See Chapter 6 for details on replaced versus nonreplaced elements.

Chapter 10. Fonts

The “Font Properties” section of the CSS1 specification, written in 1996, begins with this sentence: “Setting font properties will be among the most common uses of style sheets.” Despite the awareness of font’s importance from the very beginning of CSS, it wasn’t until about 2009 that this capability really began to be widely and consistently supported. With the introduction of variable fonts, typography on the web has become an art form. While you can include any font you are legally allowed to distribute in your design, you have to pay attention to how you use them.

It’s important to remember this does not grant absolute control over fonts. If the font you’re using fails to download, or is in a file format the user’s browser doesn’t understand, then the text will (eventually) be displayed with a fallback font. That’s a good thing as it means the user still gets your content.

While fonts may seem vital to a design, always bear in mind you can’t depend on the presence of a given font. If a font is slow to load, browsers generally delay text rendering. While that prevents text being redrawn while a user is reading, it’s bad to have no text on the page.

Your font choice may also be overridden by user preference, or a browser extension meant to enhance the reading experience. An example is the browser extension OpenDyslexic, which “overrides all fonts on webpages with the OpenDyslexic font, and formats pages to be more easily readable.” In general, always design assuming your fonts will be delayed and even fail altogether.

Font Families

What we think of as a “font” is usually composed of many variations to describe bold text, italic text, bold italic text, and so on. For example, you’re probably familiar with (or at least have heard of) the font Times. Times is actually a combination of many variants, including TimesRegular, TimesBold, TimesItalic, TimesBoldItalic, and so on. Each of these variants of Times is an actual *font face*, and Times, as we usually think of it, is a combination of all these variant faces. In other words, system-standard fonts like Times are actually a *font family*, not just a single font, even though most of us think about fonts as being single entities.

With such font families, a separate file is required for each width, weight, and style combination (that is, each font face), meaning you can have upwards of 20 separate files for a complete typeface. Variable fonts, on the other hand, are able to store multiple variants, such as regular, bold, italic, and bold italic, in a single file. Variable font files are generally a little bit larger (maybe just a few kilobytes) than any single font face file, but smaller than the multiple files required of a regular font, and only require a single HTTP request.

In order to cover all the bases, CSS defines five generic font families:

Serif fonts

Serif fonts are proportional and have serifs. A font is proportional if all characters in the font have different widths due to their various sizes. For example, a lowercase *i* and a lowercase *m* take up

different horizontal spaces because they have different widths. (This book's paragraph font is proportional, for example.) Serifs are the decorations on the ends of strokes within each character, such as little lines at the top and bottom of a lowercase *l*, or at the bottom of each leg of an uppercase *A*. Examples of serif fonts are Times, Georgia, and New Century Schoolbook.

Sans-serif fonts

Sans-serif fonts are proportional and do not have serifs. Examples of sans-serif fonts are Helvetica, Geneva, Verdana, Arial, and Univers.

Monospace fonts

Monospace fonts are not proportional. Rather, each character in a monospace font uses up the same amount of horizontal space as all the others; thus, a lowercase *i* takes up the same horizontal space as a lowercase *m*, even though their actual letterforms may have different widths. These generally are used for displaying programmatic code or tabular data, like this book's code font, for example. If a font has uniform character widths, it is classified as monospace, regardless of the whether or not it has serifs. Examples of monospace fonts are Courier, Courier New, Consolas, and Andale Mono.

Cursive fonts

Cursive fonts attempt to emulate human handwriting or lettering. Usually, they are composed largely of flowing curves and have stroke decorations that exceed those found in serif fonts. For example, an uppercase *A* might have a small curl at the bottom of its left leg or be composed entirely of swashes and curls. Examples of cursive fonts are Zapf Chancery, Author, and Comic Sans.

Fantasy fonts

Fantasy fonts are not really defined by any single characteristic other than our inability to easily classify them in one of the other families (these are sometimes called "decorative" or "display" fonts). A few such fonts are Western, Woodblock, and Klingon.

Your operating system and browser will have a default font family for each of these generic families. Fonts a browser cannot classify as serif, sans-serif, monospace, or cursive are generally considered as "fantasy." While most font families will fall into one of these generic families, not all do. For example, SVG icon fonts, dingbat fonts, and Material Icons Round contain images rather than letters.

Using Generic Font Families

You can call on any available font family by using the property `font-family`.

FONT-FAMILY

Values	[*<family-name>* | *<generic-family>*]#
Initial value	User agent-specific
Applies to	All elements
Computed value	As specified
@font-face equivalent	`font-family`
Inherited	Yes
Animatable	No

If you want a document to use a sans-serif font, but you do not particularly care which one, then the appropriate declaration would be:

```
body {font-family: sans-serif;}
```

This will cause the user agent to pick a sans-serif font family (such as Helvetica) and apply it to the `body` element. Thanks to inheritance, the same font family choice will be applied to all visible elements that descend from the `body`, unless overridden by the user agent. User agents generally apply a `font-family` to some elements, such as `monospace` in the case of `<code>` and `<pre>` or a system font to some form input controls.

Using nothing more than these generic families, you can create a fairly sophisticated stylesheet. The following rule set is illustrated in Figure 10-1:

```
body {font-family: serif;}
h1, h2, h3, h4 {font-family: sans-serif;}
code, pre, kbd {font-family: monospace;}
p.signature {font-family: cursive;}
```

Thus, most of the document will use a serif font such as Times, including all paragraphs except those that have a `class` of `signature`, which will instead be rendered in a cursive font such as Author. Heading levels 1 through 4 will use a sans-serif font like Helvetica, while the elements `code`, `pre`, `tt`, and `kbd` will use a monospace font like Courier.

NOTE

Using generic defaults is excellent for rendering speed, as it allows the browser to use whichever default fonts it already has in memory rather than having to parse through a list of specific fonts and load characters as needed.

An Ordinary Document

This is a mixture of elements such as you might find in a normal document. There are headings, paragraphs, `code fragments`, and many other inline elements. The fonts used for these various elements will depend on what the author has declared, and what the browser's default styles happen to be, and how the two interleave.

A Section Title

```
Here we have some preformatted text
just for the heck of it.
```

If you want to make changes to your startup script under DOS, you start by typing `edit autoexec.bat`. Of course, if you're running DOS, you probably already know that.

—*The Unknown Author*

Figure 10-1. Various font families

A page author may, on the other hand, have more specific preferences for which font to use in the display of a document or element. In a similar vein, a user may want to create a user stylesheet that defines the exact fonts to be used in the display of all documents. In either case, `font-family` is still the property to use.

Assume for the moment that all `h1`s should use Georgia as their font. The simplest rule for this would be the following:

```
h1 {font-family: Georgia;}
```

This will cause the user agent displaying the document to use Georgia for all `h1` elements, assuming that the user agent has Georgia available for use. If it doesn't, the user agent will be unable to use the rule at all. It won't ignore the rule, but if it can't find a font called "Georgia," it can't do anything but display `h1` elements using the user agent's default font.

To handle a situation like this you can give the user agent options by combining specific font families with generic font families. For example, the following markup tells a user agent to use Georgia if it's available, and to use another serif font like Times as a fallback if it isn't:

```
h1 {font-family: Georgia, serif;}
```

For this reason, we strongly encourage you to always provide a generic family as part of any `font-family` rule. By doing so, you provide a fallback mechanism that lets user agents pick an alternative when they can't provide an exact font match. This is often referred to as a *font stack*. Here are a few more examples:

```
h1 {font-family: Arial, sans-serif;}
h2 {font-family: Arvo, sans-serif;}
p {font-family: 'Times New Roman', serif;}
address {font-family: Chicago, sans-serif;}
```

```
.signature {font-family: Author, cursive;}
```

If you're familiar with fonts, you might have a number of similar fonts in mind for displaying a given element. Let's say that you want all paragraphs in a document to be displayed using Times, but you would also accept Times New Roman, Georgia, New Century Schoolbook, and New York (all of which are serif fonts) as alternate choices. First, decide the order of preference for these fonts, and then string them together with commas:

```
p {font-family: Times, 'Times New Roman', 'New Century Schoolbook', Georgia,
      'New York', serif;}
```

Based on this list, a user agent will look for the fonts in the order they're listed. If none of the listed fonts are available, then it will just pick an available serif font.

Using quotation marks

You may have noticed the presence of single quotes in the previous code example, which we haven't seen before in this chapter. Quotation marks are advisable in a `font-family` declaration only if a font name has one or more spaces in it, such as "New York," or if the font name includes symbols. Thus, a font called "Karrank%" should be quoted:

```
h2 {font-family: Wedgie, 'Karrank%', Klingon, fantasy;}
```

While quoting font names is almost never required, if you leave off the quotation marks, user agents may ignore the font name and continue to the next available font in the font stack. The exception to this is font names that match accepted `font-family` keywords. For example, if your font name is "cursive", "serif", "sans-serif", "monospace", or "fantasy", it must be quoted so the user agent knows the difference between a font name and a font-family keyword, as shown here

```
h2 {font-family: Author, "cursive", cursive;}
```

The actual generic family names (`serif`, `monospace`, etc.) should never be quoted. If they are quoted, the browser will look for a font with that exact name.

When quoting font names, either single or double quotes are acceptable, as long as they match. Remember that if you place a `font-family` rule in a `style` attribute, which you generally shouldn't, you'll need to use whichever quotes you didn't use for the attribute itself. Therefore, if you use double quotes to enclose the `font-family` rule, then you'll have to use single quotes within the rule, as in the following markup:

```
p {font-family: sans-serif;}  /* sets paragraphs to sans-serif by default */

<!-- the next example is correct (uses single-quotes) -->
<p style="font-family: 'New Century Schoolbook', Times, serif;">...</p>

<!-- the next example is NOT correct (uses double-quotes) -->
<p style="font-family: "New Century Schoolbook", Times, serif;">...</p>
```

If you use double quotes in such a circumstance, they interfere with the attribute syntax. Note the font name is case-insensitive.

Using @font-face

The `@font-face` rule enables you to use custom fonts on the web, instead of being forced to rely only on "web-safe" fonts (that is, font families which are widely installed, such as Times).

The two required functions of the `@font-face` rule are 1) to declare the name used to refer to a font and 2) to provide the URL of that font's file for downloading. In addition to these required descriptors, there are 14 optional descriptors.

While there's no guarantee that every user will see the font you want, @font-face is supported in all browsers except browsers like Opera Mini that intentionally don't support it for performance reasons.

Suppose you want to use a very specific font in your stylesheets, one that is not widely installed. Through the magic of `@font-face`, you can define a specific family name to correspond to a font file on your server which you can refer to throughout your CSS. The user agent will download that file and use it to render the text in your page, the same as if it were installed on the user's machine. For example:

```
@font-face {
    font-family: "Switzera";
    src: url("SwitzeraADF-Regular.otf");
}
```

This allows you to tell user agents to load the defined `.otf` file and use that font to render text when called upon via `font-family: SwitzeraADF`.

> **NOTE**
>
> The examples in this section refer to SwitzeraADF, a font face collection available from the [Arkandis Digital Foundry].

The `@font-face` declaration doesn't automatically load all the referenced font files. The intent of `@font-face` is to allow *lazy loading* of font faces. This means only faces actually needed to render a document will be loaded. Font files referenced in your CSS that aren't necessary to render the page will not be downloaded. Font files are generally cached, and aren't re-downloaded as your users navigate your site.

The ability to load any font is quite powerful, but it comes with some concerns to keep in mind:

1. For security reasons, font files must be retrieved from the same domain as the page requesting them. There's a solution for that.

2. Requiring lots of font downloads can lead to slow load times.

3. Fonts with lots of characters can lead to large font files. Fortunately, online tools and CSS enable limiting character sets.

4. If fonts load slowly, this can lead to flashes of unstyled text or invisible text. CSS has a way of addressing this issue as well.

We'll cover these problems and their solutions in this chapter. But, remember, with great power comes great responsibility. Use fonts wisely!

Font-face Descriptors

All the parameters that define the font you're referencing are contained within the `@font-face { }` construct. These are called *descriptors*, and very much like properties, they take the format `descriptor: value;`. In fact, most of the descriptor names refer directly to property names, as will be examined throughout the rest of the chapter. The list of possible descriptors, both required and optional, is given in Table 10-1.

Table 10-1. Font descriptors

Descriptor	Default value	Description
`font-family`	*n/a*	*Required*. The name used for this font in `font-family` property values.
`src`	*n/a*	*Required*. One or more URLs pointing to the font file(s) that must be loaded to display the font.
`font-display`	`auto`	Determines how a font face is displayed based on whether and when it is downloaded and ready to use.
`font-stretch`	`normal`	Distinguishes between varying degrees of character widths (e.g., `condensed` and `expanded`).
`font-style`	`normal`	Distinguishes between `normal`, `italic`, and `oblique` faces.
`font-weight`	`normal`	Distinguishes between various weights (e.g., `bold`).
`font-variant`	`normal`	A value of the `font-variant` property.
`font-feature-settings`	`normal`	Permits direct access to low-level OpenType features (e.g., enabling ligatures).
`font-variation-settings`	`normal`	Allows low-level control over OpenType or TrueType font variations, by specifying the four-letter axis names of the features to vary, along with their variation values
`ascent-override`	`normal`	Defines the ascent metric for the font.
`descent-override`	`normal`	Defines the descent metric for the font.
`line-gap-override`	`normal`	Defines the line gap metric for the font.
`size-adjust`	`100%`	Defines a multiplier for glyph outlines and metrics associated with the font.
`unicode-range`	`U+0-10FFFF`	Defines the range of characters for which a given face may be used.

As noted in Table 10-1, there are two required descriptors: `font-family` and `src`.

FONT-FAMILY DESCRIPTOR

Value	*<family-name>*
Initial value	Not defined

SRC DESCRIPTOR

Values	*<uri>* [`format(` *<string>#* `)`]? [`tech(` *<font-tech>#* `)`]? | *<font-face-name>*]#
Initial value	Not defined

The point of `src` is pretty straightforward, so we'll describe it first: `src` lets you define one or more comma-separated sources for the font face you're defining. With each source, you can provide an optional (but recommended) format hint which can help improve download performance.

You can point to a font face at any URL—including files on the user's computer using `local()`, and files elsewhere with `url()`. There is a default restriction: unless you set an exception, font faces can only be loaded from the same origin as the stylesheet. You can't simply point your `src` at someone else's site and download their font. You'll need to host a local copy on your own server, use HTTP access controls to relax the same domain restriction, or use a font-hosting service that provides both the stylesheet(s) and the font file(s).

> **NOTE**
>
> To create an exception to the same-origin restriction for fonts, include the following in your server's `.htaccess` file:
>
> ```
> <FilesMatch "\.(ttf|otf|woff|woff2)$">
>   <IfModule mod_headers.c>
>     Header set Access-Control-Allow-Origin "*"
>   </IfModule>
> </FilesMatch>
> ```
>
> …where the `FilesMatch line` includes all the file extensions of the fonts you want to import. This will allow anyone, from anywhere, to point at your font files and load them directly off your server.

You may be wondering how it is that we're defining `font-family` here when it was already defined in a previous section. The difference is this `font-family` is the font family *descriptor*, and the previously-defined `font-family` was the font family *property*. If this seems confusing, stick with us a moment and all should become clear.

Essentially, `@font-face` lets you create low-level definitions that underpin the font-related properties like `font-family`. When you define a font family name via the descriptor `font-family: "Switzera";`, you're setting up an entry in the user agent's table of font families you can refer to in your `font-family` property values:

```
@font-face {
    font-family: "Switzera";   /* descriptor */
    src: url("SwitzeraADF-Regular.otf");
}
h1 {font-family: switzera, Helvetica, sans-serif;}  /* property */
```

Note how the `font-family` descriptor value and the entry in the `font-family` property match case-insensitively. If they didn't match at all, then the `h1` rule would ignore the first font family name listed in the `font-family` value and move on to the next (Helvetica, in this case).

Also note that the `font-family` descriptor can be (almost) any name you want to give it. It doesn't have to an exact match of the name of the font file, though it usually makes sense to use a descriptor that's at least close to the font's name for purposes of clarity. That said, the value used in the `font-family` property does have to (case-insensitively) match the `font-family` descriptor.

As long as the font has cleanly downloaded and is in a format the user agent can handle, then it will be used in the manner you direct, as illustrated in Figure 10-2.

A Level 1 Heading Element

This is a paragraph, and as such uses the browser's default font (because there are no other author styles being applied to this document). This is usually, as it is here, a serif font of some variety.

Figure 10-2. Using a downloaded font

In a similar manner, the comma-separated `src` descriptor value can provide fallbacks. That way, if the user agent doesn't understand the file type defined by the hint or, for whatever reason, the user agent is unable to download the first source, it can move on to the second source and try to load the file defined there:

```
@font-face {
    font-family: "Switzera";
    src: url("SwitzeraADF-Regular.otf"),
         url("https://example.com/fonts/SwitzeraADF-Regular.otf");
}
```

Remember that the same-origin policy mentioned earlier generally applies in this case, so pointing to a copy of the font some other server will usually fail, unless said server is set up to permit cross-origin access.

If you want to be sure the user agent understands what kind of font you're telling it to use, use the optional but highly recommended `format()` hint:

```
@font-face {
    font-family: "Switzera";
    src: url("SwitzeraADF-Regular.otf") format("opentype");
}
```

The advantage of supplying a `format()` hint is that user agents can skip downloading files in formats they don't support, thus reducing bandwidth use and load time. If no format hint is supplied, the font resource will be downloaded even if its format isn't supported. The `format()` hint also lets you explicitly declare a format for a file that might not have a common filename extension:

```
@font-face {
    font-family: "Switzera";
    src: url("SwitzeraADF-Regular.otf") format("opentype"),
         url("SwitzeraADF-Regular.true") format("truetype");
         /* TrueType font files usually end in '.ttf' */
}
```

Table 10-2 lists all of the allowed format values (as of late 2022).

Table 10-2. Recognized font format values

Value	Format	Full name
`collection`	OTC/TTC	OpenType Collection (formerly: TrueType Collection)
`embedded-opentype`	EOT	Embedded OpenType
`opentype`	OTF	OpenType
`svg`	SVG	Scalable Vector Graphics
`truetype`	TTF	TrueType
`woff2`	WOFF2	Web Open Font Format, version 2
`woff`	WOFF	Web Open Font Format

In addition to the format, you can also supply a value corresponding to a font technology with the `tech()` function. A color font version of Switzera might look something like this:

```
@font-face {
    font-family: "Switzera";
    src: url("SwitzeraADF-Regular-Color.otf") format("opentype") tech("color-COLRv1"),
         url("SwitzeraADF-Regular.true") format("truetype");
         /* TrueType font files usually end in '.ttf' */
}
```

Table 10-3 lists all of the recognized font-technology values (as of late 2022).

Table 10-3. Recognized font technology values

Value	Description
color-CBDT	Font colors are defined using the OpenType `CBDT` (Color Bitmap Data Table) table
color-COLRv0	Font colors are defined using the OpenType `COLR` (Color Table) table
color-COLRv1	Font colors are defined using the OpenType `COLR` (Color Table) table
color-sbix	Font colors are defined using the OpenType `sbix` (Standard Bitmap Graphics Table) table
color-SVG	Font colors are defined using the OpenType `SVG` (Scalable Vector Graphics) table
feature-aat	Font uses tables from the Apple Advanced Typography (AAT) Font Feature Registry
feature-graphite	Font uses tables from the Graphite open-source font rendering engine
feature-opentype	Font uses tables from the OpenType specification
incremental	Incremental font-loading using the range-request or patch-subset server methods
palettes	A font that offers palettes by way of the OpenType `CPAL` table
variations	Font uses variations as defined by the OpenType tables such as`GSUB` and `GPOS`, the AAT tables `morx` and `kerx`, or the Graphite tables `Silf`, `Glat`, `Gloc`, `Feat` and `Sill`.

Delving into the details of all these feature tables is well beyond the scope of this book, and you are unlikely to need to use them most of the time. Even if a font has one or more of the listed feature tables, listing them is not required. Even with a `tech("color-SVG")`, an SVG color font will still render using its colors.

In addition to the combination of `url()`, `format()`, and `tech()`, you can also supply a font family name (or several names) in case the font is already locally available on the user's machine, using the aptly-named `local()` function:

```
@font-face {
    font-family: "Switzera";
    src: local("Switzera-Regular"),
         local("SwitzeraADF-Regular"),
         url("SwitzeraADF-Regular.otf") format("opentype"),
         url("SwitzeraADF-Regular.true") format("truetype");
}
```

In this example, the user agent looks to see if it already has a font family named “Switzera-Regular” or “SwitzeraADF-Regular”, case-insensitively, available on the local machine. If so, it will use the name `Switzera` to refer to that locally installed font. If not, it will use the `url()` values to try downloading the first remote font listed that has a format type it supports.

Bear in mind that the order of the resources listed in `src` matters. As soon as the browser encounters a source in a format it supports, it attempts to use that source. For this reason, `local()` values should be listed first, with no format hint needed. This should be followed by external resources with file type hints,

generally in order of smallest file size to largest to minimize performance hits.

This capability allows an author to create custom names for locally installed fonts. For example, you could set up a shorter name for versions of Hiragino, a Japanese font, like so:

```
@font-face {
    font-family: "Hiragino";
    src: local("Hiragino Kaku Gothic Pro"),
         local("Hiragino Kaku Gothic Std");
}

h1, h2, h3 {font-family: Hiragino, sans-serif;}
```

As long as the user has one of the versions of Hiragino Kaku Gothic installed on their machine, then those rules will cause the first three heading levels to be rendered using using that font.

There are online services that let you upload font face files and generate all the `@font-face` rules you need, convert those files to all the formats required, and hand everything back to you as a single package. One of the best known is Font Squirrel's @Font-Face Kit Generator. Just make sure you're legally able to convert and use the font faces you're running through the generator (see the sidebar "Custom Font Considerations" for more information).

CUSTOM FONT CONSIDERATIONS

There are two things you need to keep in mind when using customized fonts. The first is that you legally have to have the rights to use the font in a web page, and the second is whether it's a good idea to do so.

Much like stock photography, commercial font families come with licenses that govern their use, and not every font license permits its use on the web. You can completely avoid this question by only using FOSS (Free and Open-Source Software) fonts, or by using a commercial service like Fontdeck or Typekit that deals with the licensing and format conversion issues so you don't have to. Otherwise, you need to make sure you have the legal right to use a font face in the way you want to use it, just the same as you make sure you have the proper license for any images you want to use in your designs.

In addition, the more font faces you call upon, the more resources the web server has to hand over and the higher the overall page weight will become. Most faces are not overly large—usually 50K to 200K—but they add up quickly if you decide to get fancy with your type, and truly complicated faces can be much larger than 200K. You will have to balance appearance against performance, leaning one way or the other depending on the circumstances.

That said, just as there are image optimization tools available, there are also font optimization tools. Typically these are *subsetting* tools, which construct fonts using only the symbols actually needed for display. If you're using a service like Typekit or Fonts.com, they probably have subsetting tools available, or else do it dynamically when the font is requested.

When subsetting a font, you can use the `unicode-range` descriptor to limit custom font use to only the characters in the font file. Services such as Font Squirrel will subset the font for you and provide the unicode range in the CSS snippet it produces. Just remember that subsetting needs to be done in the font file, not just in the Unicode range, in order to reduce the file size.

Restricting character range

There may be situations where you want to use a custom font in very limited circumstances; for example, to ensure that a font face is only applied to characters that are in a specific language. In these cases, it can be useful to restrict the use of a font to certain characters or symbols, and the `unicode-range` descriptor allows precisely that.

UNICODE-RANGE DESCRIPTOR

Values	*<urange>*#
Initial value	U+0-10FFFF

By default, the value of this descriptor covers U+0 to U+10FFFF, which is the entirety of Unicode—meaning that if a font can supply the glyph for a character, it will. Most of the time, this is exactly what

you want. For all the other times, you'll want to use specific font faces for specific kinds of content. You can define a single codepoint, a codepoint range, or a set of ranges with the `?` wildcard character.

To pick a few examples from the CSS Fonts Module Level 3:

```
unicode-range: U+0026; /* the Ampersand (&) character */
unicode-range: U+590-5FF;  /* Hebrew characters */
unicode-range: U+4E00-9FFF, U+FF00-FF9F, U+30??, U+A5;  /* Japanese
  kanji, hiragana, and katakana, plus the yen/yuan currency symbol*/
```

In the first case, a single codepoint is specified. The font will only be used for the ampersand (&) character. If the ampersand character is not used, the font is not downloaded. If it is used, the entire font file is downloaded. For this reason, it is sometimes good to optimize your font files to only include characters in the specified unicode range, especially if, like in this case, you're only using one character from a font that could contain several thousand characters.

In the second case, a single range is specified, spanning Unicode character code point 590 through code point 5FF. This covers the 111 total characters used when writing Hebrew. Thus, an author might specify a Hebrew font and restrict it to only be used for Hebrew characters, even if the face contains glyphs for other code points:

```
@font-face {
    font-family: "CMM-Ahuvah";
    src: url("cmm-ahuvah.otf" format("opentype");
    unicode-range: U+590-5FF;
}
```

In the third case, a series of ranges are specified in a comma-separated list to cover all the Japanese characters. The interesting feature there is the `U+30??` value, with a question mark, which is a special format permitted in `unicode-range` values. The question marks are wildcards meaning "any possible digit," making `U+30??` equivalent to `U+3000-30FF`. The question mark is the only "special" character pattern permitted in the value.

Ranges must always ascend. Any descending range, such as `U+400-300`, is treated as a parsing error and ignored.

Because `@font-face` is designed to optimize lazy loading, it's possible to use `unicode-range` to download only the font faces a page actually needs, with possibly a much smaller file size when using a font file optimized to contain only the defined subset character range. If the page doesn't use any character in the range, the font is not downloaded. If a single character on a page requires a font, the whole font is downloaded.

Suppose you have a website that uses a mixture of English, Russian, and basic mathematical operators, but you don't know which will appear on any given page. There could be all English, a mixture of Russian and math, and so on. Furthermore, suppose you have special font faces for all three types of content. You can make sure a user agent only downloads the faces it actually needs with a properly-constructed series of `@font-face` rules:

```
@font-face {
```

```
    font-family: "MyFont";
    src: url("myfont-general.otf" format("opentype");
}
@font-face {
    font-family: "MyFont";
    src: url("myfont-cyrillic.otf" format("opentype");
    unicode-range: U+04??, U+0500-052F, U+2DE0-2DFF, U+A640-A69F, U+1D2B-1D78;
}
@font-face {
    font-family: "MyFont";
    src: url("myfont-math.otf" format("opentype");
    unicode-range: U+22??;   /* equivalent to U+2200-22FF */
}

body {font-family: MyFont, serif;}
```

Because the first rule doesn't specify a Unicode range, it is always downloaded—unless a page happens to contain no characters at all (and maybe even then). The second rule causes `myfont-cyrillic.otf` to be downloaded only if the page contains characters in its declared Unicode range; the third rule does the same for mathematical symbols.

If the content calls for the mathematical character U+2222 (, the "spherical angle" character), `myfont-math.otf` will be downloaded and the character from `myfont-math.otf` will be used, even if `myfont-general.otf` has that character.

A more likely way to use this capability would be our ampersand example; we can include a fancy ampersand from a cursive font and use it in place of the ampersand found in a headline font. Something like this:

```
@font-face {
    font-family: "Headline";
    src: url("headliner.otf" format("opentype");
}
@font-face {
    font-family: "Headline";
    src: url("cursive-font.otf" format("opentype");
    unicode-range: U+0026;
}

h1, h2, h3, h4, h5, h6 {font-face: Headline, cursive;}
```

In a case like this, to keep page weights low, take a cursive font (that you have the rights to use) and minimize it down to contain just the ampersand character. You can use a tool like Font Squirrel to create a single character font file.

NOTE

Remember that pages can be translated with automated services like Google Translate. If you too aggressively restrict your Unicode ranges, say to the range of unaccented letters used in English, an auto-translated version of the page into French or Swedish, for example, could end up a mishmash of characters in different font faces, as the accented characters in those languages would use a fallback font and the unaccented characters would be in your intended font.

Font display

If you're a designer or developer of a certain vintage, you may remember the days of FOUC: the Flash of Unstyled Content. This happened in earlier browsers that would load the HTML and display it to the screen before the CSS was finished loading, or, at least, before the layout of the page via CSS was finished.

FOUT (Flash of Unstyled Text) happens when a browser has loaded the page and the CSS and displays the laid-out page, along with all the text, before it's done loading custom fonts. FOUT causes text to appear in the default font, or a fallback font, before being replaced by text using the custom-loaded font.

There is a cousin to this problem, which is the FOIT (Flash of Invisible Text). FOIT was the user-agent solution to FOUT, and is caused when the browser detects if text is set in a custom font that hasn't loaded yet and makes the text invisible until the font loads or a certain amount of time passed.

Since the replacement of text can change its size, whether via FOUT or FOIT, take care when selecting fallback fonts. If there is a significant height difference between the font used to initially display the text and the custom font eventually loaded and used, significant page reflows are likely to occur.

In an attempt to help with this, the `font-display` descriptor guides the browser to proceed with text rendering when a web font has yet to load.

FONT-DISPLAY DESCRIPTOR

Values	`auto` | `block` | `swap` | `fallback` | `optional`
Initial value	`auto`
Applies to	All elements
Computed value	as specified

What we can call the "font display timeline timer" starts when the user agent first paints the page. The timeline is divided into the three periods: block, swap, and failure.

During the font-block period, if the font face is not loaded, the browser renders any content that should use that font using an invisible fallback font face, meaning the text content is not visible but the space is reserved. If the font loads successfully during the block period, the text is rendered with the downloaded font and made visible.

During the swap period, if the font face is not loaded, the browser renders the content using a visible fallback font face, most likely one it has installed locally (e.g., Helvetica). If the font loads successfully, the fallback font face is swapped to the downloaded font.

Once the failure period is entered, the user agent treats it as a failed load, falls back to an available font, and will not swap the font if it does eventually load. If the swap period is infinite, the failure period is never entered.

The values of the `font-display` descriptor match these periods of the timeline, and their effect is to emphasize one part of the timeline at the expense of the others. The effects are summarized in "font-display values".

FONT-DISPLAY VALUES

Value	Block period*	Swap period*	Failure period*
`auto`	Browser defined	Browser defined	Browser defined
`block`	3s	infinite	*n/a*
`swap`	<100ms	infinite	*n/a*
`fallback`	<100ms	3s	infinite
`optional`	0	0	infinite

- Recommended period length; actual times may vary

Let's consider each value in turn.

`block` tells the browser to hold open space for the font for a few seconds (three is what the specification recommends, but browsers may choose their own values), and then enters an infinitely long swap period. That means that if the font ever finally loads, even if it's 10 minutes later, the fallback font that was used in its place will be replaced with the loaded font.

`swap` is similar, except it doesn't hold the space open for longer than a fraction of a second (100 milliseconds is the recommendation). A fallback font is then used, and is replaced with the intended font whenever it finally loads.

`fallback` gives the same brief block window that `swap` does, and then enters a short period in which the fallback font can be replaced by the intended font. If that short period (three seconds is recommended) is exceeded, then the fallback font is used indefinitely, and the user agent may cancel the download of the intended font since there will never be a swap.

`optional` is the most stringent of them all: if the font isn't immediately available at first paint, then it goes straight to the fallback font and skips right over the block and swap periods to sit in the failure period for the rest of the page's life.

Combining Descriptors

Something that might not be immediately obvious is that you can supply multiple descriptors in order to assign specific faces for specific property combinations. For example, you can assign one face to bold text, another to italic text, and a third to text that is both bold and italic.

This is actually implicit in the fact that any undeclared descriptor is assigned its default value. Let's

consider a basic set of three face assignments, using both descriptors we've covered and a few we'll get to in a bit:

```
@font-face {
    font-family: "Switzera";
    font-weight: normal;
    font-style: normal;
    font-stretch: normal;
    src: url("SwitzeraADF-Regular.otf") format("opentype");
}
@font-face {
    font-family: "Switzera";
    font-weight: 500;
    font-style: normal;
    font-stretch: normal;
    src: url("SwitzeraADF-Bold.otf") format("opentype");
}
@font-face {
    font-family: "Switzera";
    font-weight: normal;
    font-style: italic;
    font-stretch: normal;
    src: url("SwitzeraADF-Italic.otf") format("opentype");
}
```

You may have noticed that we've explicitly declared some descriptors with their default values, even though we didn't need to. The previous example is exactly the same as a set of three rules in which we remove every descriptor that shows a value of `normal`:

```
@font-face {
   font-family: "Switzera";
   src: url("SwitzeraADF-Regular.otf") format("opentype");
}
@font-face {
   font-family: "Switzera";
   font-weight: 500;
   src: url("SwitzeraADF-Bold.otf") format("opentype");
}
@font-face {
   font-family: "Switzera";
   font-style: italic;
   src: url("SwitzeraADF-Italic.otf") format("opentype");
}
```

In all three rules, there is no font-stretching beyond the default `normal` amount, and the values of `font-weight` and `font-style` vary by which face is being assigned. So what if we want to assign a specific face to unstretched text that's both bold and italic?

```
@font-face {
   font-family: "Switzera";
   font-weight: bold;
   font-style: italic;
   font-stretch: normal;
   src: url("SwitzeraADF-BoldItalic.otf") format("opentype");
}
```

And then what about bold, italic, condensed text?

```
@font-face {
   font-family: "Switzera";
   font-weight: bold;
   font-style: italic;
   font-stretch: condensed;
   src: url("SwitzeraADF-BoldCondItalic.otf") format("opentype");
}
```

How about normal-weight, italic, condensed text?

```
@font-face {
   font-family: "Switzera";
   font-weight: normal;
   font-style: italic;
   font-stretch: condensed;
   src: url("SwitzeraADF-CondItalic.otf") format("opentype");
}
```

We could keep this up for quite a while, but let's stop there. If we take all those rules and strip out anything with a `normal` value, we end up with the following result, illustrated in Figure 10-3:

```
@font-face {
   font-family: "Switzera";
   src: url("SwitzeraADF-Regular.otf") format("opentype");
}
@font-face {
   font-family: "Switzera";
   font-weight: bold;
   src: url("SwitzeraADF-Bold.otf") format("opentype");
}
@font-face {
   font-family: "Switzera";
   font-style: italic;
   src: url("SwitzeraADF-Italic.otf") format("opentype");
}
@font-face {
   font-family: "Switzera";
   font-weight: bold;
   font-style: italic;
   src: url("SwitzeraADF-BoldItalic.otf") format("opentype");
}
@font-face {
   font-family: "Switzera";
   font-weight: bold;
   font-stretch: condensed;
   src: url("SwitzeraADF-BoldCond.otf") format("opentype");
}
@font-face {
   font-family: "Switzera";
   font-style: italic;
   font-stretch: condensed;
   src: url("SwitzeraADF-CondItalic.otf") format("opentype");
}
@font-face {
```

```
    font-family: "Switzera";
    font-weight: bold;
    font-style: italic;
    font-stretch: condensed;
    src: url("SwitzeraADF-BoldCondItalic.otf") format("opentype");
}
```

This element contains serif text, unstretched **bold** and *italic* text in SwitzeraADF, and unstretched ***bold and italic*** text in SwitzeraADF.

This element contains serif text, condensed **bold** and *italic* text in SwitzeraADF, and condensed ***bold and italic*** text in SwitzeraADF.

Figure 10-3. Employing a variety of faces

If you declare `html { +font-family: switzera;}`, you don't have to declare the font family again for additional selectors that use `switzera`, as the browser will use the correct font file for your bold, italic, stretched, and normal text depending on what your selector specific values for the `font-weight`, `font-style`, and `font-stretch` property values are.

The point is, we can have a specific font file for every weight, style and stretch. The ability to declare all the variations within a few `@font-face` rules with a single `font-family` name ensures cohesive typeface design and avoids font synthesis even when using non-variable fonts. Declaring all the variations of a font via @font-face, with the same `font-family` descriptor name, reduces `font-family` property overrides, reducing the chance of other developers on your team using the wrong font file for a specific selector.

As you can see, when using standard fonts, there are a lot of possible combinations just for those three descriptors—consider that there are 10 possible values for `font-stretch`—but you'll likely never have to run through them all. In fact, most font families don't have as many faces as SwitzeraADF offers (24 at last count), so there wouldn't be much point in writing out all the possibilities. Nevertheless, the options are there, and in some cases you may find that you need to assign, say, a specific face for bold condensed text so that the user agent doesn't try to compute them for you. Or else use a variable font that has weight and condensing axes.

Now that we've covered `@font-face` and provided an overview of a few descriptors, let's get back to properties.

Font Weights

Most of us are used to normal and bold text, at the very least, which are sort of the two most basic font weights available. CSS gives you a lot more control over font weights with the property `font-weight`.

FONT-WEIGHT

Values	`normal` | `bold` | `bolder` | `lighter` | *<number>*
Initial value	`normal`
Applies to	All elements
Computed value	One of the numeric values (`100`, etc.), or one of the numeric values plus one of the relative values (`bolder` or `lighter`)
@font-face equivalent	`font-weight`
Variable axis	`"wght"`
Inherited	Yes
Animatable	No

The *<number>* value can be from 1 to 1000, inclusive, where 1 is the lightest and 1000 is the heaviest possible weight. Unless you are using variable fonts, discussed later, there are almost always limited weights available for a font family (sometimes there is only a single weight).

Generally speaking, the heavier a font weight becomes, the darker and "more bold" a font appears. There are a great many ways to label a heavy font face. For example, the font family known as SwitzeraADF has a number of variants, such as SwitzeraADF Bold, SwitzeraADF Extra Bold, SwitzeraADF Light, and SwitzeraADF Regular. All of these use the same basic font shapes, but each has a different weight.

If the specified weight doesn't exist, a nearby weight is used. Table 10-4 lists the numbers used for each of the commonly accepted font weight labels, as defined in the `"wght"` variation axis. If a font has only two weights corresponding to `400` and `700` (normal and bold), then any number value for `font-weight` will be mapped to the closest value. Thus, any `font-weight` value from `1` through `550` will be mapped to `400`, and any value greater than `550` up through `1000` will be mapped to `700`.

Table 10-4. Weight mappings

Value	Mapping
1	Lowest valid value
100	Thin
200	Extra Light (Ultra Light)
300	Light
400	Normal
500	Medium
600	Semi Bold (Demi Bold)
700	Bold
800	Extra Bold (Ultra Bold)
900	Black (Heavy)
950	Extra Black (Ultra Black)
1000	Highest valid value

Let's say that you want to use SwitzeraADF for a document, but you'd like to make use of all those different heaviness levels. If your user has all the font files locally on their machine and you didn't use `@font-face` to rename all the options to "Switzera", you could refer to them directly through the `font-family` property… but you really shouldn't have to do that. It's no fun having to write a stylesheet like this:

```
h1 {font-family: 'SwitzeraADF Extra Bold', sans-serif;}
h2 {font-family: 'SwitzeraADF Bold', sans-serif;}
h3 {font-family: 'SwitzeraADF Bold', sans-serif;}
h4, p {font-family: 'SwitzeraADF Regular', sans-serif;}
small {font-family: 'SwitzeraADF Light', sans-serif;}
```

That's pretty tedious. This is a perfect example of why specifying a single font family for the whole document and then assigning different weights to various elements by using `@font-face` is so powerful: you can include several `@font-face` declarations, each with the same `font-family` name, but with various values for the `font-weight` descriptors. Then you can use different font files with fairly simple `font-weight` declarations:

```
strong {font-weight: bold;}
b {font-weight: bolder;}
```

The first declaration says the `strong` element should be displayed using a bold font face; or, to put it another way, a font face that is heavier than the normal font face. The second declaration says `b` should

use a font face that is the inherited `font-weight` value plus 100.

What's really happening behind the scenes is that heavier faces of the font are used for displaying `strong` and `b` elements. Thus, if you have a paragraph displayed using Times, and part of it is bold, then there are really two faces of the same font in use: Times and TimesBold. The regular text is displayed using Times, and the bold and bolder text are displayed using TimesBold.

If the font doesn't have a bold face version, it may be synthesized by the browser, creating a "faux" bold. (To prevent this, use `font-synthesis` property, which is described later.)

How Weights Work

To understand how a user agent determines the heaviness, or weight, of a given font variant (not to mention how weight is inherited), it's easiest to start by talking about the values `1` through `1000` inclusive, specifically the values divisible by 100, `100` through `900`. These number values were defined to map to a relatively common feature of font design in which a font is given nine levels of weight. If a non-variable font family has faces for all nine weight levels available, then the numbers are mapped directly to the predefined levels, with `100` as the lightest variant of the font and `900` as the heaviest.

In fact, there is no intrinsic weight in these numbers. The CSS specification says only that each number corresponds to a weight at least as heavy as the number that precedes it. Thus, `100`, `200`, `300`, and `400` might all map to a single relatively lightweight variant; `500` and `600` could correspond to a single medium-heavy font variant; and `700`, `800`, and `900` could all produce the same very heavy font variant. As long as no number corresponds to a variant that is lighter than the variant assigned to the previous lower number, everything will be all right.

When it comes to non-variable fonts, these numbers are defined to be equivalent to certain common variant names. `400` is defined to be equivalent to `normal`, and `700` corresponds to `bold`.

A user agent has to do some calculations if there are fewer than nine weights in a given font family. In this case, it must fill in the gaps in a predetermined way:

- If the value `500` is unassigned, it is given the same font weight as that assigned to `400`.
- If `300` is unassigned, it is given the next variant lighter than `400`. If no lighter variant is available, `300` is assigned the same variant as `400`. In this case, it will usually be "Normal" or "Medium." This method is also used for `200` and `100`.
- If `600` is unassigned, it is given the next variant darker than that assigned for `500`. If no darker variant is available, `600` is assigned the same variant as `500`. This method is also used for `700`, `800`, and `900`.

To illustrate this weighting scheme more clearly, let's look at a couple examples of font weight assignment. In the first example, assume that the font family "Karrank%" is an OpenType font, so it has nine weights already defined. In this case, the numbers are assigned to each level, and the keywords `normal` and `bold` are assigned to the numbers `400` and `700`, respectively.

In our second example, consider the font family SwitzeraADF. Hypothetically, its variants might be

assigned numeric values for `font-weight`, as shown in Table 10-5.

Table 10-5. Hypothetical weight assignments for a specific font family

Font face	Assigned keyword	Assigned number(s)
SwitzeraADF Light		`100` through `300`
SwitzeraADF Regular	`normal`	`400`
SwitzeraADF Medium		`500`
SwitzeraADF Bold	`bold`	`600` through `700`
SwitzeraADF Extra Bold		`800` through `900`

The first three number values are assigned to the lightest weight. The “Regular” face gets the keyword `normal` and the number weight `400`. Since there is a “Medium” font, it’s assigned to the number `500`. There is nothing to assign to `600`, so it’s mapped to the “Bold” font face, which is also the variant to which `700` and `bold` are assigned. Finally, `800` and `900` are assigned to the “Black” and “UltraBlack” variants, respectively. Note that this last assignment would happen only if those faces had the top two weight levels already assigned. Otherwise, the user agent might ignore them and assign `800` and `900` to the “Bold” face instead, or it might assign them both to one or the other of the “Black” variants.

`font-weight` is inherited, so if you set a paragraph to be `bold`:

```
p.one {font-weight: bold;}
```

Then all of its children will inherit that boldness, as we see in Figure 10-4.

Within this paragraph we find some *italicized text*, a bit of <u>underlined text</u>, and the occasional stretch of <u>hyperlinked text</u> for our viewing pleasure.

Figure 10-4. Inherited font-weight

This isn’t unusual, but the situation gets interesting when you use the last two values we have to discuss: `bolder` and `lighter`. In general terms, these keywords have the effect you’d anticipate: they make text more or less bold compared to its parent’s font weight. How they do so is slightly complicated. First, let’s consider `bolder`.

If you set an element to have a weight of `bolder` or `lighter`, then the user agent first must determine what `font-weight` value was inherited from the parent element. Once it has that number (say, `400`), it then changes the value as shown in Table 10-6.

Table 10-6. `bolder` and `lighter` weight mappings

Inherited value	`bolder`	`lighter`
value < 100	400	No change
100 ≤ value < 350	400	100
350 ≤ value < 550	700	100
550 ≤ value < 750	900	400
750 ≤ value < 900	900	700
900 ≤ value	No change	700

Thus, you might encounter the following situations, illustrated in Figure 10-5:

```
p {font-weight: normal;}
p em {font-weight: bolder;}  /* inherited value '400', evaluates to '700' */

h1 {font-weight: bold;}
h1 b {font-weight: bolder;}  /* inherited value '700', evaluates to '900' */

div {font-weight: 100;}
div strong {font-weight: bolder;} /* inherited value '100', evaluates to '400' */
```

Within this paragraph we find some ***emphasized text***.

This H1 contains bold text!

Meanwhile, this DIV element has some strong text but it shouldn't look much different, at least in terms of font weight.

Figure 10-5. Text trying to be bolder

In the first example, the user agent moves up from `400` to `700`. In the second example, `h1` text is already set to `bold`, which equates to `700`. If there is no bolder face available, then the user agent sets the weight of `b` text within an `h1` to `900`, since that is the next step up from `700`. Since `900` is assigned to the same font face as `700`, there is no visible difference between normal `h1` text and bold `h1` text, but the weight values are different nonetheless.

As you might expect, `lighter` works in much the same way, except it causes the user agent to move down the weight scale instead of up.

The font-weight descriptor

With the `font-weight` descriptor, authors can assign faces of varying weights to the weighting levels permitted by the `font-weight` property. The allowable values are different for the descriptor, which

supports `auto`, `normal`, `bold`, or one to two numeric values as a range. Neither `lighter` nor `bolder` are supported.

For example, the following rules explicitly assign five faces to six different `font-weight` values:

```
@font-face {
   font-family: "Switzera";
   font-weight: 1 250;
   src: url("f/SwitzeraADF-Light.otf") format("opentype");
}
@font-face {
   font-family: "Switzera";
   font-weight: normal;
   src: url("f/SwitzeraADF-Regular.otf") format("opentype");
}
@font-face {
   font-family: "Switzera";
   font-weight: 500 600;
   src: url("f/SwitzeraADF-DemiBold.otf") format("opentype");
}
@font-face {
   font-family: "Switzera";
   font-weight: bold;
   src: url("f/SwitzeraADF-Bold.otf") format("opentype");
}
@font-face {
   font-family: "Switzera";
   font-weight: 800 1000;
   src: url("f/SwitzeraADF-ExtraBold.otf") format("opentype");
}
```

With these faces assigned, the author now has a number of weighting levels available for their use, as illustrated in Figure 10-6:

```
h1, h2, h3, h4 {font-family: SwitzeraADF, Helvetica, sans-serif;}
h1 {font-size: 225%; font-weight: 900;}
h2 {font-size: 180%; font-weight: 700;}
h3 {font-size: 150%; font-weight: 500;}
h4 {font-size: 125%; font-weight: 300;}
```

A Level 1 Heading Element

A Level 2 Heading Element

A Level 3 Heading Element

A Level 4 Heading Element

Figure 10-6. Using declared font-weight faces

In any given situation, the user agent picks which face to use depending on the exact value of a `font-`

`weight` property, using the resolution algorithm detailed in the earlier section, "How Weights Work". While the font-weight property has numerous keyword values, the `font-weight` descriptor only accepts `normal` and `bold` as keywords, and any number between 1 and 1000 inclusive.

Font Size

While size doesn't have a `@font-face` descriptor, we need to understand the `font-size` property to better understand some of the descriptors to come, so we'll explore it now. The methods for determining font size are both very familiar and very different.

FONT-SIZE

Values	`xx-small` | `x-small` | `small` | `medium` | `large` | `x-large` | `xx-large` | `xxx-large` | `smaller` | `larger` | *<length>* | *<percentage>*
Initial value	`medium`
Applies to	All elements
Percentages	Calculated with respect to the parent element's font size
Computed value	An absolute length
Inherited	Yes
Animatable	Yes (numeric keywords only)

What can be a real head-scratcher at first is that different fonts declared to be the same size may not appear to be the same size. This is because the actual relation of the `font-size` property to what you see rendered is determined by the font's designer. This relationship is set as an *em square* (some call it an *em box*) within the font itself. This em square (and thus the font size) doesn't have to refer to any boundaries established by the characters in a font. Instead, it refers to the distance between baselines when the font is set without any extra leading (`line-height` in CSS).

The effect of `font-size` is to provide a size for the em box of a given font. This does not guarantee that any of the actual displayed characters will be this size. It is quite possible for fonts to have characters that are taller than the default distance between baselines. For that matter, a font might be defined such that all of its characters are smaller than its em square, as many fonts do. Some hypothetical examples are shown in Figure 10-7.

A font with an em square smaller than some characters.

A font with an em square taller than all characters.

A font with an em square which is exactly large enough to contain all characters.

Figure 10-7. Font characters and em squares

Absolute Sizes

Having established all that, we turn now to the absolute-size keywords. There are seven absolute-size values for `font-size`: `xx-small`, `x-small`, `small`, `medium`, `large`, `x-large`, `xx-large`, and the relatively new `xxx-large`. These are not defined precisely, but instead are defined relative to each other, as Figure 10-8 demonstrates:

```
p.one {font-size: xx-small;}
p.two {font-size: x-small;}
p.three {font-size: small;}
p.four {font-size: medium;}
p.five {font-size: large;}
p.six {font-size: x-large;}
p.seven {font-size: xx-large;}
p.eight {font-size: xxx-large;}
```

This paragraph (class 'one') has a font size of 'xx-small'.

This paragraph (class 'two') has a font size of 'x-small'.

This paragraph (class 'three') has a font size of 'small'.

This paragraph (class 'four') has a font size of 'medium'.

This paragraph (class 'five') has a font size of 'large'.

This paragraph (class 'six') has a font size of 'x-large'.

This paragraph (class 'seven') has a font size of 'xx-large'.

This paragraph (class 'eight') has a font size of 'xxx-large'.

Figure 10-8. Absolute font sizes

In the CSS1 specification, the difference (or *scaling factor*) between one absolute size and the next was 1.5 going up the ladder, or 0.66 going down. This was determined to be too large a scaling factor. In CSS2, the suggested scaling factor for computer screen between adjacent indexes was 1.2. This didn't resolve all the issues, though, as it created issues for the small sizes.

The CSS Fonts Level 4 specification doesn't have a one-size-fits-all scaling factor. Rather, each absolute-size keyword value has a size-specific scaling factor based on the value of `medium`. `small` is listed as eight-ninths the size of `medium`, while `xx-small` is three-fifths. In any case, the scaling factors are guidelines, and user agents are free to alter them for any reason.

Table 10-7. font-size mappings

CSS absolute-size values	`xx-small`	`x-small`	`small`	`medium`	`large`	`x-large`
Scaling factor	3/5	3/4	8/9	1	6/5	3/2
Sizes at `medium`== `16px`	`9px`	`10px`	`13px`	`16px`	`18px`	`24px`
HTML heading equivalent	`h6`	-	`h5`	`h4`	`h3`	`h2`

Note that we've set the default size, `medium`, explicitly to `16px`. The default `font-size` value is the same, `medium`, for all generic font families, but the `medium` keyword may have different definitions based on operating system or browser user settings. For example, in many browsers, serif and sans-serif fonts have `medium` equal to `16px`, but `monospace` set to `13px`.

> **WARNING**
>
> As of late 2022, the `xxx-large` keyword was not supported by Safari or Opera, either on desktop or mobile.

Relative Sizes

In a fashion very similar to the `font-weight` keywords `bolder` and `lighter`, the property `font-size` has relative-size keywords called `larger` and `smaller`. Much as with relative font weights, these keywords cause the computed value of `font-size` to move up and down a scale of size values.

The `larger` and `smaller` keywords are fairly straightforward: they cause the size of an element to be shifted up or down the absolute-size scale, relative to their parent element:

```
p {font-size: medium;}
strong, em {font-size: larger;}

<p>This paragraph element contains <strong>a strong-emphasis element
which itself contains <em>an emphasis element that also contains
```

```
<strong>a strong element.</strong></em></strong></p>

<p> medium <strong>large <em> x-large <strong>xx-large</strong> </em> </strong>
    </p>
```

Unlike the relative values for weight, the relative-size values are not necessarily constrained to the limits of the absolute-size range. Thus, a font's size can be pushed beyond the sizes for `xx-small` and `xxx-large`. If the parent element font-size is the largest or smallest absolute value, the browser will use a scaling factor between 1.2 and 1.5 to create an even smaller or larger font size. For example:

```
h1 {font-size: xxx-large;}
em {font-size: larger;}

<h1>A Heading with <em>Emphasis</em> added</h1>
<p>This paragraph has some <em>emphasis</em> as well.</p>
```

A Heading with *Emphasis* added

This paragraph has some *emphasis* as well.

xxx-large *(larger)* xxx-large

Figure 10-9. Relative font sizing at the edges of the absolute sizes

As you can see in Figure 10-9, the emphasized text in the `h1` element is slightly larger than `xxx-large`. The amount of scaling is left up to the user agent, with a scaling factor in the range of 1.2 to 1.5 being

preferred, but not required. The `em` text in the paragraph is shifted one slot up 140%.

> **WARNING**
>
> User agents are not required to increase or decrease font size beyond the limits of the absolute-size keywords, but they may do so anyway. Also, while it is technically possible to declare `smaller` than `xx-small`, small text can be very difficult to read on-screen, leading to content being not accessible to users. Use very small text sparingly, and with a great deal of caution.

Percentages and Sizes

In a way, percentage values are very similar to the relative-size keywords. A percentage value is always computed in terms of whatever size is inherited from an element's parent. Unlike the size keywords previously discussed, percentages permit much finer control over the computed font size. Consider the following example, illustrated in Figure 10-10:

```
body {font-size: 15px;}
p {font-size: 12px;}
em {font-size: 120%;}
strong {font-size: 135%;}
small, .fnote {font-size: 70%;}

<body>
<p>This paragraph contains both <em>emphasis</em> and <strong>strong
emphasis</strong>, both of which are larger than their parent element.
The <small>small text</small>, on the other hand, is smaller by a quarter.</p>
<p class="fnote">This is a 'footnote' and is smaller than regular text.</p>

<p> 12px <em> 14.4px </em> 12px <strong> 16.2px </strong> 12px
<small> 9px </small> 12px </p>
<p class="fnote"> 10.5px </p>
</body>
```

Figure 10-10. Throwing percentages into the mix

In this example, the exact pixel size values are shown. These are the values calculated by the browser, regardless of the actual displayed size of the characters onscreen, which may have been rounded to the nearest whole number of pixels.

When using `em` measurements, the same principles apply as with percentages, such as the inheritance of computed sizes and so forth. CSS defines the length value `em` to be equivalent to percentage values, in the sense that `1em` is the same as `100%` when sizing fonts. Thus, the following would yield identical results, assuming that both paragraphs have the same parent element:

```
p.one {font-size: 166%;}
```

```
p.two {font-size: 1.66em;}
```

As with the relative-size keywords, percentages are effectively cumulative. Thus, the following markup is displayed as shown in Figure 10-11:

```
p {font-size: 12px;}
em {font-size: 120%;}
strong {font-size: 135%;}

<p>This paragraph contains both<em>emphasis and <strong>strong
emphasis</strong></em>, both of which are larger than the paragraph text. </p>

<p>12px <em>14.4px <strong> 19.44px </strong></em> 12px</p>
```

This paragraph contains both*emphasis and* ***strong emphasis***, both of which are larger than the paragraph text.

12px *14.4px* ***19.44px*** 12px

Figure 10-11. The issues of inheritance

The size value for the `strong` element shown in Figure 10-11 is computed as follows:

$$12 \text{ px} \times 120\% = 14.4 \text{ px} + 14.4 \text{ px} \times 135\% = 19.44 \text{ px}$$

The problem of runaway scaling can go the other direction, too. Imagine the effect of the following rule on a nested list item if we have lists nested four levels deep:

```
ul {font-size: 80%;}
```

The unordered list nested four levels deep would have a computed `font-size` value 40.96 percent the size of the parent of the top-level list. Every nested list would have a font size 80 percent as big as its parent list, causing each level to become harder and harder to read.

Automatically Adjusting Size

Two of the main factors that influence a font’s legibility are its size and its *x-height*, which is the height of a lowercase “x” character in the font. The number that results from dividing the x-height by the `font-size` is referred to as the *aspect value*. Fonts with higher aspect values tend to be legible as the font’s size is reduced; conversely, fonts with low aspect values become illegible more quickly. CSS provides a way to deal with shifts in aspect values between font families, as well as ways to use different metrics to compute an aspect value, with the property `font-size-adjust`.

FONT-SIZE-ADJUST

<table>
<tr><td>Values</td><td><code>[ex-height | cap-height | ch-width | ic-width | ic-height]? [from-font | </code><<i>number</i>><code>] | none | auto</code></td></tr>
<tr><td>Initial value</td><td><code>none</code></td></tr>
<tr><td>Applies to</td><td>All elements</td></tr>
<tr><td>@font-face equivalent</td><td><code>size-adjust</code></td></tr>
<tr><td>Inherited</td><td>Yes</td></tr>
<tr><td>Animatable</td><td>Yes</td></tr>
</table>

The goal of this property is to preserve legibility when the font used is not the author’s first choice. Because of the differences in font appearance, while one font may be legible at a certain size, another font at the same size is difficult or impossible to read.

The property value can be `none`, `from-font`, or a number. The number specified should generally be the aspect value (the ratio of a given font metric to font size) of the first-choice font-family. In order to pick the font metric used to compute the aspect ratio, you can add a keyword specifying it. If not included, it defaults to `ex-height`, which normalizes the aspect value of the fonts using the x-height divided by the font size.

The other possibilities for the font metric keyword are:

`cap-height`

Use the cap-height (height of capital letters) of the font.

`ch-width`

Use the horizontal pitch (also the width of `1ch`) of the font.

`ic-width`

Use the width of the font using the CJK water ideograph, “” (U+6C34), of the font.

`ic-height`

Use the height of the ideograph “” (U+6C34), of the font.

Declaring `font-size-adjust: none` will suppress any adjustment of font sizes. This is the default state.

The `from-font` keyword directs the user agent to use the built-in value of the specified font metric from the first available font, rather than requiring the author to figure out what that value is and write it

explicitly. Thus, writing `font-size-adjust: cap-height from-font` will automatically set an aspect value by dividing the cap-height by the em-square height.

A good example is to compare the common fonts Verdana and Times. Consider Figure 10-12 and the following markup, which shows both fonts at a `font-size` of `10px`:

```
p {font-size: 10px;}
p.cl1 {font-family: Verdana, sans-serif;}
p.cl2 {font-family: Times, serif; }
```

Donec ut magna. Aliquam erat volutpat. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Nulla facilisi. Aenean mattis, dui et ullamcorper ornare, erat est sodales mi, non blandit sem ipsum quis justo. Nulla tincidunt.

Quisque et orci nec lacus hendrerit fringilla. Sed quam nibh, elementum et, scelerisque a, aliquam vestibulum, sapien. Etiam commodo auctor sapien. Pellentesque tincidunt lacus nec quam. Integer sit amet neque vel eros interdum ornare. Sed consequat.

Figure 10-12. Comparing Verdana and Times

The text in Times is much harder to read than the Verdana text. This is partly due to the limitations of pixel-based display, but it is also because Times becomes harder to read at smaller font sizes.

As it turns out, the ratio of x-height to character size in Verdana is 0.58, whereas in Times it is 0.46. What you can do to make these font faces look more consistent with each other is declare the aspect value of Verdana, and have the user agent adjust the size of the text that's actually used. This is accomplished using the formula:

> Declared `font-size` × (`font-size-adjust` value ÷ aspect value of available font) = Adjusted `font-size`

So, in a situation where Times is used instead of Verdana, the adjustment is as follows:

> `10px` × (`0.58` ÷ `0.46`) = `12.6px`

which leads to the result shown in Figure 10-13:

```
p {font: 10px Verdana, sans-serif; font-size-adjust: ex-height 0.58;}
p.cl2 {font-family: Times, serif; }
```

Donec ut magna. Aliquam erat volutpat. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Nulla facilisi. Aenean mattis, dui et ullamcorper ornare, erat est sodales mi, non blandit sem ipsum quis justo. Nulla tincidunt.

Quisque et orci nec lacus hendrerit fringilla. Sed quam nibh, elementum et, scelerisque a, aliquam vestibulum, sapien. Etiam commodo auctor sapien. Pellentesque tincidunt lacus nec quam. Integer sit amet neque vel eros interdum ornare. Sed consequat.

Figure 10-13. Adjusting Times

The catch is that for a user agent to intelligently make size adjustments, it first has to know the aspect value of the fonts you specify. User agents that support `@font-face` will be able to pull that information directly from the font file, assuming the files contain the information—any professionally-produced font should, but there's no guarantee. If a font file doesn't contain the aspect value, a user agent may try to compute it; but again, there's no guarantee that they will or even can.

If the user agent can't find or figure out aspect values on its own, the `auto` value for `font-size-adjust` is a way of getting the desired effect even if you don't know the actual aspect value of your first-choice font. For example, assuming that the user agent can determine that the aspect value of Verdana is 0.58, then the following will have the same result as that shown in Figure 10-13:

```
p {font: 10px Verdana, sans-serif; font-size-adjust: auto;}
p.cl2 {font-family: Times, serif; }
```

> **WARNING**
>
> As of late 2022, the only user agent line to support `font-size-adjust` was the Gecko (Firefox) family.

Understanding font size adjustment comes in handy when considering the `size-adjust` font descriptor. The `size-adjust` font descriptor behaves in a similar fashion to the `font-size-adjust` property, though it's restricted to comparing only x- heights instead of the range of font metrics available for `font-size-adjust`.

SIZE-ADJUST DESCRIPTOR

Values	*<percentage>*
Initial value	`100%`

The `font-size-adjust` property is a rare case where the property and descriptor names are not the same: The descriptor is `size-adjust`. The value is any positive percentage value (from zero to infinity) by which you want the fallback font scaled so it better matches the primary font selected. That percentage is used as a multiplier for the glyph outline sizes and other metrics of the font.

```
@font-face {
  font-family: myPreferredFont;
  src: url("longLoadingFont.otf");
}

@font-face {
  font-family: myFallBackFont;
  src: local(aLocalFont);
  size-adjust: 87.3%;
}
```

> **WARNING**
>
> As of late 2022, the only user agent line that did *not* support the `size-adjust` descriptor was the WebKit (Safari) family.

Font Style

`font-style` sounds very simple: you can choose between three values, and optionally provide an angle for oblique text if you're using it.

FONT-STYLE

Values	`italic` ||[`oblique` <*angle*>?] | `normal`
Initial value	`normal`
Applies to	All elements
Computed value	As specified
@font-face equivalent	`font-style`
Variable axis	`"slnt"` (slant) or `"ital"` (italic)
Inherited	Yes
Animatable	Yes for variable fonts that define a ranged axis for italic or oblique; otherwise no

The default value of `font-style` is `normal`. This value refers to *upright* text, which is best described as text that is not italic or otherwise slanted. For instance, the vast majority of text in this book is upright.

Italic font faces are usually a bit cursive in appearance, and generally use less horizontal space than the `normal` version of the same font. In standard fonts, italic text is a separate font face, with small changes made to the structure of each letter to account for the altered appearance. This is especially true of serif fonts where, in addition to the fact that the text characters "lean," the serifs may be altered. Font faces with labels like "Italic," "Cursive," and "Kursiv" are usually mapped to the `italic` keyword.

Oblique text, on the other hand, is a slanted version of the normal, upright text. Oblique text is generally not altered from the upright text other than being given a slope. If a font has oblique versions, they are often in faces with labels such as "Oblique," "Slanted," and "Incline."

When fonts don't have italic or oblique versions, the browser can simulate italic and oblique fonts by artificially sloping the glyphs of the regular face. (To prevent this from happening, use `font-synthesis: none`, covered later in the chapter.)

Italic and oblique text at the same angle are not the same: italic is stylized and usually obsessively designed, and oblique is merely slanted. By default, if `oblique` is declared without an angle, an value of `14deg` is used.

When oblique is given an angle, such as `font-style: oblique 25deg`, the browser selects the face classified as oblique, if available. If one or more oblique faces are available in the chosen font family, the one most closely matching the specified angle by the `font-style` descriptor is chosen. If no

oblique faces are available, the browser may synthesize an oblique version of the font by slanting a normal face by the specified angle.

Unless further limited by the font or the descriptor, the oblique angle specified must be between `90deg` and `-90deg`, inclusive. If the given value is outside those limits, the declaration is ignored. Positive values are slanted toward the end (inline-end) of the line, while negative values are slanted towards the beginning (inline-start) of the line.

To visualize the difference between italic and oblique text, it's easiest to refer to Figure 10-14, which illustrates the differences.

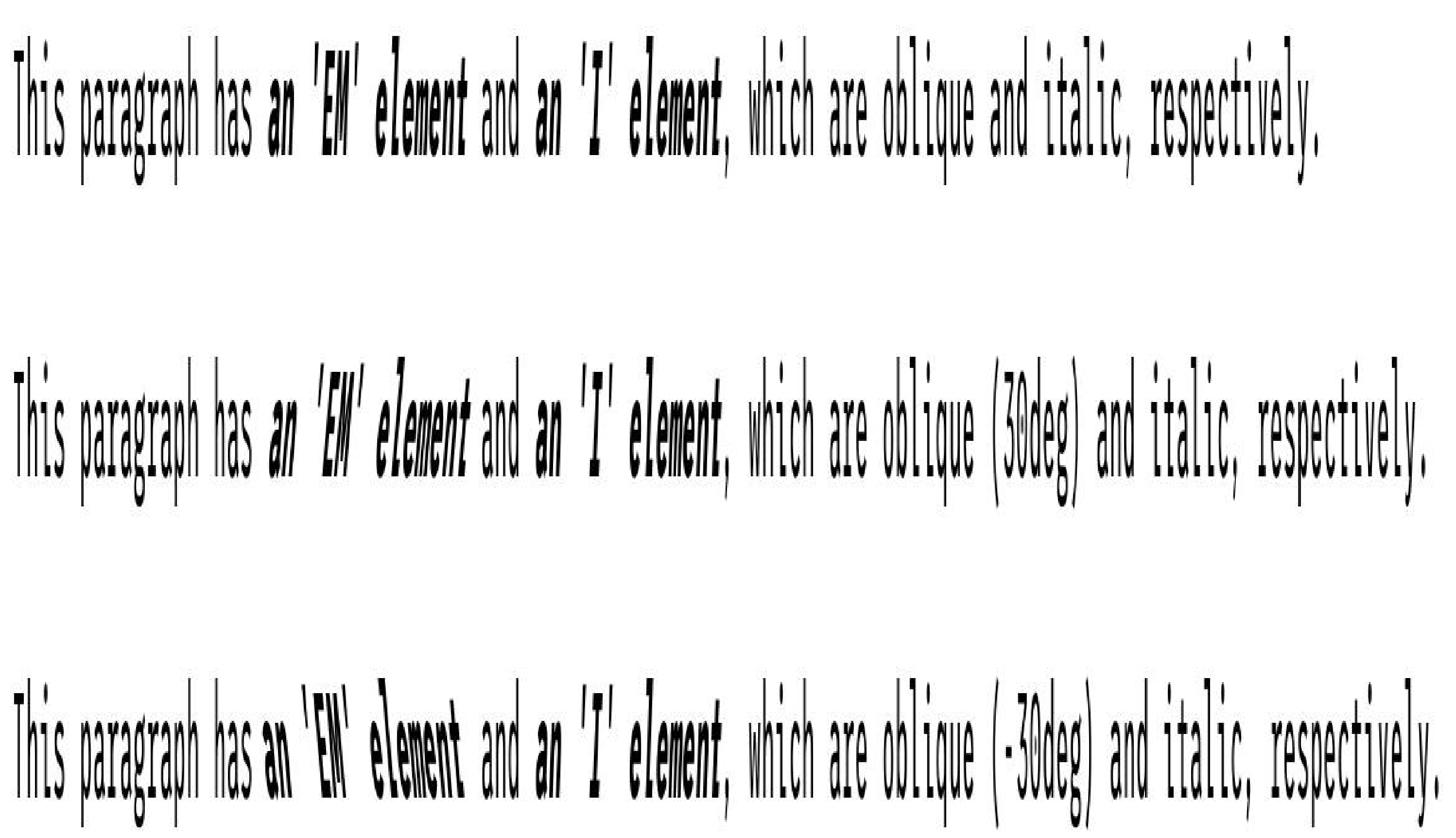


Figure 10-14. Italic and oblique text in detail

For TrueType or OpenType variable fonts, the "slnt" variation axis is used to implement varying slant angles for oblique, and the "ital" variation axis with a value of `1` is used to implement italic values. See

font-variation-settings, later in the chapter, for more details.

If you want to make sure that a document uses italic text in familiar ways, you could write a stylesheet like this:

```
p {font-style: normal;}
em, i {font-style: italic;}
```

These styles would make paragraphs use an upright font, as usual, and cause the em and i elements to use an italic font, also as usual. On the other hand, you might decide that there should be a subtle difference between em and i:

```
p {font-style: normal;}
em {font-style: oblique;}
i {font-style: italic;}
b {font-style: oblique -8deg;}
```

If you look closely at Figure 10-15, you'll see there is no apparent difference between the em and i elements. In practice, not every font is so sophisticated as to have both an italic face and an oblique face, and even fewer web browsers are sophisticated enough to tell the difference when both faces do exist.

This paragraph has a 'font-style' of 'normal', which is why it looks... normal. The exception are those elements which have been given a different style, such as *the 'EM' element* and *the 'I' element*, which get to be oblique and italic, respectively.

Figure 10-15. More font styles

The equivalent font-variation-settings setting for italic is "ital". For the oblique _<angle>_ value, the equivalent is "slnt", which is used to vary between upright and slanted text. Just like with font-style, the slant axis is interpreted as the angle of slant in counter-clockwise degrees from upright: inline-start-leaning oblique design will have a negative slant value, whereas inline-end-leaning needs a positive value.

The font-style descriptor

As a descriptor, font-style lets an author link specific faces to specific font-style values.

Values	normal | italic | oblique <angle>{0,2}
Initial value	auto

For example, we might want to assign very particular faces of Switzera to the various kinds of font-style property values. Given the following, the result would be to render h2 and h3 elements using "SwitzeraADF-Italic" instead of "SwitzeraADF-Regular," as illustrated in Figure 10-16:

```
@font-face {
```

```
    font-family: "Switzera";
    font-style: normal;
    src: url("SwitzeraADF-Regular.otf") format("opentype");
}
@font-face {
    font-family: "Switzera";
    font-style: italic;
    src: url("SwitzeraADF-Italic.otf") format("opentype");
}
@font-face {
    font-family: "Switzera";
    font-style: oblique;
    src: url("SwitzeraADF-Italic.otf") format("opentype");
}

h1, h2, h3 {font-family: SwitzeraADF, Helvetica, sans-serif;}
h1 {font-size: 225%;}
h2 {font-size: 180%; font-style: italic;}
h3 {font-size: 150%; font-style: oblique;}
```

A Level 1 Heading Element

A Level 2 Heading Element

A Level 3 Heading Element

Figure 10-16. Using declared font-style faces

Ideally, if there were a SwitzeraADF face with an oblique typeface, a page author could point to it instead of the italic variant. There isn't such a face, though, so the author mapped the italic face to both the `italic` and `oblique` values. As with `font-weight`, the `font-style` descriptor can take all of the values of the `font-style` property *except* for `inherit`.

Oblique text changes the angle of letterforms without performing any kind of character substitution. Any variable font that supports oblique text also supports normal or upright text: upright text is oblique text at a `0deg` angle. For example:

```
@font-face {
  font-family: "varFont";
  src: url("aVariableFont.woff2") format("woff2-variations");
  font-weight: 1 1000;
  font-stretch: 75% 100%;
  font-style: oblique 0deg 20deg;
  font-display: swap;
}

body { font-family: varFont, sans-serif; font-style: oblique 0deg; }
em { font-style: oblique 14deg; }
```

The angle given in the CSS value `oblique 3deg` is a clockwise slant of 3 degrees. Positive angles are clockwise slants, whereas negative angles are counter-clockwise slants. If no angle is included, it is the same as writing `oblique 14deg`. The degree angle can be any value between `-90deg` and `90deg`, inclusive.

Font Stretching

In some font families, there are a number of variant faces that have wider or narrower letterforms. These often take names like “Condensed,” “Wide,” “Ultra Expanded,” and so on. The utility of such variants is that a designer can use a single font family while also having skinny and fat variants. CSS provides a property that allows an author to select among such variants, when they exist, without having to explicitly define them in `font-family` declarations. It does this via the somewhat misleadingly named `font-stretch`.

FONT-STRETCH

Values	`normal` | `ultra-condensed` | `extra-condensed` | `condensed` | `semi-condensed` | `semi-expanded` | `expanded` | `extra-expanded` | `ultra-expanded` | *<percentage>*
Initial value	`normal`
Applies to	All elements
@font-face equivalent	`font-stretch`
Variable axis	`"wdth"`
Inherited	Yes
Animatable	Yes in a variable font that defines a stretch axis; otherwise no

You might expect from the property name that `font-stretch` will stretch or squeeze a font like saltwater taffy, but that’s actually not the case. This property instead behaves very much like the absolute-size keywords (e.g., `xx-large`) for the `font-size` property. You can set a percentage between 50%

and 200% inclusive, or use a range of keyword values that have defined percentage equivalents. Table 10-8 shows the mapping between keyword values and numeric percentages.

Table 10-8. percent equivalents for font-stretch keyword values

Keyword	Percentage
ultra-condensed	50%
extra-condensed	62.5%
condensed	75%
semi-condensed	87.5%
normal	100%
semi-expanded	112.5%
expanded	125%
extra-expanded	150%
ultra-expanded	200%

For example, an author might decide to stress the text in a strongly emphasized element by changing the font characters to a wider face than their parent element's font characters.

The catch is that this property only works if the font family in use actually *has* wider and narrower faces, which mostly only come with very expensive traditional fonts. (They're much more widely available in variable fonts.)

For example, consider the very common font Verdana, which has only one width face; this is equivalent to `font-stretch: normal`. Declaring the following will have no effect on the width of the displayed text:

```
body {font-family: Verdana;}
strong {font-stretch: extra-expanded;}
footer {font-stretch: extra-condensed;}
```

All of the text will be at Verdana's usual width. However, if the font family is changed to one that has a number of width faces, such as Futura, then things will be different, as shown in Figure 10-17:

```
body {font-family: Verdana;}
strong {font-stretch: extra-expanded;}
footer {font-stretch: extra-condensed;}
```

If there is one thing I can't **stress enough**, it's the value of Photoshop in producing books like this one.

Especially in footers.

Figure 10-17. Stretching font characters

For variable fonts that support the `"wdth"` axis, set the width in `font-variation-settings` to a value greater than 0. This controls the glyph width or stroke thickness, depending on the font design.

The font-stretch Descriptor

Much as with the `font-weight` descriptor, the `font-stretch` descriptor allows authors to explicitly assign faces of varying widths to the width values permitted in the `font-stretch` property. For example, the following rules explicitly assign three faces to the most directly analogous `font-stretch` values:

```
@font-face {
   font-family: "Switzera";
   font-stretch: normal;
   src: url("SwitzeraADF-Regular.otf") format("opentype");
}
@font-face {
   font-family: "Switzera";
   font-stretch: condensed;
   src: url("SwitzeraADF-Cond.otf") format("opentype");
}
@font-face {
   font-family: "Switzera";
   font-stretch: expanded;
   src: url("SwitzeraADF-Ext.otf") format("opentype");
}
```

In a parallel to what you saw in previous sections, you can call on these different width faces through the `font-stretch` property, as illustrated in Figure 10-18:

```
h1, h2, h3 {font-family: SwitzeraADF, Helvetica, sans-serif;}
h1 {font-size: 225%;}
h2 {font-size: 180%; font-stretch: condensed;}
h3 {font-size: 150%; font-stretch: expanded;}
```

A Level 1 Heading Element

A Level 2 Heading Element

A Level 3 Heading Element

Figure 10-18. Using declared font-stretch faces

If you use a variable font that contains the full spectrum of font stretch sizing, you can import a single font file with `@font-face`, then use it for all of your text font-stretch requirements. This produces the same degree of horizontal stretching as shown in Figure 10-18, albeit with a different font:

```
@font-face {
  font-family: 'League Mono Var';
  src: url('LeagueMonoVariable.woff2') format('woff2');
  font-weight: 100 900;
  font-stretch: 50% 200%;
  font-display: swap;
}

h1, h2, h3 {font-family: "League Mono Var", Helvetica, sans-serif;}
h2 {font-size: 180%; font-stretch: 75%;}
h3 {font-size: 150%; font-stretch: 125%;}
```

The `font-stretch` descriptor can take all of the values of the `font-stretch` property *except* for `inherit`.

If you do want to use a different font for your variable fonts depending on whether the text is extended or condensed, use the `"wdth"` value in the comma separated value of the `@font-face` `font-variation-settings` descriptor, as in the following example:

```
@font-face {
  font-family: 'League Mono Var';
  src: url('LeagueMonoVariable.woff2') format('woff2');
  font-weight: 100 900;
  font-stretch: 50% 200%;
}
strong {
  font-family: LeagueMono;
  font-variation-settings: "wdth" 100;
}
```

Font Synthesis

It is sometimes the case that a given font family will lack alternate faces for things like bold or italic text or small capital letters. In such situations, the user agent may attempt to synthesize a face from the faces it has available, but this can lead to unattractive letterforms. To address this, CSS offers `font-synthesis`, which lets authors say how much synthesis they will or won't permit in the rendering of a page. This doesn't have a `@font-face` descriptor, but it has bearing on all the font variants to follow, so we're dealing with it now.

FONT-SYNTHESIS

Values	`none` | `weight` | `style` | `small-caps`
Initial value	`weight style`
Applies to	All elements
Inherited	Yes
Animatable	No

In many user agents, a font family that has no bold face can have one computed for it. This might be done by adding pixels to either side of each character glyph, for example. While this might seem useful, it can lead to results that are visually unappealing, especially at smaller font sizes. This is why most font families actually have bold faces included: the font's designer wanted to make sure that bolded text in that font looked good.

Similarly, a font family that lacks an italic face can have one synthesized by simply slanting the characters in the normal face. This tends to look even worse than synthesized bold faces, particularly when it comes to serif fonts. Compare the difference between the actual italic face included in Georgia and a synthesized italic version of Georgia (which we're calling "oblique" here), illustrated in Figure 10-19.

italic text sample

oblique text sample

Figure 10-19. Synthesized versus designed italics

In supporting user agents, declaring `font-synthesis: none` blocks the user agent from doing any such synthesis for the affected elements. You can block it for the whole document with `html {font-synthesis: none;}`, for example. The downside is that any attempts to create variant text using a font that doesn't offer the appropriate faces will stay the normal face, instead of even approximating what was intended. The upside is that you don't have to worry about a user agent trying to synthesize those variants and doing a poor job of it.

Font Variants

Beyond font weights, font styles, and so forth, there are font variants. These are embedded within a font face and can cover things like various styles of historical ligatures, small-caps presentation, ways of presenting fractions, the spacing of numbers, whether zeroes get slashes through them, and much more. CSS lets authors invoke these variants, when they exist, through shorthand property `font-variant`.

FONT-VARIANT

Values	[*<font-variant-caps>* ‖ *<font-variant-numeric>* ‖ *<font-variant-alternates>* ‖ *<font-variant-ligatures>* ‖ *<font-variant-east-asian>*] | `normal` | `none`
Initial value	`normal`
Applies to	All elements
Computed value	As specified
@font-face equivalent	`font-variant`
Inherited	Yes
Animatable	No

This property is a shorthand for five separate properties, which we'll get to in just a moment. The most common values you'll find in the wild are the default of `normal`, which describes ordinary text, and `small-caps`, which is a value that's existed since CSS1.

First, however, let's cover the two values that's don't correspond to other properties.

`none`

Disables all variants of any kind by setting `font-feature-ligatures` to `none` and all the other font variant properties to `normal`.

`normal`

Disables most variants by setting all the font variant properties, including `font-feature-ligatures`, to `normal`.

Understanding the variant aspect of `small-caps` might help explain the idea of variants, making all the other properties easier to understand. `small-caps` call for the use of small-caps (`font-feature-settings: "smcp"`). Instead of upper- and lowercase letters, a small-caps font employs capital letters of different sizes. Thus, you might see something like that shown in Figure 10-20:

```
h1 {font-variant: small-caps;}
h1 code, p {font-variant: normal;}


<h1>The Uses of <code>font-variant</code></h1>
<p>
The property <code>font-variant</code> is very interesting...
</p>
```

THE USES OF font-variant

The property font-variant is very interesting. Given how common its use is in print media and the relative ease of its implementation, it should be supported by every CSS1-aware browser.

Figure 10-20. The small-caps value in use

As you may notice, in the display of the `h1` element, there is a larger capital letter wherever an uppercase letter appears in the source and a small capital letter wherever there is a lowercase letter in the source. This is very similar to `text-transform: uppercase`, with the only real difference being that, here, the capital letters are of different sizes. However, the reason that `small-caps` is declared using a font property is that some fonts have a specific small-caps face, which a font property is used to select.

What happens if no font face variant, such as small-caps, exists? There are two options provided in the specification. The first is for the user agent to create a small-caps face by scaling capital letters on its own. The second is to make all letters uppercase and the same size, exactly as if the declaration `text-transform: uppercase` had been used instead. This is not an ideal solution, but it is permitted.

> **WARNING**
>
> Bear in mind that not every font supports every variant. For example, most Latin fonts won't support any of the East Asian variants; for another, not every font will include support for, say, some of the numeric and ligature variants. Many fonts will support *none* of the variants.
>
> To find out what a given font supports, you have to consult its documentation, or else do a lot of testing if no documentation is available. Most commercial fonts do come with documentation, and most free fonts don't. Fortunately, some browser developer tools (not including Chromium browsers, as of late 2022) have a tab that provides information about font variants and feature settings.

Capital font variants

In addition to the `small-caps` value we just discussed, there are a number of other capital-text variants. These are addressed via the property `font-variant-caps`.

FONT-VARIANT-CAPS

Values	`normal` | `small-caps` | `all-small-caps` | `petite-caps` | `all-petite-caps` | `titling-caps` | `unicase`
Initial value	`normal`
Applies to	All elements
Computed value	specified keyword
@font-face equivalent	`font-variant`
Inherited	Yes
Animatable	No

The default value is `normal`, which means no capital-letter variant is used. From there, we have the following options:

`small-caps`

> Renders all of the letters using capital letters. The capital letters for characters that are uppercase in the source text are the same height as uppercase letters. Characters which are lowercase in the text are rendered as smaller capitals, usually a bit taller than the font's x-height.

`all-small-caps`

> The same as `small-caps`, except all letters are rendered as smaller capitals, even those that are uppercase in the source text.

`petite-caps`

> Similar to `small-caps`, except the capitals used for lowercase letters are equal in height to, or even a bit shorter than, the font's x-height. If there is no petite-caps variant for the font, the result is likely to be the same as for `small-caps`.

`all-petite-caps`

> The same as `petite-caps`, except all letters are rendered as smaller capitals, even those that are uppercase in the source text.

`titling-caps`

> In cases where there are multiple uppercase letters in a row, alternate capital forms are used to keep the letters from appearing too visually strong. Usually these are thinner versions of the normal capitals in the font.

unicase

The text is rendered using a mixture of capital and non-capital letterforms, usually all the same height. This can vary widely even among the few fonts that offer this variant.

The following code is illustrated in Figure 10-21. Note that the values marked with a dagger (†) were faked in one way or another.

```
.variant1 {font-variant-caps: small-caps;}
.variant2 {font-variant-caps: all-small-caps;}
.variant3 {font-variant-caps: petite-caps;}
.variant4 {font-variant-caps: all-petite-caps;}
.variant5 {font-variant-caps: titling-caps;}
.variant6 {font-variant-caps: unicase;}
```

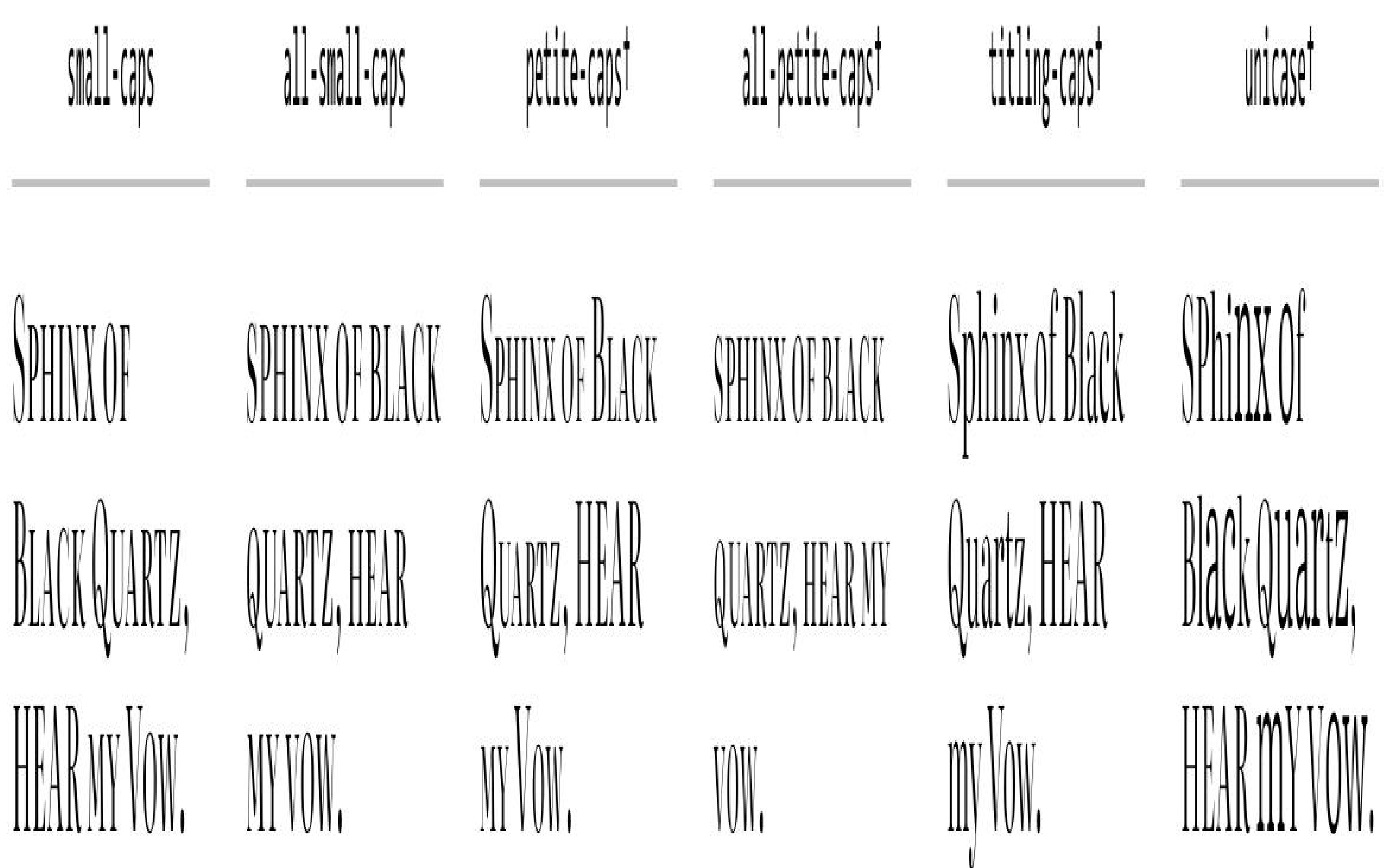


Figure 10-21. Different types of capital variants

Why were some of the examples in Figure 10-21 faked? In part, because finding a single font that contains all the capital variants is exceedingly difficult, and it was literally faster to fake some results than dig up a font, or set of fonts, that might work.

The other part was to highlight that exact situation: most of the time, you're going to get either a fallback (as from `petite-caps` to `small-caps`) or no variant at all. Because of this, make sure to use the @font-face `font-variant` descriptor to define what should happen. Otherwise, if a `font-variant-caps` category variant is not available, the browser will decide how to render it. For example, if `petite-caps` is specified and the font doesn't have a petite-caps face or variable axis defined, the user agent may render the text using small capital glyphs. If small capital glyphs are not included in the font, the browser may synthesize them by proportionally shrinking uppercase glyphs.

Alternatively, you can use `{font-synthesis: none;}` to prevent the browser from synthesizing the text. You can also include `{font-synthesis: small-caps;}`, or omit `font-synthesis` altogether, to allow small-caps typeface to be synthesized if needed.

Fonts sometimes include special glyphs for various caseless characters like punctuation marks to match the cap variant text. The browser will not synthesize caseless characters on its own.

All the values of `font-variant-caps` other than `normal` have defined equivalent OpenType features. These are summarized in Table 10-9.

Table 10-9. `font-variant-caps` values and equivalent OpenType features

Value	OpenType feature
`normal`	*n/a*
`small-caps`	`"smcp"`
`all-small-caps`	`"c2sc"`, `"smcp"`
`petite-caps`	`"pcap"`
`all-petite-caps`	`"c2pc"`, `"pcap"`
`titling-caps`	`"titl"`
`unicase`	`"unic"`

Numeric font variants

Many font faces have a variety of variant behaviors for use when rendering numerals. When available, these can be accessed via the `font-variant-numeric` property. The various values of this property affect the usage of alternate glyphs for numbers, fractions, and ordinal markers.

FONT-VARIANT-NUMERIC

Values	`normal` | `[ lining-nums` | `oldstyle-nums ]` ‖ `[ proportional-nums` | `tabular-nums ]` ‖ `[ diagonal-fractions` | `stacked-fractions ]` ‖ `ordinal` ‖ `slashed-zero ]`
Initial value	`normal`
Applies to	All elements
Computed value	specified keyword
Inherited	Yes
Animatable	No

The default value, `normal`, means that nothing special will be done when rendering numbers. They'll just appear the same as they usually do for the font face. All the values are demonstrated in Figure 10-22, and as before, the examples marked with a dagger (†) were faked in one way or another due to fonts lacking those features.

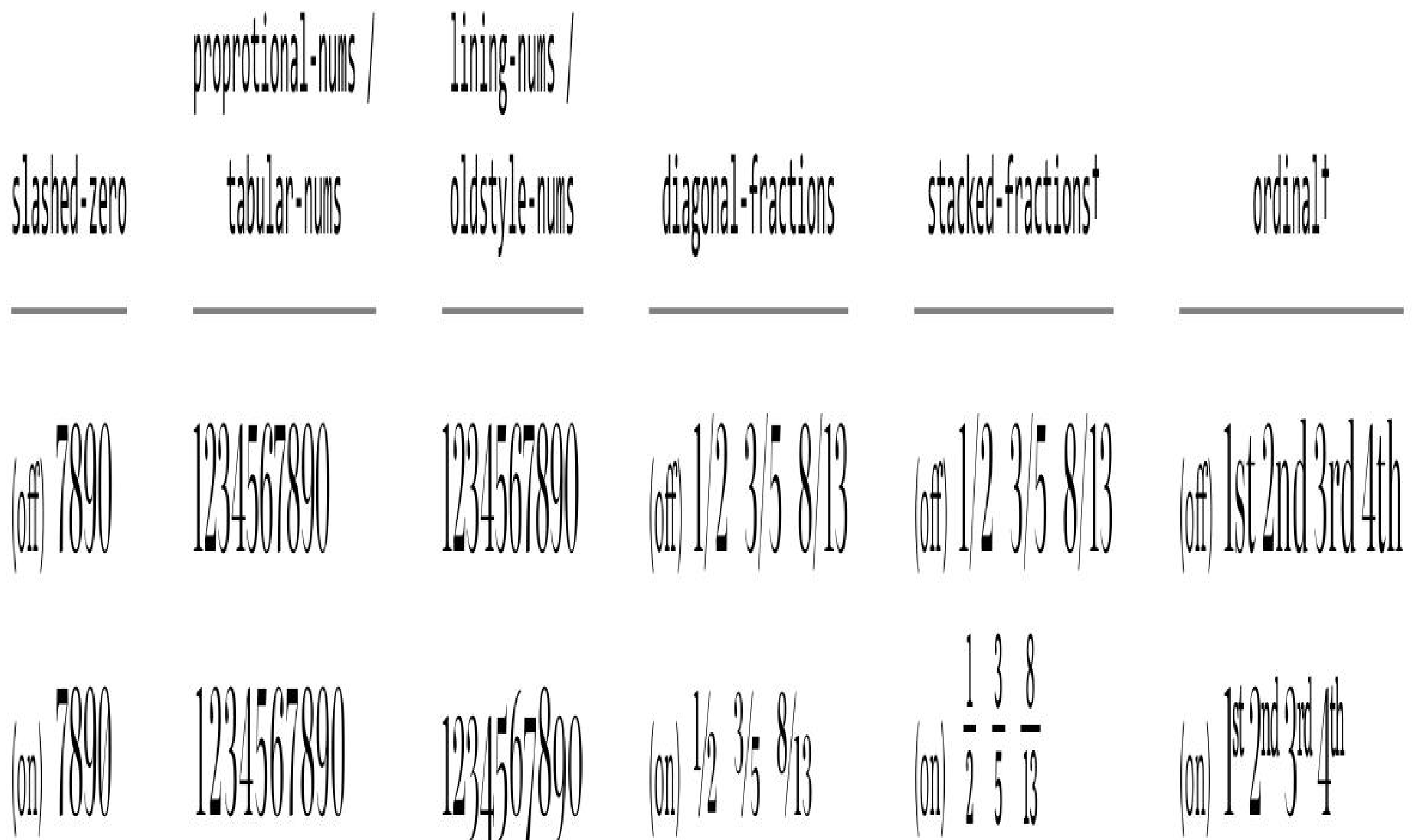


Figure 10-22. Different types of capital variants

Perhaps the simplest numeric variant is `slashed-zero`. This causes the numeral 0 to appear with a

slash through it, most likely on a diagonal. Slashed zeroes are often the default rendering in monospace fonts, where distinguishing zero from capital "O" can be difficult. In serif and sans-serif fonts, they are usually not the default appearance of zeroes. Setting `font-variant-numeric: slashed-zero` will bring out a slashed zero if one is available.

Speaking of diagonal slashes, the value `diagonal-fractions` causes characters arranged as a fraction (e.g., "1/2") to be rendered as smaller numbers, the first raised up, separated by a diagonal slash. `stacked-fractions` renders the fraction as the first number above the second, and the two separated by a horizontal slash.

If the font has features that turn ordinal labels, like the letters following the numbers of 1st, 2nd, 3rd, 4th in English, `ordinal` enables the use of those special glyphs. These will generally look like superscripted, smaller versions of the letters.

Authors can affect the figures used for numbers with `lining-nums`, which sets all numbers on the baseline; and `oldstyle-nums`, which enables numbers like 3, 4, 7, and 9 to descend below the baseline. Georgia is a common example of a font that has oldstyle numbers.

You can also influence the sizing of figures used for numbers. `proportional-nums` enables the numbers to be proportional, as in proportional fonts; and `tabular-nums` makes all numbers have the same width, as in monospace fonts. The advantage of these values is that you can, assuming there are glyphs to support them in the font face, get the monospace effect in proportional fonts without converting the numbers to a monospace face, and similarly cause monospace numbers to be sized proportionally.

You can include multiple values, but only one value from each of the numeric-value sets.

```
@font-face {
  font-family: 'mathVariableFont';
  src: local("math");
  font-feature-settings: "tnum" on, "zero" on;
}
.number {
  font-family: mathVariableFont, serif;
  font-feature-settings: "tnum" on, "zero" on;
  font-variant-numeric: ordinal slashed-zero oldstyle-nums stacked-fractions;
}
```

All the values of `font-variant-numeric` other than `normal` have defined equivalent OpenType features. These are summarized in Table 10-10.

Table 10-10. `font-variant-numeric` values and equivalent OpenType features

Value	OpenType feature
`normal`	*n/a*
`ordinal`	`"ordn"`
`slashed-zero`	`"zero"`
`lining-nums`	`"lnum"`
`oldstyle-nums`	`"onum"`
`proportional-nums`	`"pnum"`
`tabular-nums`	`"tnum"`
`diagonal-fractions`	`"frac"`
`stacked-fractions`	`"afrc"`

Ligature variants

A ligature is a joining of two (or more) characters into one shape. As an example, two lowercase “f” characters could have their cross-bars merged into a single line when the appear next to each other, or the crossbar could extend over a lowercase “i” and replace its usual dot in the sequence “fi”. More archaically, a combination like “st” could have a swash curve from one to the other. When available, these features can be enabled or disabled with the `font-variant-ligatures` property.

FONT-VARIANT-LIGATURES

Values	`normal` | `none` | `[[` `common-ligatures` | `no-common-ligatures` `]` ‖ `[` `discretionary-ligatures` | `no-discretionary-ligatures` `]` ‖ `[` `historical-ligatures` | `no-historical-ligatures` `]` ‖ `[` `contextual` | `no-contextual` `]]`
Initial value	`normal`
Applies to	All elements
Computed value	specified keyword
Inherited	Yes
Animatable	No

The values have the following effects:

common-ligatures

Enables the use of common ligatures, such as those combining “f” or “t” with letters that follow them. In French, the sequence “oe” is more usually rendered using the ligature “œ”. Browsers usually have these enabled by default, so if you want to disable them, use `no-common-ligatures` instead.

discretionary-ligatures

Enables the use of special ligatures created by font designers which are unusual or otherwise not regarded as common.

historical-ligatures

Enables the use of historical ligatures, which are generally those found in the typography of centuries past but are not used today. For example, the German the “tz” digraph used to be rendered as “”.

contextual-ligatures

Enables the use of ligatures that appear based on context, such as a cursive font enabling connecting curves from one letter to the next depending on not just the character that follows, but possibly also what characters came before. These are also sometimes used in programming fonts, where sequences like “!=” may be rendered as “≠” instead.

no-common-ligatures

Explicitly disables the use of common ligatures.

no-discretionary-ligatures

Explicitly disables the use of discretionary ligatures.

no-historical-ligatures

Explicitly disables the use of historical ligatures.

no-contextual-ligatures

Explicitly disables the use of contextual ligatures.

The default value, `normal`, turns off all these ligatures *except* common ligatures, which are enabled by default. This is especially relevant because `font-variant: normal` turns off all the `font-variant-ligatures` except the common ones, whereas `font-variant: none` turns them all off *including* common ligatures.

Table 10-11. `font-variant-ligatures` values and equivalent OpenType features

Value	OpenType feature
`common-ligatures`	`"clig" on`, `"liga" on`
`discretionary-ligatures`	`"dlig" on`
`historical-ligatures`	`"hlig" on`
`contextual-ligatures`	`"calt" on`
`no-common-ligatures`	`"clig" off`, `"liga" off`
`no-discretionary-ligatures`	`"dlig" off`
`no-historical-ligatures`	`"hlig" off`
`no-contextual-ligatures`	`"calt" off`

Less likely to be used or supported by browsers are the `font-variant-alternates` and `font-variant-east-asian` properties.

Alternate variants

For any given character, a font may include alternate glyphs in addition to the default glyph for that character. The `font-variant-alternates` property affects the usage of those alternate glyphs.

FONT-VARIANT-ALTERNATES

Values	`normal` | [`historical-forms` ‖ stylistic() ‖ historical-forms ‖ styleset() ‖ character-variant() ‖ swash() ‖ ornaments() ‖ annotation()]
Initial value	`normal`
Applies to	All elements
Computed value	as specified
Inherited	Yes
Animatable	Discrete

The default value, `normal`, means don't use any alternate variants. The `historical-forms` keyword enables historical forms; that is, glyphs that were common in the past but not today. All the other values are functions.

These alternate glyphs may be referenced by alternative names defined in `@font-feature-values`.

With `@font-feature-values`, you can define a common name for the `font-variant-alternates` function values to activate OpenType features.

The `@font-feature-values` at-rule may be used either at the top level of your CSS or inside any CSS conditional-group at-rule.

Table 10-12. `font-variant-alternates` values and equivalent OpenType features

Value	OpenType feature
`annotation()`	`"nalt"`
`character-variant()`	`"cvXY"`
`historical-forms`	`"hist"`
`ornaments()`	`"ornm"`
`styleset()`	`"ssXY"`
`stylistic()`	`"salt"`
`swash()`	`"swsh"`, `"cswh"`

In the Table 10-12, XY is replaced by a number representing the feature set. With OpenType fonts and `font-feature-settings`, some features are already defined. For example, the opentype equivalent of the `styleset()` function is `"ssXY"`. As of late 2022, ss01 through ss20 are currently defined. Values higher than 99 are allowed, but they don't map to any OpenType values and will be ignored.

There is also an at-rule version of `font-variant-alternates` called `@font-feature-values`, which allows authors to define labels for alternate values of `font-variant-alternates` using at-rules of their own. The following two styles (taken from the CSS specification) demonstrate how to label the numeric values of the `swash` alternate, and then use them later in `font-variant-alternates`:

```
@font-feature-values Noble Script { @swash { swishy: 1; flowing: 2; } }

p {
  font-family: Noble Script;
  font-variant-alternates: swash(flowing); /* use swash alternate #2 */
}
```

Without the presence of the `@font-feature-values` at-rule, the paragraph styles would have to say `font-variant-alternates: swash(2)` instead of using `flowing` for the value of the `swash` function.

WARNING

As of late 2022, while all browsers support `font-variant` and its associated subproperties, only Firefox and Safari have `font-variant-alternates` and `@font-feature-values` support. You can more reliably set these variants using the `font-feature-settings` property.

East Asian font variants

The values of the `font-variant-east-asian` property allow for controlling glyph substitution and sizing in East Asian text.

FONT-VARIANT-EAST-ASIAN

Values	`normal` | `[` `[` `jis78` | `jis83` | `jis90` | `jis04` | `simplified` | `traditional` `]` ‖ `[` `full-width` | `proportional-width` `]` ‖ `ruby` `]`
Initial value	`normal`
Applies to	All elements
Computed value	specified keyword
Inherited	Yes
Animatable	No

The assorted JIS variants reflect the glyph forms defined in different Japanese national standards. Fonts generally include glyphs defined by the most recent national standard. JIS values allow for the inclusion of older Japanese glyph variations when such variants are needed, such as when reproducing historical documents.

Similarly, the `simplified` and `traditional` values allow control over the glyph forms for characters which have been simplified over time but for which the older, traditional form is still used in some contexts.

The `ruby` value enables display of Ruby variant glyphs. Ruby text is generally smaller than the associated body text. This property value allows font designers to include glyphs better suited for smaller typography than scaled-down versions of the default glyphs would be. Only glyph selection is affected; there is no associated font scaling.

Font variant position

Compared to the previous variants, `font-variant-position` is fairly straightforward. It’s strange, then, that it’s so poorly supported.

FONT-VARIANT-POSITION

Values	`normal` | `sub` | `super`
Initial value	`normal`
Applies to	All elements
Computed value	specified keyword
Inherited	Yes
Animatable	No

This property can be used to enable specialized variant glyphs that are meant solely for superscripted and subscripted text. As it says in the CSS specification, these glyphs are:

> *…designed within the same em-box as default glyphs and are intended to be laid out on the same baseline as the default glyphs, with no resizing or repositioning of the baseline. They are explicitly designed to match the surrounding text and to be more readable without affecting the line height.*
>
> —https://www.w3.org/TR/css-fonts-4/#font-variant-position-prop

This is in contrast to what happens with super- and subscripted text in fonts that lack such alternates, which is usually just smaller text that's been shifted up or down from the baseline. This sort of synthesis of super- and subscripted text often leads to line-height increases, which variant glyphs are generally designed to prevent.

Font Feature Settings

Throughout this chapter we've discussed font features, but have yet to cover the `font-feature-settings` property or descriptor. In a manner similar to `font-variant`, `font-feature-settings` allows authors to exercise low-level control over which OpenType font features are available for use.

FONT-FEATURE-SETTINGS

Values	`normal` | *<feature-tag-value>*#
Initial value	`normal`

The `font-feature-settings` property controls advanced typographic features in OpenType fonts, as opposed to `font-variation-settings` property, which provides low-level control over variable font characteristics.

You can list one or more comma-separated OpenType features, as defined by the OpenType specification. For example, enabling common ligatures, small caps, and slashed zeroes would look something like this:

```
font-feature-settings: "liga" on, "smcp" on, "zero" on;
```

The exact format of a <*feature-tag-value*> value is:

<*feature-tag-value*>

<*string*> [<*integer*> | `on` | `off`]?

For many features, the only permitted integer values are `0` and `1`, which are equivalent to `off` and `on` (and vice versa). There are some features that allow a range of numbers, however, in which case values greater than 1 both enable the feature and define the feature's selection index. If a feature is listed but no number is provided, `1` (on) is assumed. Thus, the following descriptors are all equivalent:

```
font-feature-settings: "liga";     /* 1 is assumed */
font-feature-settings: "liga" 1;   /* 1 is declared */
font-feature-settings: "liga" on;  /* on = 1 */
```

Remember that all <*string*> values *must* be quoted. Thus, the first of the following descriptors will be recognized, but the second will be ignored:

```
font-feature-settings: "liga", dlig;
/* common ligatures are enabled; we wanted discretionary ligatures, but forgot
   quotes, so they are not enabled */
```

A further restriction is that OpenType requires that all feature tags be four ASCII characters long. Any feature name longer or shorter, or that uses non-ASCII characters, is invalid and will be ignored. (This isn't something you personally need to worry about unless you're using a font that has it own made-up feature names and the font's creator didn't follow the naming rules.)

By default, OpenType fonts *always* have the following features enabled unless the author explicitly disables them via `font-feature-settings` or `font-variant`:

`"calt"`

Contextual alternates

`"ccmp"`

Composed characters

`"clig"`

Contextual ligatures

`"liga"`

Standard ligatures

"locl"

Localized forms

"mark"

Mark to base positioning

"mkmk"

Mark to mark positioning

"rlig"

Required ligatures

Additionally, other features may be enabled by default in specific situations, such as vertical alternatives (`"vert"`) for vertical runs of text.

The OpenType `font-feature-setting` values we've discussed so far are all listed in this table of OpenType codes, along with a few others we didn't touch on for lack of support:

Table 10-13. OpenType values

Code	Meaning	Long hand
`"afrc"`	Alternative fractions	`stacked-fractions`
`"c2pc"`	Petite capitals	`petite-caps`
`"c2sc"`	Small capitals from capitals	`all-small-caps`
`"calt"`	Contextual alternates	`contextual`
`"case"`	Case-sensitive forms	
`"clig"`	Common ligatures	`common-ligatures`
`"cswh"`	Swash function	`swash()`
`"cv01"`	Character variants (01-99)	`character-variant()`
`"dnom"`	Denominators	
`"frac"`	Fractions	`diagonal-fractions`
`"fwid"`	Full width variants	`full-width`
`"hist"`	Enable historical forms	`historical-forms`
`"liga"`	Standard ligatures	`common-ligatures`
`"lnum"`	Lining figures	`lining-nums`
`"locl"`	Localized forms	

"numr"	Numerators	
"nalt"	Annotation function	annotation()
"onum"	Oldstyle figures	oldstyle-nums
"ordn"	Ordinal markers	ordinal
"ornm"	Ornaments (function)	ornaments()
"pcap"	Petite capitals	petite-caps
"pnum"	Proportional figures	
"pwid"	Proportionally-spaced variants	proportional-width
"ruby"	Ruby	ruby
"salt"	Stylistic function	stylistic()
"sinf"	Scientific inferiors	
"smcp"	Small capitals	small-caps
"smpl"	Simplified forms	simplified
"ss01"	Stylistic set 1 (numero correct)	styleset()
"ss07"	Stylistic set (1-20)	styleset()
"subs"	Subscript	
"sups"	Superscript	
"swsh"	Swash function	swash()
"titl"	Titling capitals	titling-caps
"tnum"	Tabular figures	tabular-nums
"trad"	Traditional forms	traditional
"unic"	Unicase	unicase
"zero"	Slashed zero	slashed-zero

The complete list of standard OpenType feature names can be found at microsoft.com/typography/otspec/featurelist.htm.

That said, `font-feature-settings` is a low-level feature designed to handle special cases where no other way exists to enable or access an OpenType font feature. It's also the case that you have to list all of the feature settings you want to use in a single property value. Whenever possible, use the `font-variant` shorthand property or one of the six associated longhand properties including `font-variant-ligatures`, `font-variant-caps`, `font-variant-east-asian`, `font-`

`variant-alternates`, and `font-variant-numeric`.

The font-feature-settings Descriptor

The `font-feature-settings` descriptor lets you decide which of an OpenType font face's settings can or cannot be used, specified as a space-separated list.

Now, hold up a second—isn't that almost exactly what we did with `font-variant` just a few paragraphs ago? Yes! The `font-variant` descriptor covers nearly everything `font-feature-settings` does, plus a little more besides. It just does so in a more CSS-like way, with value names instead of cryptic OpenType identifiers and Boolean toggles. Because of this, the CSS specification explicitly encourages authors to use `font-variant` instead of `font-feature-settings`, except in those cases where there's a font feature that the value list of `font-variant` doesn't include.

Keep in mind that this descriptor merely makes features available for use (or suppresses their use). It does not actually turn them on for the display of text; for that, see the section on the `font-feature-settings` property.

Just as with the `font-variant` descriptor, the `font-feature-settings` descriptor defines which font features are enabled (or disabled) for the font face being declared in the `@font-face` rule. For example, given the following, Switzera will have alternative fractions and small-caps disabled, even if such features exist in SwitzeraADF:

```
@font-face {
  font-family: "Switzera";
  font-weight: normal;
  src: url("SwitzeraADF-Regular.otf") format("opentype");
  font-feature-settings: "afrc" off, "smcp" off;
}
```

The `font-feature-settings` descriptor can take all of the values of the `font-feature-settings` property *except* for `inherit`.

Font-variation-settings

The `font-variation-settings` property provides low-level control over variable font characteristics, by specifying a four letter axis name along with a value.

Values	`normal` | [`<string>` `<number>`]#
Initial value	`normal`
Applies to	All elements
Computed value	as specified
Inherited	Yes
Animatable	Yes

There are five registered axes. We have covered almost all of them:

Table 10-14. Font variation axes

Axis	Property	Property value
`"wght"`	`font-weight`	`1` – `1000`
`"slnt"`	`font-style`	`oblique` / `oblique` *<angle>*
`"ital"`	`font-style`	`italic`
`"opsz"`	`font-optical-sizing`	
`"wdth"`	`font-stretch`	

We used the term “registered axes” because font developers are not limited to weight, width, optical size, slant and italics: They can create custom axes, and “register” them by giving them a four-letter label. The simplest way to know if a font has such axes is to look at the font’s documentation; otherwise, you have to know how to dig into the internals of a font’s file(s) to find out. These axes can control any aspect of the font’s appearance, such as the size of the dot on lowercase i and j. Creating custom axes is beyond the scope of this book, but calling on them where they exist is not.

Because these axes are string values, they have to be quoted and they are case sensitive, always lowercase. Imagine a font where the size of the dots (which are properly called *diacritic marks* or just *diacritics*) over lowercase i and j can be changed by way of an axis called `DCSZ` (for “diacritic size”). Furthermore, this axis has been defined by the font’s designer to allow values from 1 to 10. The diacritic size could be maximzed as follows:

```
p {font-family: DotFont, Helvetica, serif; font-variation-settings: "DCSZ" 10;}
```

The `font-variation-settings` descriptor is the same as the property. Instead of declaring each registered axis separately, they are declared on one line, comma separated.

```
@font-face {
```

```
    font-family: 'LeagueMono';
    src: url('LeagueMonoVariable.woff2') format('woff2');
    font-weight: 100 900;
    font-stretch: 50% 200%;
    font-variation-settings: 'wght' 100 900, 'wdth' 50 200;
    font-display: swap;
  }
```

> **TIP**
>
> Although you can set the weight, style, and so forth of a given font using `font-variation-settings`, it is recommended that you use the more widely-supported and human-readable properties `font-weight`, `font-style`, and so forth instead.

font-optical-sizing

Text rendered at different sizes often benefits from slightly different visual representations. For example, to aid reading at small text sizes, glyphs have less detail and strokes are often thicker with larger serifs. Larger text can have more features and a greater contrast between thicker and thinner strokes. The property `font-optical-sizing` allows authors to enable or disable this feature of variable fonts.

Values	`auto` | `none`
Initial value	`auto`
Applies to	All elements and text
Computed value	as specified
Varable font axis	`"opsz"`
Inherited	Yes
Animatable	Discrete

By default, `auto`, browsers can modify the shape of glyphs based on font-size and pixel density. The `none` value tells the browser to *not* do this.

> **TIP**
>
> In fonts that support it, optical sizing is usually defined as a range of numbers. If you want to explicitly change the optical sizing of a given element’s font to be a specific number, perhaps to make text sturdier or more delicate than it would be by default, use the `font-variation-settings` property and give it a value something like `'opsz' 10` (where `10` can be any number in the optical-sizing range).

Override descriptors

This brings us to the last three `@font-face` descriptors that we have yet to discuss. There are three descriptors that enable override setting for font families, `ascent-override`, `descent-override`, and `line-gap-override`, which define the ascent, descent, and line gap metrics, respectively. All three descriptors take the same values: `normal` or a `<percentage>`.

ascent-override, descent-override, line-gap-override descriptors

Values	`normal` | `<percentage>`
Initial value	`normal`

The goal of these descriptors is to help fallback fonts better match a primary font by overriding the metrics of the fallback font to better match those of the primary font.

The ascent metric is the distance above the baseline used to lay out line boxes; that is, the distance from the baseline to the top of the em box. The descent metric is the distance below the baseline used to lay out line boxes; that is, the distance from the baseline to the bottom of the em box. The line-gap metric is the font's recommended distance between adjacent lines of text, which is sometimes called *external leading*.

Here's an example of a hypothetical font and its ascent, descent, and line-gap override descriptors:

```
@font-face {
  font-family: "PreferredFont";
  src: url("PreferredFont.woff");
}

@font-face {
  font-family: FallbackFont;
  src: local(FallbackFont);
  ascent-override: 110%;
  descent-override: 95%;
  line-gap-override: 105%;
}
```

This will direct the browser to alter the ascent and descent heights by 110% and 95% respectively, and increase the line-gap to 105% the distance in the fallback font.

Font Kerning

A font property that doesn't have a descriptor equivalent is `font-kerning`. Some fonts contain data regarding how characters should be spaced relative to each other, known as *kerning*. Kerning can make character spacing more visually appealing and pleasant to read.

Kerning space varies depending on how characters are combined; for example, the character pair *oc* may

have a different spacing than the pair *ox*. Similarly, *AB* and *AW* may have different separation distances, to the point that in some fonts, the top-right tip of the *W* is actually placed to the left of the bottom-right tip of the *A*. This kerning data can be explicitly called for or suppressed using the property `font-kerning`.

FONT-KERNING

Values	`auto` | `normal` | `none`
Initial value	`auto`
Applies to	All elements
Inherited	Yes
Animatable	No

The value `none` is pretty simple: it tells the user agent to ignore any kerning information in the font. `normal` tells the user agent to kern the text normally; that is, according to the kerning data contained in the font. `auto` tells the user agent to do whatever it thinks best, possibly depending on the type of font in use. The OpenType specification, for example, recommends (but does not require) that kerning be applied whenever the font supports it. Furthermore, as per the CSS specification:

> *…[browsers] may synthetically support the kern feature with fonts that contain kerning data in the form of a `kern` table but lack kern feature support in the `GPOS` table.*
>
> —https://www.w3.org/TR/css-fonts-4/#font-kerning-prop

Which means, in effect, that if there is kerning information built into the font, browsers are allowed to enforce it even if the font lacks an explicit enabling of kerning via a feature table.

NOTE

Note that if the property `letter-spacing` (see Chapter 11) is applied to kerned text, the kerning is done first and *then* the letters' spacing is adjusted according to the value of `letter-spacing`, not the other way around.

The font Property

All of the properties discussed thus far are very sophisticated, but writing them all out could get a little tedious:

```
h1 {font-family: Verdana, Helvetica, Arial, sans-serif; font-size: 30px;
    font-weight: 900; font-style: italic; font-variant-caps: small-caps;}
h2 {font-family: Verdana, Helvetica, Arial, sans-serif; font-size: 24px;
    font-weight: bold; font-style: italic; font-variant-caps: normal;}
```

Some of this problem could be solved by grouping selectors, but wouldn't it be easier to combine

everything into a single property? Enter `font`, which is a shorthand property encompassing most (not quite all) the other font properties, and a little more besides.

FONT

Values	[[*<font-style>* ‖ [`normal` | `small-caps`] ‖ *<font-weight>* #&8214; *<font_stretch_css3>*]? *<font-size>* [/ *<line-height>*]? *<font-family>*] | `caption` | `icon` | `menu` | `message-box` | `small-caption` | `status-bar`
Initial value	Refer to individual properties
Applies to	All elements
Percentages	Calculated with respect to the parent element for *<font-size>* and with respect to the element's *<font-size>* for *<line-height>*
Computed value	See individual properties (`font-style`, etc.)
Inherited	Yes
Animatable	Refer to individual properties
Note	See the next section for an explanation of *<font_stretch_css3>*

Generally speaking, a `font` declaration can have any one value from each of the listed font properties, or else a system font value (described in "Using System Fonts"). Therefore, the preceding example could be shortened as follows (and have exactly the same effect, as illustrated by Figure 10-23):

```
h1 {font: italic 900 small-caps 30px Verdana, Helvetica, Arial, sans-serif;}
h2 {font: bold normal italic 24px Verdana, Helvetica, Arial, sans-serif;}
```

A LEVEL 1 HEADING ELEMENT

A Level 2 Heading Element

Figure 10-23. Typical font rules

We say that the styles "could be" shortened in this way because there are a few other possibilities, thanks to the relatively loose way in which `font` can be written. If you look closely at the preceding example, you'll see that the first three values don't occur in the same order. In the `h1` rule, the first three values are the values for `font-style`, `font-weight`, and `font-variant`, in that order. In the second, they're ordered `font-weight`, `font-variant`, and `font-style`. There is nothing wrong here because these three can be written in any order. Furthermore, if any of them has a value of `normal`, that can be left out altogether. Therefore, the following rules are equivalent to the previous example:

```
h1 {font: italic 900 small-caps 30px Verdana, Helvetica, Arial, sans-serif;}
```

```
h2 {font: bold italic 24px Verdana, Helvetica, Arial, sans-serif;}
```

In this example, the value of `normal` was left out of the `h2` rule, but the effect is exactly the same as in the preceding example.

It's important to realize, however, that this free-for-all situation applies only to the first three values of `font`. The last two are much stricter in their behavior. Not only must `font-size` and `font-family` appear in that order as the last two values in the declaration, but both must always be present in a `font` declaration. Period, end of story. If either is left out, then the entire rule will be invalidated and will be ignored completely by a user agent. Thus, the following rules will get you the result shown in Figure 10-24:

```
h1 {font: normal normal italic 30px sans-serif;}   /* no problem here */
h2 {font: 1.5em sans-serif;}   /* also fine; omitted values set to 'normal' */
h3 {font: sans-serif;}     /* INVALID--no 'font-size' provided */
h4 {font: lighter 14px;}   /* INVALID--no 'font-family' provided */
```

A Level 1 Heading Element

A Level 2 Heading Element

A Level 3 Heading Element

A Level 4 Heading Element

Figure 10-24. The necessity of both size and family

Font Property Limitations

Because the `font` property has been part of CSS since the very beginning, and because so many properties dealing with all the variants that came later, there are some limitations to the `font` property when it comes to font variations.

First, it's important to remember that when using the `font` shorthand property, the following properties are all set to their default values even though they cannot be represented in `font`:

- `font-feature-settings`
- `font-kerning`
- `font-language-override`
- `font-optical-sizing`
- `font-palette`
- `font-size-adjust`
- `font-variant-alternates`

- `font-variant-caps` (unless `small-caps` is included in the `font` value)
- `font-variant-east-asian`
- `font-variant-ligatures`
- `font-variant-numeric`
- `font-variation-settings`

Second, and following on the note in the previous list, there are only two variation values permitted: `small-caps` and `normal`. The numeric, ligature, alternate, East Asian, and many of the caps variants cannot be set via the `font` property. If you want, for example, to use small caps and slashed zeroes in your top-level headings, you would need to write something like:

```
h1 {font: bold small-caps 3em/1.1 Helvetica, sans-serif;
    font-variant-numeric: slashed-zero;
```

Third, another property value that suffers from the weight of history is font stretching. As we discussed earlier in the chapter, `font-stretch` allows authors to choose from a number of keywords or to set a percentage in the rage of 50% to 200% (inclusive). The keywords may be used in `font`, but the percentage value may not.

Adding the Line Height

It is also possible to set the value of the property `line-height` using `font`, despite the fact that `line-height` is a text property (not covered in this chapter), not a font property. It's done as a sort of addition to the `font-size` value, separated from it by a forward slash (`/`):

```
body {font-size: 12px;}
h2 {font: bold italic 200%/1.2 Verdana, Helvetica, Arial, sans-serif;}
```

These rules, demonstrated in Figure 10-25, set all `h2` elements to be bold and italic (using face for one of the sans-serif font families), set the `font-size` to `24px` (twice the `body`'s size), and set the `line-height` to `28.8px`.

Figure 10-25. Adding line height to the mix

This addition of a value for `line-height` is entirely optional, just as the first three `font` values are. If you do include a `line-height`, remember that the `font-size` always comes before `line-height`, never after, and the two are always separated by a slash.

WARNING

This may seem repetitive, but it's one of the most common errors made by CSS authors, so we can't say it enough: the required values for `font` are `font-size` and `font-family`, in that order. Everything else is strictly optional.

Using Shorthands Properly

It is important to remember that `font`, being a shorthand property, can act in unexpected ways if you are careless with its use. Consider the following rules, which are illustrated in Figure 10-26:

```
h1, h2, h3 {font: italic small-caps 250% sans-serif;}
h2 {font: 200% sans-serif;}
h3 {font-size: 150%;}

<h1>This is an h1 element</h1>
<h2>This is an h2 element</h2>
<h3>This is an h3 element</h3>
```

A LEVEL 1 HEADING ELEMENT

A Level 2 Heading Element

A LEVEL 3 HEADING ELEMENT

Figure 10-26. Shorthand changes

Did you notice that the `h2` element is neither italicized nor small-capped, and that none of the elements are bold? This is the correct behavior. When the shorthand property `font` is used, any omitted values are reset to their defaults. Thus, the previous example could be written as follows and still be exactly equivalent:

```
h1, h2, h3 {font: italic normal small-caps 250% sans-serif;}
h2 {font: normal normal normal 200% sans-serif;}
h3 {font-size: 150%;}
```

This sets the `h2` element's font style and variant to `normal`, and the `font-weight` of all three elements to `normal`. This is the expected behavior of shorthand properties. The `h3` does not suffer the same fate as the `h2` because you used the property `font-size`, which is not a shorthand property and therefore affects only its own value.

Using System Fonts

In situations where you want to make a web page blend in with the user's operating system, the system font values of `font` come in handy. These are used to take the font size, family, weight, style, and variant

of elements of the operating system, and apply them to an element. The values are as follows:

`caption`

Used for captioned controls, such as buttons

`icon`

Used to label icons

`menu`

Used in menus—that is, drop-down menus and menu lists

`message-box`

Used in dialog boxes

`small-caption`

Used for labeling small controls

`status-bar`

Used in window status bars

For example, you might want to set the font of a button to be the same as that of the buttons found in the operating system. For example:

```
button {font: caption;}
```

With these values, it is possible to create web-based applications that look very much like applications native to the user’s operating system.

Note that system fonts may only be set as a whole; that is, the font family, size, weight, style, etc., are all set together. Therefore, the button text from our previous example will look exactly the same as button text in the operating system, whether or not the size matches any of the content around the button. You can, however, alter the individual values once the system font has been set. Thus, the following rule will make sure the button’s font is the same size as its parent element’s font:

```
button {font: caption; font-size: 1em;}
```

If you call for a system font and no such font exists on the user’s machine, the user agent may try to find an approximation, such as reducing the size of the `caption` font to arrive at the `small-caption` font. If no such approximation is possible, then the user agent should use a default font of its own. If it can find a system font but can’t read all of its values, then it should use the default value. For example, a user agent may be able to find a `status-bar` font but not get any information about whether the font is small-caps. In that case, the user agent will use the value `normal` for the `small-caps` property.

Font Matching

As we've seen, CSS allows for the matching of font families, weights, and variants. This is all accomplished through font matching, which is a vaguely complicated procedure. Understanding it is important for authors who want to help user agents make good font selections when displaying their documents. We left it for the end of the chapter because it's not really necessary to understand how the font properties work, and some readers will probably want to skip this part. If you're still interested, here's how font matching works:

1. The user agent creates, or otherwise accesses, a database of font properties. This database lists the various CSS properties of all of the fonts to which the user agent has access. Typically, this will be all fonts installed on the machine, although there could be others (for example, the user agent could have its own built-in fonts). If the user agent encounters two identical fonts, it will just ignore one of them.
2. The user agent takes apart an element to which font properties have been applied and constructs a list of font properties necessary for the display of that element. Based on that list, the user agent makes an initial choice of a font family to use in displaying the element. If there is a complete match, then the user agent can use that font. Otherwise, it needs to do a little more work.
3. A font is first matched against the `font-stretch` property.
4. A font is next matched against the `font-style` property. The keyword `italic` is matched by any font that is labeled as either "italic" or "oblique." If neither is available, then the match fails.
5. The next match is to `font-weight`, which can never fail thanks to the way `font-weight` is handled in CSS (explained in the earlier section, "How Weights Work").
6. Then, `font-size` is tackled. This must be matched within a certain tolerance, but that tolerance is defined by the user agent. Thus, one user agent might allow matching within a 20 percent margin of error, whereas another might allow only 10 percent differences between the size specified and the size that is actually used.
7. If there was no font match in Step 2, the user agent looks for alternate fonts within the same font family. If it finds any, then it repeats Step 2 for that font.
8. Assuming a generic match has been found, but it doesn't contain everything needed to display a given element—the font is missing the copyright symbol, for instance—then the user agent goes back to Step 3, which entails a search for another alternate font and another trip through Step 2.
9. Finally, if no match has been made and all alternate fonts have been tried, then the user agent selects the default font for the given generic font family and does the best it can to display the element correctly.

Furthermore, the user agent does the following to resolve handling of font variants and features:

1. First, check for font features enabled by default, including features required for a given script. The core set of default-enabled features is `"calt"`, `"ccmp"`, `"clig"`, `"liga"`, `"locl"`, `"mark"`,

"mkmk", and "rlig".

2. Then, if the font is defined via an `@font-face` rule, check for the features implied by the `font-variant` descriptor in the `@font-face` rule. Then check for the font features implied by the `font-feature-settings` descriptor in the `@font-face` rule.
3. Then check feature settings determined by properties other than `font-variant` or `font-feature-settings`. (For example, setting a non-default value for the `letter-spacing` property will disable ligatures.)
4. Then check for features implied by the value of the `font-variant` property, the related `font-variant` subproperties (e.g., `font-variant-ligatures`), and any other property that may call for the use of OpenType features (e.g., `font-kerning`).
5. Finally, check for the features implied by the value of `font-feature-settings` property.

The whole process is long and tedious, but it helps to understand how user agents pick the fonts they do. For example, you might specify the use of Times or any other serif font in a document:

```
body {font-family: Times, serif;}
```

For each element, the user agent should examine the characters in that element and determine whether Times can provide characters to match. In most cases, it can do so with no problem. Assume, however, that a Chinese character has been placed in the middle of a paragraph. Times has nothing that can match this character, so the user agent has to work around the character or look for another font that can fulfill the needs of displaying that element. Any Western font is highly unlikely to contain Chinese characters, but should one exist (let's call it AsiaTimes), the user agent could use it in the display of that one element—or simply for the single character. Thus, the whole paragraph might be displayed using AsiaTimes, or everything in the paragraph might be in Times except for the single Chinese character, which is displayed in AsiaTimes.

Summary

From what was initially a very simplistic set of font properties, CSS has grown to allow fine-grained and wide-ranging influence over how fonts are displayed on the web. From custom fonts downloaded over the web to custom-built families assembled out of a variety of individual faces, authors may be fairly said to overflow with font power.

The typographic options available to authors today are far stronger than ever, but always remember: you must use this power wisely. While you can have 17 different fonts in use on your site, that definitely doesn't mean that you should. Quite aside from the aesthetic difficulties this could present for your users, it would also make the total page weight much, much higher than it needs to be. As with any other aspect of web design, authors are advised to use their power wisely, not wildly.

Chapter 11. Text Properties

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 14th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at rfernando@oreilly.com.

Because text is so important, there are many CSS properties that affect it in one way or another. But didn't we just cover that in Chapter 10? Not exactly: we only covered fonts — the importing and usage of type faces. Text styles are different.

Okay, so what is the difference between text and fonts? At the simplest level, text is the content, and fonts are used to display that content. Fonts provide the shape for the letters. Text is the styling around those shapes. Using text properties, you can affect the position of text in relation to the rest of the line, superscript it, underline it, and change the capitalization. You can affect the size, color, and placement of text decorations.

Indentation and Inline Alignment

Let's start with a discussion of how you can affect the inline positioning of text within a line. Think of these basic actions as the same types of steps you might take to create a newsletter or write a report.

Originally, CSS was written on concepts of "horizontal" and "vertical." To better support all languages and writing directions, CSS now uses the terms "block direction" and "inline direction." If your primary language is Western-derived, then you're accustomed to a block direction of top to bottom, and an inline direction of left to right.

The *block direction* is the direction in which block elements are placed by default in the current writing mode. In English, for example, the block direction is top to bottom, or vertical, as one paragraph (or other text element) is placed beneath the one before. Some languages have vertical text, like Mongolian. When text is vertical, the block direction is horizontal.

The *inline direction* is the direction in which inline elements are written within a block. To again take English as an example, the inline direction is left to right, or horizontal. In languages like Arabic and Hebrew, the inline direction is right to left instead. To re-use the example from the last paragraph, Mongolian's inline direction is top to bottom.

Let's reconsider English for a moment. A plain page of English text, displayed on a screen, has a vertical block direction (from top to bottom) and a horizontal inline direction (from left to right). But if the page is rotated 90 degrees anticlockwise using CSS Transforms, then suddenly the block direction is horizontal and the inline direction is vertical. (And bottom to top, at that.)

> **TIP**
>
> You can still find a lot of English-centric blog posts and other CSS-related documentation on the Web using the terms "vertical" and "horizontal" when talking about writing directions. When you do, mentally translate them to "block" and "inline" as needed.

Indenting Text

Most paper books we read in Western languages format paragraphs of text with the first line indented, and no blank line between paragraphs. If you want to recreate that look, CSS provides the property `text-indent`.

TEXT-INDENT

Values	[*<length>* | *<percentage>*] && hanging && each-line
Initial value	`0`
Applies to	Block-level elements
Percentages	Refer to the width of the containing block
Computed value	For percentage values, as specified; for length values, the absolute length
Inherited	Yes
Animatable	Yes
Notes	`hanging` and `each-line` were still experimental as of mid-2022

Using `text-indent`, the first line of any element can be indented by a given length, even if that length is negative. A common use for this property is to indent the first line of a paragraph:

```
p {text-indent: 3em;}
```

This rule will cause the first line of any paragraph to be indented three ems, as shown in Figure 11-1.

This is a paragraph element, which means that the first line will be indented by 3em (i.e., three times the computed `font-size` of the text in the paragraph). The other lines in the paragraph will not be indented, no matter how long the paragraph may be.

Figure 11-1. Text indenting

In general, you can apply `text-indent` to any element that generates a block box, and the indentation will occur along the inline direction. You can't apply it to inline elements or on replaced elements such as images. However, if you have an image within the first line of a block-level element, it will be shifted over with the rest of the text in the line.

> **NOTE**
>
> If you want to "indent" the first line of an inline element, you can create the effect with left padding or margin.

You can also set negative values for `text-indent` to create a *hanging indent*, where the first line hangs out to one side of the rest of the element:

```
p {text-indent: −4em;}
```

Be careful when setting a negative value for `text-indent` ; the first few words may be chopped off by the edge of the browser window if you aren't careful. To avoid display problems, we recommend you use a margin or padding to accommodate the negative indentation:

```
p {text-indent: −4em; padding-left: 4em;}
```

Any unit of length, including percentage values, may be used with `text-indent`. In the following case, the percentage refers to the width of the parent element of the element being indented. In other words, if you set the indent value to `10%`, the first line of an affected element will be indented by 10 percent of its parent element's width, as shown in Figure 11-2:

```
div {width: 400px;}
p {text-indent: 10%;}

<div>
<p>This paragraph is contained inside a DIV, which is 400px wide, so the
first line of the paragraph is indented 40px (400 * 10% = 40).  This is
because percentages are computed with respect to the width of the element.</p>
</div>
```

This paragraph is contained inside a DIV, which is 400px wide, so the first line of the paragraph is indented 40px (400 * 10% = 40). This is because percentages are computed with respect to the width of the element.

Figure 11-2. Text indenting with percentages

Note that because `text-indent` is inherited, some browsers, like the Yandex browser, inherit the computed values, while Safari, Firefox, Edge, and Chrome inherit the declared value. In the following, both bits of text will be indented 5em in Yandex and 10% of the current element's width in other browsers, because the value of `5em` is inherited by the paragraph from its parent `<div>` in Yandex and older versions of WebKit, whereas most evergreen browsers inherit the declared value of 10%.

```
div#outer {width: 50em;}
div#inner {text-indent: 10%;}
p {width: 20em;}

<div id="outer">
<div id="inner">
This first line of the DIV is indented by 5em.
<p>
This paragraph is 20em wide, and the first line of the paragraph
is indented 5em in Webkit and 2em elsewhere.  This is because
computed values for 'text-indent' are inherited in Webkit,
while the declared values are inherited elsewhere.
</p>
</div>
</div>
```

Experimental values

As of late 2022, there were two keywords being considered for addition to `text-indent`. They were:

`hanging`

Inverts the indentation effect; that is, `text-indent: 3em hanging` would indent all the lines of text *except* the first line of text. This is similar to the negative-value indentation discussed previously, but without risking cutting off text, because instead of pulling the first line out of the content box, all the lines but the first are indented away from the edge of the content box.

`each-line`

Indents the first line of the element plus any line that starts after a forced line break, such as that caused by a `<br>`, but not lines that follow a soft line break.

When supported, either keyword can be used in conjunction with a length or percentage, such as:

```
p {text-indent: 10% hanging;}
pre {text-indent: 5ch each-line;}
```

Text Alignment

Even more basic than `text-indent` is the property `text-align`, which affects how the lines of text in an element are aligned with respect to one another.

TEXT-ALIGN

Values	`start` | `end` | `left` | `right` | `center` | `justify` | `justify-all` | `match-parent`
Initial value	`start`
Applies to	Block-level elements
Computed value	As specified, except in the case of `match-parent`
Inherited	Yes
Animatable	No
Note	`justify-all` is not supported as of mid-2022

The quickest way to understand how these values work is to examine Figure 11-3, which demonstrates the most widely-used values. The values `left`, `right`, and `center` cause the text within elements to be aligned exactly as described by these words in horizontal languages like English or Arabic, regardless of the language's inline direction.

This paragraph is styled `text-align: left;`, which causes the line boxes within the element to line up along the left inner content edge of the paragraph.

This paragraph is styled `text-align: right;`, which causes the line boxes within the element to line up along the right inner content edge of the paragraph.

This paragraph is styled `text-align: center;`, which causes the line boxes within the element to line up their centers with the center of the content area of the paragraph.

Figure 11-3. Selected behaviors of the text-align property

The default value of `text-align` is `start`, which is the equivalent of `left` in left-to-right languages, and `right` in right-to-left languages. In vertical languages. `left` and `right` are mapped to the start or end edge, respectively. This is illustrated in Figure 11-4.

このテキストは、他の手段で。整列しました

このテキストは、他の手段で整列しました。

このテキストは、他の手段で。整列しました

left

right

center

Figure 11-4. Left, right, and center in vertical writing modes

Because `text-align` applies only to block-level elements, such as paragraphs, there's no way to center an anchor within its line without aligning the rest of the line (nor would you want to, since that would likely cause text overlap).

As you may expect, `center` causes each line of text to be centered within the element. If you've ever come across the long-ago deprecated `<CENTER>` element, you may be tempted to believe that `text-align: center` is the same. It is actually quite different. `<CENTER>` affected not only text, but also centered whole elements, such as tables. `text-align` does not control the alignment of elements, only

their inline content.

Start and end alignment

Remembering that CSS was written on concepts of “horizontal” and “vertical”, the initial default value was originally “a nameless value that acts as *left* if *direction* is *ltr*, *right* if *direction* is *rtl* “. The default value now has a name: `start`, which is the equivalent of `left` in left-to-right (LTR) languages, and `right` in right-to-left (RTL) languages.

The default value of `start` means that text is aligned to the start edge of its line box. In left-to-right languages like English, that’s the left edge; in right-to-left languages such as Arabic, it’s the right edge. In vertical languages, it will be the top or bottom, depending on the writing direction. The upshot is that the default value is much more aware of the document’s language direction while leaving the default behavior the same in the vast majority of existing cases.

In a like manner, `end` aligns text with the end edge of each line box—the right edge in LTR languages, the left edge in RTL languages, and so forth. The effects of these values are shown in Figure 11-5.

This paragraph is start-aligned, which causes the line boxes within the element to line up along the start edge of the paragraph.

هذه الفقرة هي بداية الانحياز، الذي يتسبب في صناديق خط داخل عنصر ليصطف على طول حافة بداية الفقرة.

This paragraph is end-aligned, which causes the line boxes within the element to line up along the end edge of the paragraph.

פסקה זו היא הסוף מזדהה, מה שגורם את תיבות השורה בתוך הרכיב בשורה לאורך קצה סוף ההפסקה.

Figure 11-5. Start and end alignment

Justified text

An often-overlooked alignment value is `justify`, which raises some issues of its own. In justified text, both ends of a line of text (except the last line, which can be set with `text-align-last`) are placed at the inner edges of the parent element, as shown in Figure 11-6. Then, the spacing between words and letters is adjusted so that the words are distributed evenly throughout the line. Justified text is common in the print world (for example, in this book), but under CSS, a few extra considerations come into play.

This paragraph is styled `text-align: justify;`, which causes the line boxes within the element to align their left and right edges to the left and right inner content edges of the paragraph. The exception is the last line box, whose right edge does not align with the right content edge of the paragraph. (In right-to-left languages, the left edge of the last line box would not be so aligned.)

This paragraph is styled `text-align: justify;`, which causes the line boxes within the element to align their left and right edges to the left and right inner content edges of the paragraph. The exception is the last line box, whose right edge does not align with the right content edge of the paragraph. (In right-to-left languages, the left edge of the last line box would not be so aligned.)

Figure 11-6. Justified text

The user agent determines how justified text should be stretched or distributed to fill the space between the left and right edges of the parent. Some browsers, for example, might add extra space only between words, while others might distribute the extra space between letters (although the CSS specification states that “user agents may not further increase or decrease the inter-character space” if the property `letter-`

`spacing` has been assigned a length value). Other user agents may reduce space on some lines, thus mashing the text together a bit more than usual.

There is also the value `justify-all`, which sets full justification for both `text-align` and `text-align-last` (covered in an upcoming section).

> **WARNING**
>
> As of mid-2022, the `justify-all` value was not supported by any browser, even though nearly all of them support `text-align: justify` and `text-align-last: justify`. This gap in support remains a mystery as of press time, but is solved in most browsers with:
>
> ```
> .justify-all {
>   text-align: justify;
>   text-align-last: justify;
>   }
> ```

Parent matching

There's one more value to be covered, which is `match-parent`. If you declare `text-align: match-parent`, and the inherited value of `text-align` is `start` or `end`, then the alignment of the `match-parent` element will be calculated with respect to the parent element's horizontal or vertical, rather than inline, direction.

For example, you could force any English element's text alignment to match the alignment of a parent element, regardless of its writing direction, as in the following example.

```
div {text-align: start;}
div:lang(en) {direction: ltr;}
div:lang(ar) {direction: rtl;}
p {text-align: match-parent;}

<div lang="en-US">
Here is some en-US text.
<p>The alignment of this paragraph will be to the left, as with its parent.</p>
</div>
<div lang="ar">
□□□□ □□ □□□.
<p>The alignment of this paragraph will be to the right, as with its parent.</p>
</div>
```

Aligning the Last Line

There may be times when you want to align the text in the very last line of an element differently than you did the rest of the content. For example, with `text-align: justify` the last line defaults to `text-align: start`. You might ensure a left-aligned last line in an otherwise fully justified block of text, or choose to swap from left to center alignment. For those situations, there is `text-align-last`.

TEXT-ALIGN-LAST

Values	`auto` | `start` | `end` | `left` | `right` | `center` | `justify`
Initial value	`auto`
Applies to	Block-level elements
Computed value	As specified
Inherited	Yes
Animatable	No

As with `text-align`, the quickest way to understand how these values work is to examine Figure 11-7.

Figure 11-7. Differently aligned last lines

As the figure shows, the last lines of the elements are aligned independently of the rest of the elements, according to the elements' `text-align-last` values.

A close study of Figure 11-7 will reveal that there's more at play than just the last lines of block-level elements. In fact, `text-align-last` applies to any line of text that immediately precedes a forced line break, whether or not said line break is triggered by the end of an element. Thus, a line break created

by a `<br>` tag will make the line of text immediately before that break use the value of `text-align-last`.

There's an interesting wrinkle in `text-align-last`: if the first line of text in an element is also the last line of text in the element, then the value of `text-align-last` takes precedence over the value of `text-align`. Thus, the following styles will result in a centered paragraph, not a start-aligned paragraph:

```
p {text-align: start; text-align-last: center;}

<p>A paragraph.</p>
```

Word Spacing

The `word-spacing` property is used to modify inter-word spacing, accepting a positive or negative length. This length is *added* to the standard space between words. Therefore, the default value of `normal` is the same as setting a value of zero (`0`).

WORD-SPACING

Values	*<length>* | `normal`
Initial value	`normal`
Applies to	All elements
Computed value	For `normal`, the absolute length `0` ; otherwise, the absolute length
Inherited	Yes
Animatable	Yes

If you supply a positive length value, then the space between words will increase. Setting a negative value for `word-spacing` brings words closer together:

```
p.spread {word-spacing: 0.5em;}
p.tight {word-spacing: -0.5em;}
p.default {word-spacing: normal;}
p.zero {word-spacing: 0;}

<p class="spread">The spaces—as in those between the “words”—in this paragraph
   will be increased by 0.5em.</p>
<p class="tight">The spaces—as in those between the “words”—in this paragraph
   will be increased by 0.5em.</p>
<p class="default">The spaces—as in those between the “words”—in this paragraph
   will be neither increased nor decreased.</p>
<p class="zero">The spaces—as in those between the “words”—in this paragraph
   will be neither increased nor decreased.</p>
```

Manipulating these settings has the effect shown in Figure 11-8.

The spaces—as in those between the “words”—in this paragraph will be increased by 0.5em.

The spaces—as in those between the “words”—in this paragraph will be increased by 0.5em.

The spaces—as in those between the “words”—in this paragraph will be neither increased nor decreased.

The spaces—as in those between the “words”—in this paragraph will be neither increased nor decreased.

Figure 11-8. Changing the space between words

In CSS terms, a “word” is any string of non-whitespace characters that is surrounded by whitespace of some kind. This means `word-spacing` is unlikely to work in any languages that employ pictographs, or non-Roman writing styles. This is also why the em dashes in the previous example’s text don’t get space around them. From the CSS point of view, “spaces—as” is a single word.

Use caution. `word-spacing` allows you to create very unreadable documents, as Figure 11-9 illustrates.

The spaces between words in
this paragraph will be increased
by one inch. Room enough
for ya?

Figure 11-9. Really wide word spacing

Letter Spacing

Many of the issues you encounter with `word-spacing` also occur with `letter-spacing`. The only real difference between the two is that `letter-spacing` modifies the space between characters or letters.

LETTER-SPACING

Values	*<length>* | `normal`
Initial value	`normal`
Applies to	All elements
Computed value	For length values, the absolute length; otherwise, `normal`
Inherited	Yes
Animatable	Yes

As with the `word-spacing` property, the permitted values of `letter-spacing` include any length, though character-relative lengths like `em` (rather than root-relative lengths like `rem`) are recommended to ensure the spacing is proportional to the font size.

The default keyword is `normal`, which has the same effect as `letter-spacing: 0`. Any length value you enter will increase or decrease the space between letters by that amount. Figure 11-10 shows the results of the following markup:

```
p {letter-spacing: 0;}    /*  identical to 'normal'  */
p.spacious {letter-spacing: 0.25em;}
p.tight {letter-spacing: -0.25em;}

<p>The letters in this paragraph are spaced as normal.</p>
<p class="spacious">The letters in this paragraph are spread out a bit.</p>
<p class="tight">The letters in this paragraph are a bit smashed together.</p>
```

The letters in this paragraph are spaced as normal.

The letters in this paragraph are spread out a bit.

Figure 11-10. Various kinds of letter spacing

WARNING

If a page uses fonts with features like ligatures, and those features are enabled, then altering letter or word spacing can effectively disable them. That is to say, browsers will not recalculate ligatures or other joins when letter spacing is altered.

Spacing and Alignment

It's important to remember that space between words may be altered by the value of the property `text-align`. If an element is justified, the spaces between letters and words may be altered to fit the text along

the full width of the line. This may in turn alter the spacing declared using `word-spacing`.

If a length value is assigned to `letter-spacing`, then it cannot be changed by `text-align`; but, if the value of `letter-spacing` is `normal`, then inter-character spacing may be changed to justify the text. CSS does not specify how the spacing should be calculated, so user agents use their own algorithms. To prevent `text-align` from altering letter spacing while keeping the default letter-spacing, declare `letter-spacing: 0`

Note that computed values are inherited, so child elements with larger or smaller text will have the same word or letter spacing as their parent element. You cannot define a scaling factor for `word-spacing` or `letter-spacing` to be inherited in place of the computed value (in contrast with `line-height`). As a result, you may run into problems such as those shown in Figure 11-11:

```
p {letter-spacing: 0.25em; font-size: 20px;}
small {font-size: 50%;}
```

```
<p>This spacious paragraph features <small>tiny text that is just
as spacious</small>, even though the author probably wanted the
spacing to be in proportion to the size of the text.</p>
```

Figure 11-11. Inherited letter spacing

As `inherit` inherits the ancestor’s letter-spacing computed length, the only way to achieve letter spacing that’s in proportion to the size of the text is to set it explicitly on each element, as follows:

```
p {letter-spacing: 0.25em;}
small {font-size: 50%; letter-spacing: 0.25em;}
```

And the same goes for word spacing.

Vertical Alignment

Now that we’ve covered alignment along the inline direction, let’s move on to the vertical alignment of inline elements along the block direction—things like superscripting and “vertical alignment.” (Vertical with respect to the line of text, if the text is laid out horizontally.) Since the construction of lines is a very complex topic that merits its own small book, we’ll just stick to a quick overview here.

The Height of Lines

The distance between lines can be affected by changing the “height” of a line. Note that “height” here is

with respect to the line of text itself, assuming that the longer axis of a line is "width," even if it's written vertically. The property names we cover from here will reveal a strong bias toward Western languages and their writing directions; this is an artifact of the early days of CSS, when Western languages were the only ones that could be easily represented.

The `line-height` property refers to the distance between the baselines of lines of text rather than the size of the font, and it determines the amount by which the height of each element's box is increased or decreased. In the most basic cases, specifying `line-height` is a way to increase (or decrease) the vertical space between lines of text, but this is a misleadingly simple way of looking at how `line-height` works. `line-height` controls the *leading*, which is the extra space between lines of text above and beyond the font's size. In other words, the difference between the value of `line-height` and the size of the font is the leading.

LINE-HEIGHT

Values	<*number*> | <*length*> | <*percentage*> | `normal`
Initial value	`normal`
Applies to	All elements (but see text regarding replaced and block-level elements)
Percentages	Relative to the font size of the element
Computed value	For length and percentage values, the absolute value; otherwise, as specified
Inherited	Yes
Animatable	Yes

When applied to a block-level element, `line-height` defines the *minimum* distance between text baselines within that element. Note that it defines a minimum, not an absolute value. Baselines of text can wind up being pushed further apart than the value of `line-height`, for example, if a line contains an inline image or form control that is taller than the declared line height. +line-height` does not affect layout for replaced elements like images, but it still applies to them.

Constructing a line

Every element in a line of text generates a *content area*, which is determined by the size of the font. This content area, in turn, generates an *inline box* that is, in the absence of any other factors, exactly equal to the content area. The leading generated by `line-height` is one of the factors that increases or decreases the height of each inline box.

To determine the leading for a given element, subtract the computed value of `font-size` from the computed value of `line-height`. That value is the total amount of leading. And remember, it can be a negative number. The leading is then divided in half, and each half-leading is applied to the top and bottom of the content area. The result is the inline box for that element. In this way, each line of text is

centered within the line-height as long as the height of the line isn't forced beyond its minimum height by a replaced element or other factor.

As an example, let's say the `font-size` (and therefore the content area) is 14 pixels tall, and the `line-height` is computed to 18 pixels. The difference (4 pixels) is divided in half, and each half is applied to the top and bottom of the content area. This effectively centers the content by creating an inline box that is 18 pixels tall, with 2 extra pixels above and below the content area. This sounds like a roundabout way to describe how `line-height` works, but there are excellent reasons for the description.

Once all of the inline boxes have been generated for a given line of content, they are then considered in the construction of the line box. A line box is exactly as tall as needed to enclose the top of the tallest inline box and the bottom of the lowest inline box. Figure 11-12 shows a diagram of this process.

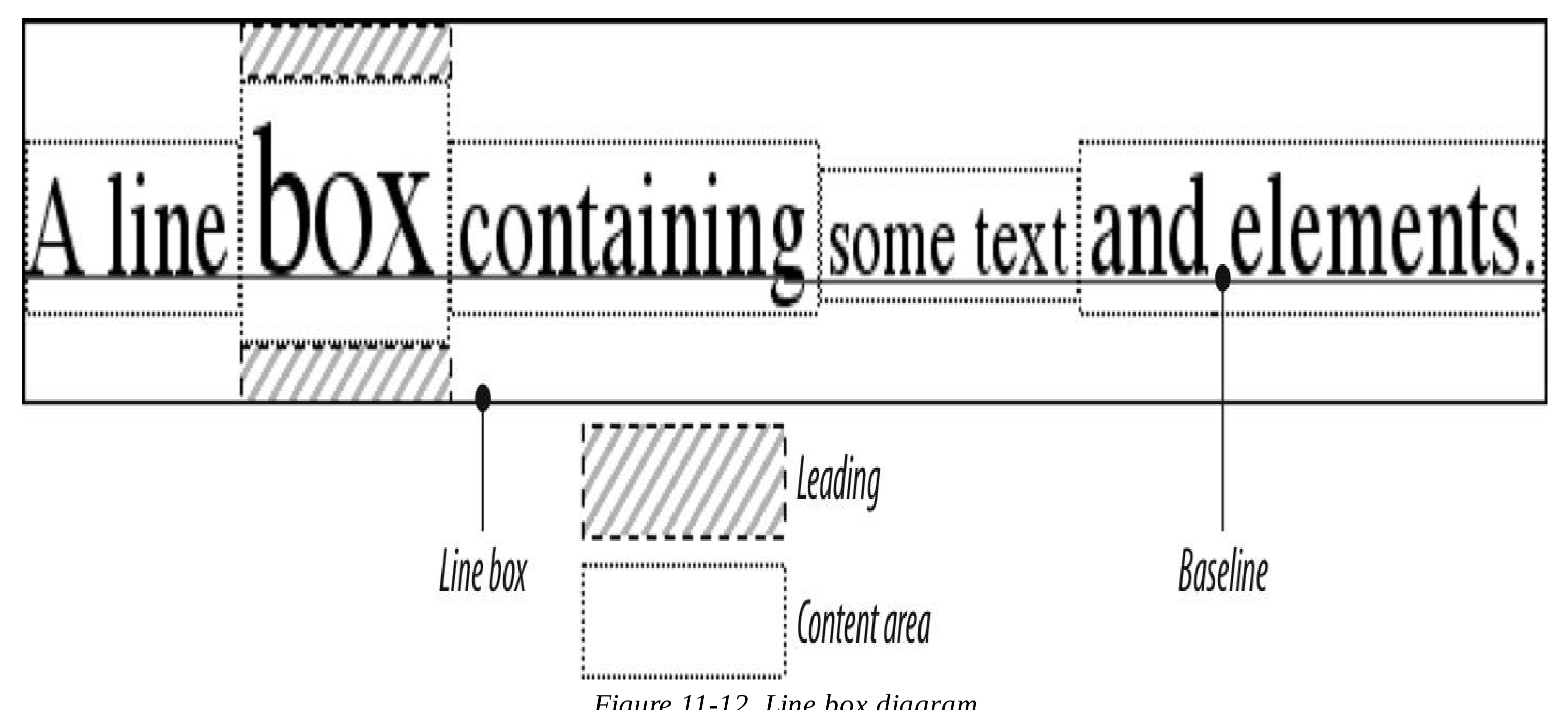


Figure 11-12. Line box diagram

Assigning values to line-height

Let's now consider the possible values of `line-height`. If you use the default value of `normal`, the user agent must calculate the space between lines. Values can vary by user agent, but the `normal` default is generally around 1.2 times the size of the font, which makes line boxes taller than the value of `font-size` for a given element.

Many values are simple length measures (e.g., `18px` or `2em`), but <*number*> values with no length unit are preferable in many situations. Be aware that even if you use a valid length measurement, such as `4cm`, the browser (or the operating system) may be using an incorrect metric for real-world measurements, so the line height may not show up as exactly four centimeters on your monitor.

> **WARNING**
>
> Be aware that even if you use a valid length measurement, such as `4cm`, the browser (or the operating system) may be using an incorrect metric for real-world measurements, so the line height may not show up as exactly four centimeters on your monitor.

`em`, `ex`, and percentage values are calculated with respect to the `font-size` of the element. The results of the following CSS and HTML are shown in Figure 11-13:

```
body {line-height: 18px; font-size: 16px;}
p.cl1 {line-height: 1.5em;}
p.cl2 {font-size: 10px; line-height: 150%;}
p.cl3 {line-height: 0.33in;}

<p>This paragraph inherits a 'line-height' of 18px from the body, as well as
a 'font-size' of 16px.</p>
<p class="cl1">This paragraph has a 'line-height' of 24px(16 * 1.5), so
it will have slightly more line-height than usual.</p>
<p class="cl2">This paragraph has a 'line-height' of 15px (10 * 150%), so
it will have slightly more line-height than usual.</p>
<p class="cl3">This paragraph has a 'line-height' of 0.33in, so it will have
slightly more line-height than usual.</p>
```

This paragraph inherits a 'line-height' of 18px from the body, as well as a 'font-size' of 16px.

This paragraph has a 'line-height' of 24px(16 * 1.5), so it will have slightly more line-height than usual.

This paragraph has a 'line-height' of 15px (10 * 150%), so it will have slightly more line-height than usual.

This paragraph has a 'line-height' of 0.33in, so it will have slightly more line-height than usual.

Figure 11-13. Simple calculations with the line-height property

Line-height and inheritance

When the `line-height` is inherited by one block-level element from another, things get a bit trickier. `line-height` values inherit from the parent element as computed from the parent, not the child. The results of the following markup are shown in Figure 11-14. It probably wasn't what the author had in mind:

```
body {font-size: 10px;}
div {line-height: 1em;}  /* computes to '10px' */
p {font-size: 18px;}

<div>
<p>This paragraph's 'font-size' is 18px, but the inherited 'line-height'
value is only 10px.  This may cause the lines of text to overlap each
other by a small amount.</p>
</div>
```

This paragraph's 'font-size' is 18px, but the inherited 'line-height' value is only 10px. This may cause the lines of text to overlap each other by a small amount.

Figure 11-14. Small line-height, large font-size, slight problem

Why are the lines so close together? Because the computed `line-height` value of `10px` was inherited by the paragraph from its parent `div+. One solution to the small +line-height` problem depicted in Figure 11-14 is to set an explicit `line-height` for every element, but that's not very practical. A better alternative is to specify a number, which actually sets a scaling factor:

```
body {font-size: 10px;}
div {line-height: 1;}
p {font-size: 18px;}
```

When you specify a number with no length unit, you cause the scaling factor to be an inherited value instead of a computed value. The number will be applied to the element and all of its child elements so that each element has a `line-height` calculated with respect to its own `font-size` (see Figure 11-15):

```
div {line-height: 1.5;}
p {font-size: 18px;}

<div>
<p>This paragraph's 'font-size' is 18px, and since the 'line-height'
set for the parent div is 1.5, the 'line-height' for this paragraph
is 27px (18 * 1.5).</p>
</div>
```

This paragraph's 'font-size' is 18px, and since the 'line-height' set for the parent div is 1.5, the 'line-height' for this paragraph is 27px (18 * 1.5).

Figure 11-15. Using line-height factors to overcome inheritance problems

Now that you have a basic grasp of how lines are constructed, let's talk about "vertically" aligning elements relative to the line box—that is, displacing them along the block direction.

Vertically Aligning Text

If you've ever used the elements `sup` and `sub` (the superscript and subscript elements), or used the deprecated `align` attribute with an image, then you've done some rudimentary vertical alignment.

> **NOTE**
>
> Because of the property name `vertical-align`, this section will use the terms "vertical" and "horizontal" to refer to the block and inline directions of the text.

VERTICAL-ALIGN

Values	`baseline` | `sub` | `super` | `top` | `text-top` | `middle` | `bottom` | `text-bottom` | *<length>* | *<percentage>*
Initial value	`baseline`
Applies to	Inline elements, the pseudo-elements `::first-letter` and `::first-line`, and table cells
Percentages	Refer to the value of `line-height` for the element
Computed value	For percentage and length values, the absolute length; otherwise, as specified
Inherited	No
Animatable	*<length>*, *<percentage>*
Note	When applied to table cells, only the values `baseline+, +top+, +middle+, and +bottom` are recognized

`vertical-align` accepts any one of eight keywords, a percentage value, or a length value. The keywords are a mix of the familiar and unfamiliar: `baseline` (the default value), `sub`, `super`, `bottom`, `text-bottom`, `middle`, `top`, and `text-top`. We'll examine how each keyword works in relation to inline elements.

> **NOTE**
>
> Remember: `vertical-align` does *not* affect the alignment of content within a block-level element, just the alignment of inline content within a line of text or a table cell. This may change in the future, but as of mid-2022, proposals to widen its scope have yet to move forward.

Baseline alignment

`vertical-align: baseline` forces the baseline of an element to align with the baseline of its parent. Browsers, for the most part, do this anyway, since you'd probably expect the bottoms of all text elements in a line to be aligned.

If a vertically aligned element doesn't have a baseline—that is, if it's an image, a form input, or another replaced element—then the bottom of the element is aligned with the baseline of its parent, as Figure 11-16 shows:

```
img {vertical-align: baseline;}

<p>The image found in this paragraph <img src="dot.gif" alt="A dot" /> has its
bottom edge aligned with the baseline of the text in the paragraph.</p>
```

The image found in this paragraph • has its bottom edge aligned with the baseline of the text in the paragraph.

Figure 11-16. Baseline alignment of an image

This alignment rule is important because it causes some web browsers to always put a replaced element's bottom edge on the baseline, even if there is no other text in the line. For example, let's say you have an image in a table cell all by itself. The image may actually be on a baseline, but in some browsers, the space below the baseline causes a gap to appear beneath the image. Other browsers will "shrink-wrap" the image with the table cell, and no gap will appear. The gap behavior is correct, despite its lack of appeal to most authors.

> **NOTE**
>
> See the deeply aged and yet somehow still relevant article "Images, Tables, and Mysterious Gaps" (2002) for a more detailed explanation of gap behavior and ways to work around it.

Superscripting and subscripting

The declaration `vertical-align: sub` causes an element to be subscripted, meaning that its baseline (or bottom, if it's a replaced element) is lowered with respect to its parent's baseline. The specification doesn't define the distance the element is lowered, so it may vary depending on the user agent.

`super` is the opposite of `sub` ; it raises the element's baseline (or bottom of a replaced element) with respect to the parent's baseline. Again, the distance the text is raised depends on the user agent.

Note that the values `sub` and `super` do *not* change the element's font size, so subscripted or superscripted text will not become smaller (or larger). Instead, any text in the sub- or superscripted element will, by default, be the same size as text in the parent element, as illustrated by Figure 11-17:

```
span.raise {vertical-align: super;}
span.lower {vertical-align: sub;}

<p>This paragraph contains <span class="raise">superscripted</span>
and <span class="lower">subscripted</span> text.</P>
```

This paragraph contains superscripted and subscripted text.

Figure 11-17. Superscript and subscript alignment

> **NOTE**
>
> If you wish to make super- or subscripted text smaller than the text of its parent element, you can do so using the `font-size` property.

Top and bottom alignment

`vertical-align: top` aligns the top of the element's inline box with the top of the line box. Similarly, `vertical-align: bottom` aligns the bottom of the element's inline box with the bottom of the line box. Thus, the following markup results in Figure 11-18:

```
.soarer {vertical-align: top;}
.feeder {vertical-align: bottom;}

<p>And in this paragraph, as before, we have
first a <img src="tall.gif" alt="tall" class="soarer" /> image and
then a <img src="short.gif" alt="short" class="soarer" /> image,
and then some text which is not tall.</p>

<p>This paragraph, as you can see, contains
first a <img src="tall.gif" alt="tall" class="feeder" /> image and
then a <img src="short.gif" alt="short" class="feeder" /> image,
and then some text which is not tall.</p>
```

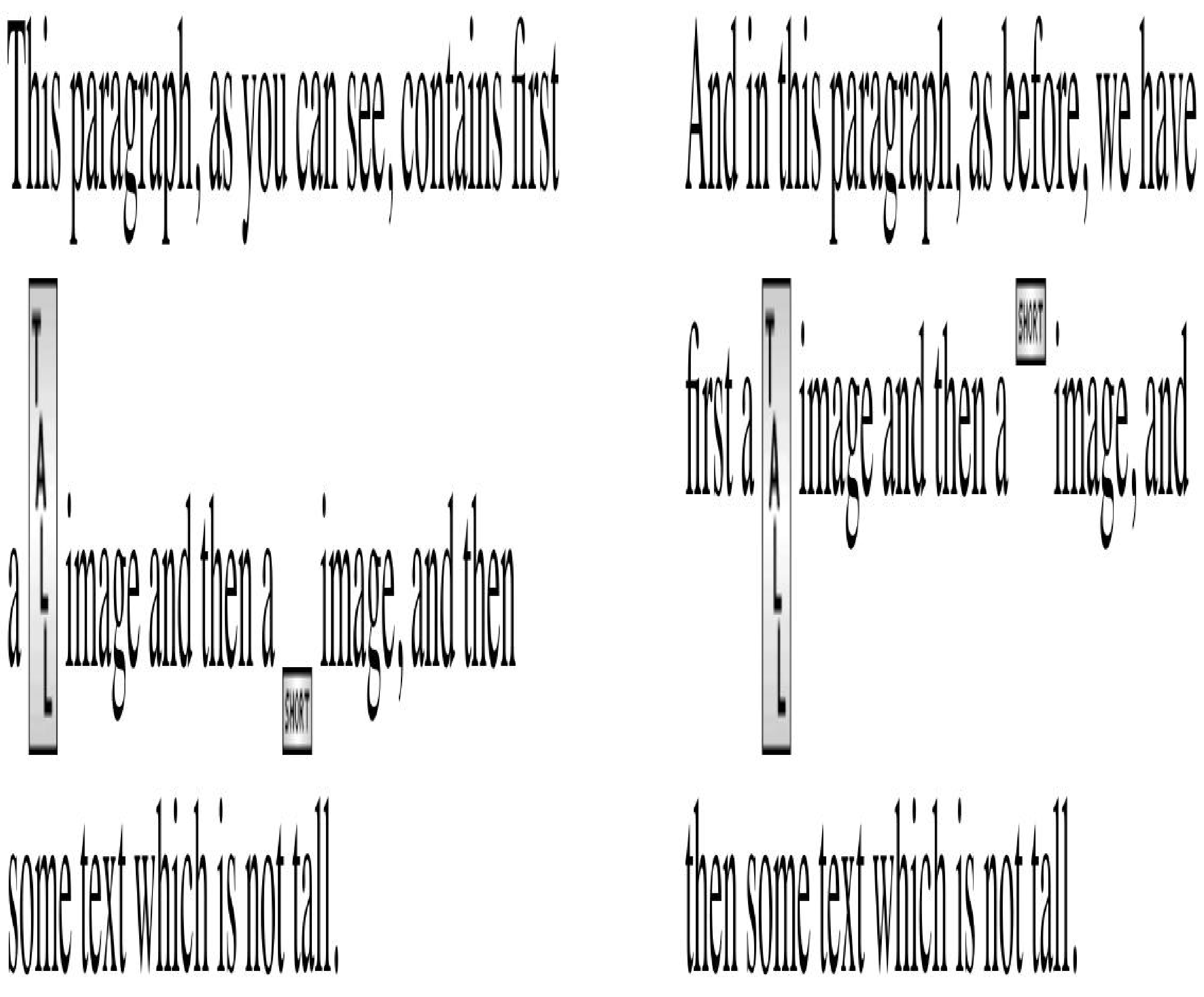


Figure 11-18. Top and bottom alignment

The second line of the first paragraph in Figure 11-18 contains two inline elements, whose top edges are aligned with each other. They're also well above the baseline of the text. The second paragraph shows the inverted case: two images whose bottoms are aligned and are well below the baseline of their line. This is because in both cases, the sizes of the elements in the line have increased the line's height beyond what the font's size would normally create.

If you want instead to align elements with the top or bottom edge of just the text in the line, then `text-top` and `text-bottom` are the values you seek. For the purposes of these values, replaced elements,

or any other kinds of non-text elements, are ignored. Instead, a “default” text box is considered. This default box is derived from the `font-size` of the parent element. The bottom of the aligned element’s inline box is then aligned with the bottom of the default text box. Thus, given the following markup, you get a result like the one shown in Figure 11-19:

```
img.ttop {vertical-align: text-top;}
img.tbot {vertical-align: text-bottom;}
```

```
<p>Here: a <img src="tall.gif" class="tbot" alt="tall" /> tall image,
and then a <img src="short.gif" class="tbot" alt="short" /> image.</p>
<p>Here: a <img src="tall.gif" class="ttop" alt="tall"> tall image,
and then a <img src="short.gif" class="ttop" alt="short" /> image.</p>
```

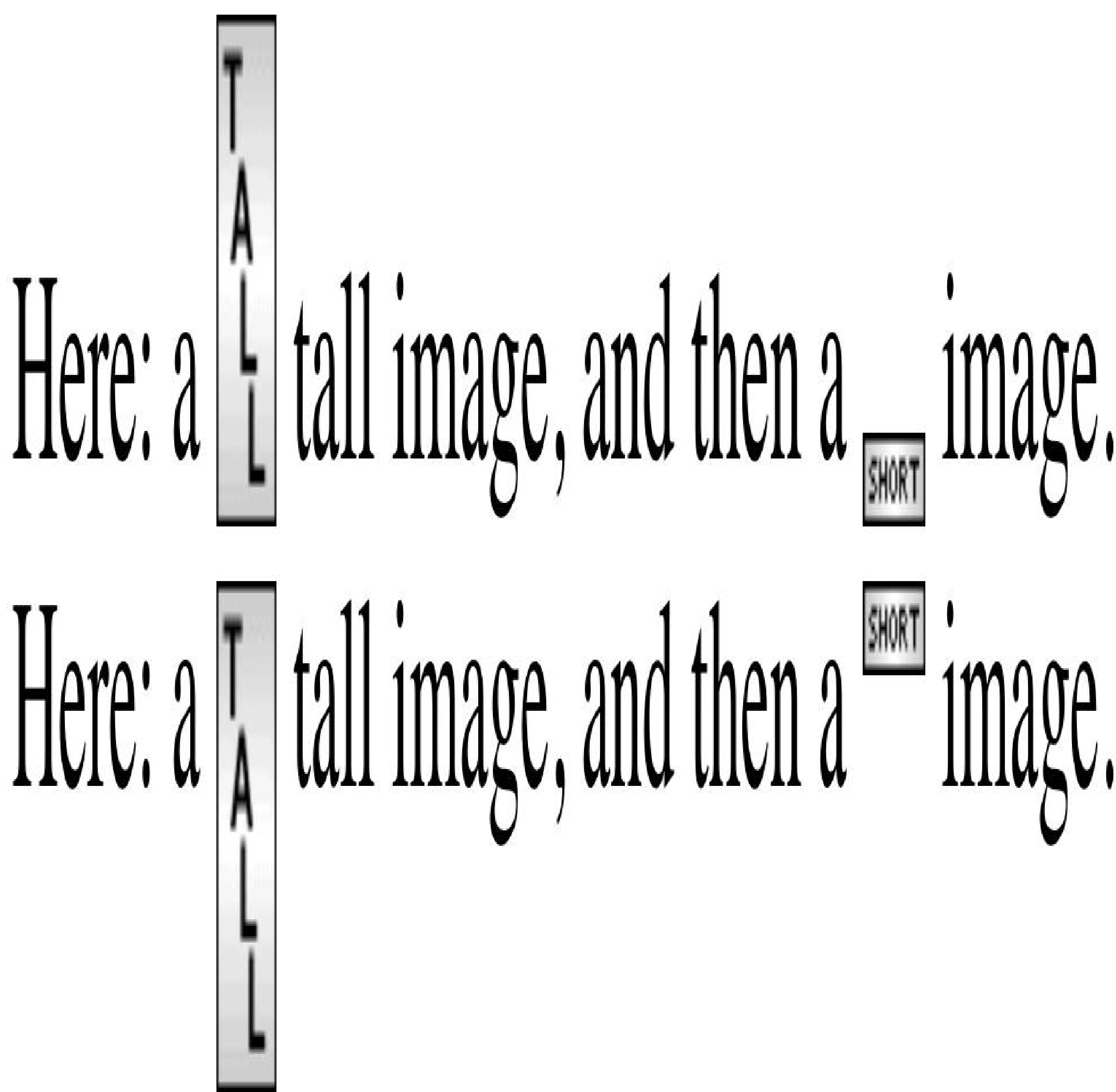


Figure 11-19. Text-top and -bottom alignment

Middle alignment

Then there's the value `middle+, which is usually (but not always) applied to images. It does not have the exact effect you might assume given its name. +middle` aligns the middle of an inline element's box with a point that is `0.5ex` above the baseline of the parent element, where `1ex` is defined relative to the `font-size` for the parent element. Figure 11-20 shows this in more detail.

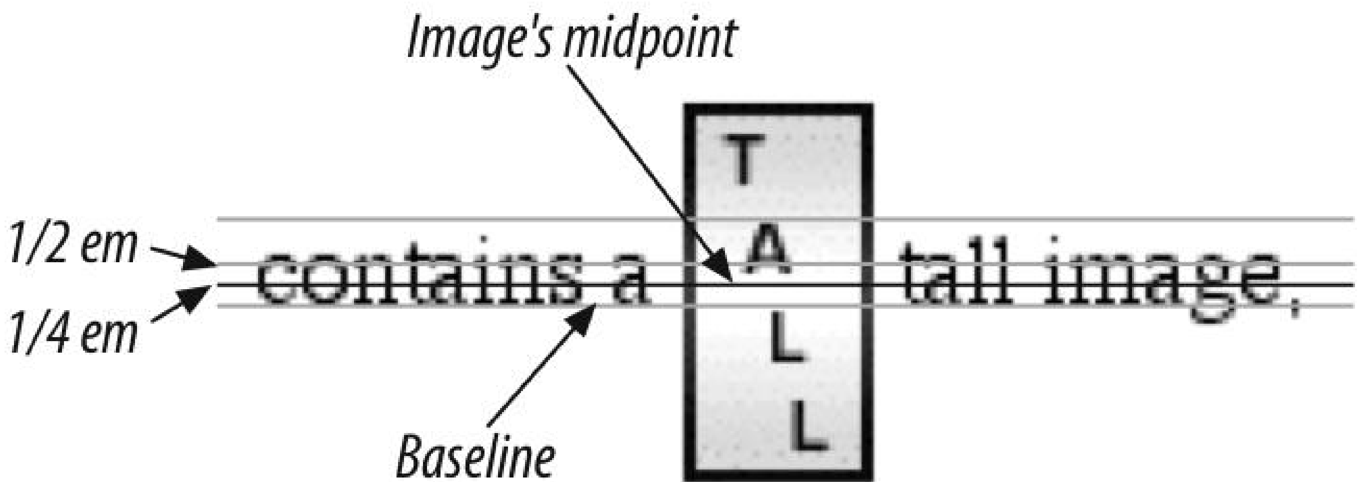


Figure 11-20. Precise detail of middle alignment

Since most user agents treat `1ex` as one-half em, `middle` usually aligns the vertical midpoint of an element with a point one-quarter em above the parent's baseline, though this is not a defined distance and so can vary from one user agent to another.

Percentages

Percentages don't let you simulate `align="middle"` for images. Instead, setting a percentage value for `vertical-align` raises or lowers the baseline of the element (or the bottom edge of a replaced element) by the amount declared, with respect to the parent's baseline. (The percentage you specify is calculated as a percentage of `line-height` for the element, *not* its parent.) Positive percentage values raise the element, and negative values lower it.

Depending on how the text is raised or lowered, it can appear to be placed in adjacent lines, as shown in Figure 11-21, so take care when using percentage values:

```
sub {vertical-align: -100%;}
sup {vertical-align: 100%;}

<p>We can either <sup>soar to new heights</sup> or, instead,
<sub>sink into despair...</sub></p>
```

soar to new heights
We can either or , instead,
sink into despair...

Figure 11-21. Percentages and fun effects

Length alignment

Finally, let's consider vertical alignment with a specific length. `vertical-align` is very basic: it shifts an element up or down by the declared distance. Thus, `vertical-align: 5px;` will shift an element upward five pixels from its unaligned placement. Negative length values shift the element downward.

It's important to realize that vertically aligned text does not become part of another line, nor does it overlap text in other lines. Consider Figure 11-22, in which some vertically aligned text appears in the

middle of a paragraph.

This paragraph contains a lot of text to be displayed, and a part of that text is
some **nicely bold text** which is raised up 100%. This makes it look as though
the bold text is part of its own line of text, when in fact the line in which is
sits is simply much taller than usual.

Figure 11-22. Vertical alignments can cause lines to get taller

As you can see, any vertically aligned element can affect the height of the line. Recall the description of a line box, which is exactly as tall as necessary to enclose the top of the tallest inline box and the bottom of the lowest inline box. This includes inline boxes that have been shifted up or down by vertical alignment.

Text Transformation

With the alignment properties covered, let’s look at ways to manipulate the capitalization of text using the property `text-transform`.

TEXT-TRANSFORM

Values	`uppercase` | `lowercase` | `capitalize` | `full-width` | `full-size-kana` | `none`
Initial value	`none`
Applies to	All elements
Computed value	As specified
Inherited	Yes
Animatable	No
Notes	`full-width` and `full-size-kana` were only supported in Firefox as of mid-2022

The default value `none` leaves the text alone and uses whatever capitalization exists in the source document. As their names imply, `uppercase` and `lowercase` convert text into all upper- or lowercase characters. `full-width` forces the writing of a character inside a square, as if on a typographical grid.

> **WARNING**
>
> Accessibility note: some screen readers will read all-uppercase text one letter at a time, as if spelling out an acronym, even if the source text is lowercase or mixed-case and the uppercasing is only enforced via CSS. For this reason, uppercasing text via CSS should be approached with caution.

Finally, the `capitalize` value capitalizes only the first letter of each word (where a "word" is defined as a string of adjacent characters surrounded by whitespace). Figure 11-23 illustrates each of these settings in a variety of ways:

```
h1 {text-transform: capitalize;}
strong {text-transform: uppercase;}
p.cummings {text-transform: lowercase;}
p.full {text-transform: full-width;}
p.raw {text-transform: none;}
```

```
<h1>The heading-one at the beginninG</h1>
<p>
By default, text is displayed in the capitalization it has in the source
document, but <strong>it is possible to change this</strong> using
the property 'text-transform'.
</p>
<p class="cummings">
For example, one could Create TEXT such as might have been Written by
the late Poet E.E.Cummings.
</p>
<p class="full">
If you need to align characters as if in a grid, as is often done in CJKV
languages, you can use 'full-width' to do so.
</p>
<p class="raw">
If you feel the need to Explicitly Declare the transformation of text
to be 'none', that can be done as well.
</p>
```

The Heading-one At The BeginninG

By default, text is displayed in the capitalization it has in the source document, but **IT IS POSSIBLE TO CHANGE THIS** using the property 'text-transform'.

for example, one could create text such as might have been written by the late poet e.e.cummings.

I f   y o u   n e e d   t o   a l i g n   c h a r a c t e r s   a s   i f   i n   a   g r i d ,   a s   i s   o f t e n   d o n e   i n   C J K V   l a n g u a g e s ,   y o u   c a n   u s e   ' f u l l - w i d t h '   t o   d o   s o .

If you feel the need to Explicitly Declare the transformation of text to be 'none', that can be done as well.

Figure 11-23. Various kinds of text transformation

> **NOTE**
>
> CJK stands for "Chinese/Japanese/Korean." CJK characters take up the majority of the entire Unicode code space, including approximately 70,000 Han characters. You may sometimes come across the abbreviation CJKV, which adds Vietnamese to the mix.

Different user agents may have different ways of deciding where words begin and, as a result, which letters are capitalized. For example, the text "heading-one" in the `h1` element, shown in Figure 11-23, could be rendered in one of two ways: "Heading-one" or "Heading-One." CSS does not say which is correct, so either is possible.

You may have also noticed that the last letter in the `h1` element in Figure 11-23 is still uppercase. This is correct: when applying a `text-transform` of `capitalize`, CSS only requires user agents to make sure the first letter of each word is capitalized. They can ignore the rest of the word.

As a property, `text-transform` may seem minor, but it's very useful if you suddenly decide to capitalize all your `h1` elements. Instead of individually changing the content of all your `h1` elements, you can just use `text-transform` to make the change for you:

```
h1 {text-transform: uppercase;}

<h1>This is an H1 element</h1>
```

The advantages of using `text-transform` are twofold. First, you only need to write a single rule to make this change, rather than changing the `h1` itself. Second, if you decide later to switch from all capitals back to initial capitals, the change is even easier:

```
h1 {text-transform: capitalize;}
```

Remember that `capitalize` is a simple letter substitution at the beginning of each "word." CSS doesn't check for grammar, so common headline-capitalization conventions, such as leaving articles ("a," "an," "the") all lowercase, won't be enforced.

Different languages have different rules for which letters should be capitalized. The `text-transform` property takes into account language-specific case mappings.

`full-width` forces the writing of a character inside a square. Most characters you can type on a keyboard come in both normal width and a full-width, with different Unicode code points. The full-width version is used when `full-width` is set and supported to mix them smoothly with Asian ideographic characters allowing ideograms and Latin scripts to be aligned.

Generally used with `<ruby>` annotation text, `full-size-kana` converts all small Kana characters to the equivalent full-size Kana, to compensate for legibility issues at the small font sizes typically used in ruby.

Text Decoration

Next we come to the topic of text decorations, and how we can affect them with various properties. The simplest text decoration, and the one that can be controlled the most, is an underline. There are also overlines, line-throughs, and even the wavy underlines you see in word processing programs to flag errors of spelling or grammar.

We'll start with the various individual properties, and then tie it all up with a shorthand property, `text-decoration`, that covers them all.

Text decoration line placement

With the property `text-decoration-line`, it's possible to set the location of one or more line decorations on a run of text. The most familiar decoration may be underlining, thanks to all the hyperlinks out there, but there are three possible visible decoration line values (plus an unsupported fourth that wouldn't draw a line at all even if it *was* supported).

TEXT-DECORATION-LINE

Values	`none` | [`underline` ‖ `overline` ‖ `line-through` ‖ `blink`]
Initial value	`none`
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	No
Notes	The `blink` value is deprecated, with all browsers treating it as `none` as of early 2022

The values are relatively self-documenting: `underline` places a line under the text, where "under" means "below the text in the block direction." `overline` is the mirror image, putting the line above the text in the block direction. `line-through` draws a line through the middle of the text.

Let's see what these decorations look like in practice. The following code is illustrated in Figure 11-24.

```
p.one {text-decoration: underline;}
p.two {text-decoration: overline;}
p.three {text-decoration: line-through;}
p.four {text-decoration: none;}
```

This text has been decorated with an underline.

This text has been decorated with an overline.

~~This text has been decorated with a line-through.~~

This text has been decorated with nothing at all.

Figure 11-24. Various kinds of text decoration

The value `none` turns off any decoration that might otherwise have been applied to an element. For example, links are usually underlined by default. If you want to suppress the underlining of hyperlinks, you can use the following CSS rule to do so:

```
a {text-decoration: none;}
```

If you explicitly turn off link underlining with this sort of rule, the only visual difference between the anchors and normal text will be their color (at least by default, though there's no ironclad guarantee that there will be a difference in their colors). Relying on color alone as the difference between regular text and links within that text is not enough to differentiate links from the rest of the text, negatively impacting user experience and making your content inaccessible to many users.

> **NOTE**
>
> Bear in mind that many users will be annoyed when they realize you've turned off link underlining, especially within blocks of text. If your links aren't underlined, users will have a hard time finding hyperlinks in your documents, and finding them can be next to impossible for users with one form or another of color blindness.

That's really all there is to `text-decoration-line`. The more veteran among you may recognize this is what `text-decoration` itself used to do, but times have moved on and there's much, much more we can do with decorations besides just place them, so these values were shifted to `text-decoration-line`.

Setting text decoration color

By default, the color of a text decoration will match the color of the text. If you need to change that, then `text-decoration-color` is here to help.

TEXT-DECORATION-COLOR

Values	*<color>* | currentColor
Initial value	`currentColor`
Applies to	All elements
Computed value	The computed color
Inherited	No
Animatable	Yes

You can use any valid color value for `text-decoration-color`, including the keyword

`currentColor` (which is the default). Suppose you want to make it clear that stricken text really is stricken. That would go something like:

```
del, strike, .removed {
        text-decoration-line: line-through;
        text-decoration-color: red;
}
```

Thus, not only will the elements shown get a line-through decoration, but the line will also be colored red. The text itself will not be red unless you change that as well by using the `color` property.

> **NOTE**
>
> Remember to keep the color contrast between decorations and the base text sufficiently high to remain accessible. It's also generally a bad idea to use color alone to convey meaning, as in "check the links with red underlines for more information!"

Setting text decoration thickness

With the property `text-decoration-thickness`, you can change the stroke thickness of a text decoration from to something beefier, or possibly less beefy, than usual.

TEXT-DECORATION-THICKNESS

Values	*<length>* | *<percentage>* | from-font | auto
Initial value	`auto`
Applies to	All elements
Computed value	As declared
Percentages	Refer to the `font-size` of the element
Inherited	No
Animatable	Yes
Notes	Was `text-decoration-width` until a name change in 2019

Supplying a length value sets the thickness of the decoration to that length; thus, `text-decoration-thickness: 3px` sets the decoration to be three pixels thick, no matter how big or small the text itself might be. A better approach is generally to use an em-based value or jump straight to using a percentage value, since percentages are calculated with respect to the value of `1em` for the element. Thus, `text-decoration-thickness: 10%` would yield a decoration thickness of 1.6 pixels in a font whose computed font size is 16 pixels, but 4 pixels for a 40-pixel font size. A few examples are shown in the following code, and illustrated in Figure 11-25.

```
h1, p {text-decoration-line: underline;}
.tiny {text-decoration-thickness: 1px;}
.embased {text-decoration-thickness: 0.333em;}
.percent {text-decoration-thickness: 10%;}
```

Figure 11-25. Various decoration thicknesses

The keyword `from-font` is interesting because it allows the browser to consult the font file to see if it defines a preferred decoration thickness; if it does, then the browser uses that thickness. If the font file doesn't recommend a thickness, then the browser falls back to the `auto` behavior, where the browser

picks whatever thickness it thinks appropriate, using inscrutable reasoning known only to itself.

Setting text decoration style

Thus far, we've seen a lot of straight, single lines. If you're yearning for something beyond that hidebound approach, then `text-decoration-style` provides a number of alternatives.

TEXT-DECORATION-STYLE

Values	solid | double | dotted | dashed | wavy
Initial value	`solid`
Applies to	All elements
Computed value	As declared
Inherited	No
Animatable	No

The exact result will depend on the value you pick and the browser you use to view the results, but the renderings of these decoration styles should be at least similar to those shown in Figure 11-26, which is the output of the following code.

```
p {text-decoration-line: underline; text-decoration-thickness: 0.1em;}
p.one {text-decoration-style: solid;}
p.two {text-decoration-style: double;}
p.three {text-decoration-style: dotted;}
p.four {text-decoration-style: dashed;}
p.five {text-decoration-style: wavy;}
```

This text has been decorated with a solid underline.

This text has been decorated with a double underline.

This text has been decorated with a dotted underline.

This text has been decorated with a dashed underline.

This text has been decorated with a wavy underline.

Figure 11-26. Various decoration styles

The decoration thickness was increased for Figure 11-26 in order to make them more legible — the default sizing of decorations can make some of the more complex decorations, like dotted, difficult to see.

The text decoration shorthand property

For those times when you just want to set a text decoration's position, color, thickness, and style in one handy declaration, `text-decoration` is the way to go.

TEXT-DECORATION

Values	*<text-decoration-line<* ‖ *<text-decoration-style<* ‖ *<text-decoration-color<* ‖ *<text-decoration-thickness<*
Initial value	See individual properties
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	As allowed by individual properties

With the `text-decoration` shorthand property, you can bring everything into one place, like so:

```
h2 {text-decoration: overline purple 10%;}
a:any-link {text-decoration: underline currentColor from-font;}
```

Be careful, though: if you have two different decorations matched to the same element, the value of the rule that wins out will completely replace the value of the loser. Consider:

```
h2.stricken {text-decoration: line-through wavy;}
h2 {text-decoration: underline overline double;}
```

Given these rules, any `h2` element with a class of `stricken` will have only a wavy line-through decoration. The doubled underline and overline decorations are lost, since shorthand values replace one another instead of accumulating.

Note also that because of the way the decoration properties work, you can only set the color and style once per element, even if you have multiple decorations. For example, the following is valid, setting both the under- and overlines to be green and dotted:

```
text-decoration: dotted green underline overline;
```

If you instead want the overline to be a different color than the underline, or set each to have its own style, you'd need to apply each to a separate element, something like this:

```
p {text-decoration: dotted green overline;}
p > span:first-child {text-decoration: silver dashed underline;}

<p><span>All this text will have differing text decorations.</span></p>
```

Offsetting underlines

Along with all the `text-decoration` properties, there's a related property that allows you to change the distance between an underline (and *only* an underline) and the text which that underline decorates: `text-underline-offset`.

TEXT-UNDERLINE-OFFSET

Values	*<length>* | *<percentage>* | auto
Initial value	`auto`
Applies to	All elements
Computed value	As specified
Percentages	Refer to the `font-size` of the element
Inherited	No
Animatable	Yes

You might wish that, say, underlines on hyperlinks were a little further away from the text's baseline, so that they're a little more obvious to the user. Setting a length value like `3px` will put the underline three pixels below the text's baseline. See Figure 11-27 for the results of the following CSS:

```
p {text-decoration-line: underline;}
p.one {text-underline-offset: auto;}
p.two {text-underline-offset: 2px;}
p.three {text-underline-offset: -2px;}
p.four {text-underline-offset: 0.5em;}
p.five {text-underline-offset: 15%;}
```

This text’s underline has been auto-placed.

This text’s underline has been offset 2px.

This text’s underline has been offset -2px.

This text’s underline has been offset 0.5em.

This text’s underline has been offset 15%.

Figure 11-27. Various underline offsets

As illustrated in Figure 11-27, the value defines an offset from the text's baseline, either positive (downward along the block axis) or negative (upward along the block axis).

As with `text-decoration-thickness`, percentage values for `text-underline-offset` are calculated with respect to the value of `1em` for the element. Thus, `text-underline-offset: 10%` would cause an offset of 1.6 pixels in a font whose computed font size is 16 pixels.

> **WARNING**
>
> As of late 2022, only Firefox supported percentage values for `text-underline-offset`, which is odd given that percentage values are a percent of 1em in the element's font. The workaround is to use em length values, such as 0.1em for 10%.

Skipping Ink

An unaddressed aspect of the past few sections has been: how exactly do browsers draw decorations over text, and more precisely, decide when to "skip over" parts of the text? This is known as "skipping ink," and the approach a browser takes can be altered with the property `text-decoration-skip-ink`.

TEXT-DECORATION-SKIP-INK

Values	all | none | auto
Initial value	`auto`
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	No

When ink skipping is turned on, the decoration is interrupted wherever it would cross over the shapes of the text. Usually, this means a small gap between the decoration and the text glyphs. See Figure 11-28 for a close-up illustration of the differences in ink-skipping approaches.

he thought sourly	he thought sourly	he thought sourly
彼は酸っぱいと思った	彼は酸っぱいと思った	彼は酸っぱいと思った
none	all	auto

Figure 11-28. Ink-skipping approaches

The three values are defined as follows:

1. `auto` (the default) means that the browser *may* interrupt under- and overlines where the line would cross the text glyphs, with a little space between the line and the glyphs. Furthermore, browsers *should* consider the glyphs used for the text, since some glyphs may call for ink skipping while others may not.

2. `all` means that browsers *must* interrupt under- and overlines where the line would cross the text glyphs, with a little space between the line and the glyphs. However, as of mid-2022, only Firefox supported this value.

3. `none` means the browser *must not* interrupt under- and overlines where the line would cross the text glyphs, but instead draw a continuous line even though it may be drawn over parts of the text glyphs.

As shown in Figure 11-28, `auto` can sometimes mean differences depending on the language, font, or based on other factors. You're really just telling the browser to do whatever it thinks is best.

> **NOTE**
>
> While this property's name begins with the label `text-decoration-`, it is *not* a property covered by the `text-decoration` shorthand property. That's why it's being discussed here, after the shorthand, and not before.

Weird Decorations

Now, let's look into the unusual side of `text-decoration`. The first oddity is that `text-decoration` is *not* inherited. No inheritance implies that any decoration lines drawn with the text—whether under, over, or through it—will always be the same color. This is true even if the descendant elements are a different color, as depicted in Figure 11-29:

```
p {text-decoration: underline; color: black;}
strong {color: gray;}

<p>This paragraph, which is black and has a black underline, also contains
<strong>strongly emphasized text</strong> which has the black underline
beneath it as well.</p>
```

This paragraph, which is black and has a black underline, also contains
strongly emphasized text which has the black underline beneath it as well.

Figure 11-29. Color consistency in underlines

Why is this so? Because the value of `text-decoration` is not inherited, the `strong` element assumes a default value of `none`. Therefore, the `strong` element has *no* underline. Now, there is very clearly a line under the `strong` element, so it seems silly to say that it has none. Nevertheless, it doesn't. What you see under the `strong` element is the paragraph's underline, which is effectively "spanning" the

`strong` element. You can see it more clearly if you alter the styles for the boldface element, like this:

```
p {text-decoration: underline; color: black;}
strong {color: gray; text-decoration: none;}

<p>This paragraph, which is black and has a black underline, also contains
<strong>strongly emphasized text</strong> which has the black underline beneath
it as well.</p>
```

The result is identical to the one shown in Figure 11-29, since all you've done is to explicitly declare what was already the case. In other words, there is no way to "turn off" the decoration generated by a parent element.

There is a way to change the color of a decoration without violating the specification. As you'll recall, setting a text decoration on an element means that the entire element has the same color decoration, even if there are child elements of different colors. To match the decoration color with an element, you must explicitly declare its decoration, as follows:

```
p {text-decoration: underline; color: black;}
strong {color: silver; text-decoration: underline;} /* could also use 'inherit' */

<p>This paragraph, which is black and has a black underline, also contains
<strong>strongly emphasized text</strong> which has the black underline
beneath it as well, but whose gray underline overlays the black underline
of its parent.</p>
```

In Figure 11-30, the `strong` element is set to be gray and to have an underline. The gray underline visually "overwrites" the parent's black underline, so the decoration's color matches the color of the `strong` element. The black underline is still there; the grey underline is just hiding it. If you move the gray underline with `text-underline-offset` or make the parent's `text-decoration-thickness` wider than its child, both underlines will be visible.

This paragraph, which is black and has a black underline, also contains
strongly emphasized text which has the black underline beneath it as well,
but whose gray underline overlays the black underline of its parent.

Figure 11-30. Overcoming the default behavior of underlines

When `text-decoration` is combined with `vertical-align`, even stranger things can happen. Figure 11-31 shows one of these oddities. Since the `sup` element has no decoration of its own, but it is elevated within an overlined element, the overline should cut through the middle of the `sup` element:

```
p {text-decoration: overline; font-size: 12pt;}
sup {vertical-align: 50%; font-size: 12pt;}
```

This paragraph, which is black and has a black overline, also contains
superscripted text through which the overline will cut.

Figure 11-31. Correct, although strange, decorative behavior

But not all browsers do this. As of mid-2022, Chrome would push the overline up so it is drawn across the top of the superscript, whereas others would not.

Text Rendering

A recent addition to CSS is `text-rendering`, which is actually an SVG property that's treated as CSS by supporting user agents. It lets authors indicate what the user agent should prioritize when displaying text.

TEXT-RENDERING

Values	`auto` | `optimizeSpeed` | `optimizeLegibility` | `geometricPrecision`
Initial value	`auto`
Applies to	All elements
Inherited	Yes
Animatable	Yes

The values `optimizeSpeed` and `optimizeLegibility` indicate that drawing speed should be favored over the use of legibility features like kerning and ligatures (for `optimizeSpeed`) or that such legibility features should be used even if that slows down text rendering (for `optimizeLegibility`).

The precise legibility features that are used with `optimizeLegibility` are not explicitly defined, and the text rendering often depends on the operating system on which the user agent is running, so the exact results may vary. Figure 11-32 shows text optimized for speed, and then optimized for legibility.

Ten Vipers Infiltrate AWACS
Ten Vipers Infiltrate AWACS

Figure 11-32. Different optimizations

As you can see in Figure 11-32, the differences between the two optimizations are objectively rather small, but they can have a noticeable impact on readability.

> **NOTE**
>
> Some user agents will always optimize for legibility, even when optimizing for speed. This is likely an effect of rendering speeds having gotten so fast in the past few years.

The value `geometricPrecision`, on the other hand, directs the user agent to draw the text as

precisely as possible, such that it could be scaled up or down with no loss of fidelity. You might think that this is always the case, but not so. Some fonts change kerning or ligature effects at different text sizes, for example, providing more kerning space at smaller sizes and tightening up the kerning space as the size is increased. With `geometricPrecision`, those hints are ignored as the text size changes. If it helps, think of it as the user agent drawing the text as though all the text is a series of SVG paths, not font glyphs.

Even by the usual standard of web standards, the value `auto` is pretty vaguely defined in SVG:

> *the user agent shall make appropriate tradeoffs to balance speed, legibility and geometric precision, but with legibility given more importance than speed and geometric precision.*

That's it: user agents get to do what they think is appropriate, leaning towards legibility.

Text Shadows

Sometimes, you just really need your text to cast a shadow, like when text overlaps a multicolored background. That's where `text-shadow` comes in. The syntax might look a little wacky at first, but it should become clear enough with just a little practice.

TEXT-SHADOW

Values	`none` | [*<length>* ‖ *<length>* *<length>* *<color>*?]#
Initial value	`none`
Applies to	All elements
Inherited	No
Animatable	Yes

The default is to not have a drop shadow for text. Otherwise, it's possible to define one or more shadows. Each shadow is defined by an optional color and three length values, the last of which is also optional.

The color sets the shadow's color so it's possible to define green, purple, or even white shadows. If the color is omitted, the shadow defaults to the color keyword *currentColor*, making it the same color as the text itself.

`currentColor` as a default color may seem counter-intuitive, as you might think shadows are purely decorative, but shadows can be used to improve legibility. A small shadow can make very thin text more legible. Defaulting to `currentColor` allows adding thickness via a shadow that will always match the color of the text.

In addition to improving accessibility by making thin text thicker, shadows can also be used to improve color contrast with a multi-colored background. For example, if you have white text on a mostly-dark black and white photo, adding a black shadow to the white text makes the edges of the white text visible

even if the text is laid over white portions of the image.

The first two length values determine the offset distance of the shadow from the text; the first is the horizontal offset and the second is the vertical offset. To define a solid, un-blurred green shadow offset five pixels to the right and half an em down from the text, as shown in Figure 11-33, you could write either of the following:

```
text-shadow: green 5px 0.5em;
text-shadow: 5px 0.5em green;
```

Negative lengths cause the shadow to be offset to the left and upward from the original text. The following, also shown in Figure 11-33, places a light blue shadow five pixels to the left and half an em above the text:

```
text-shadow: rgb(128,128,255) -5px -0.5em;
```

Figure 11-33. Simple shadows

While the offset may make the text take more visual space, shadows have no effect on line height, and therefore no impact on the box model.

The optional third length value defines a *blur radius* for the shadow. The blur radius is defined as the distance from the shadow's outline to the edge of the blurring effect. A radius of two pixels would result in blurring that fills the space between the shadow's outline and the edge of the blurring. The exact blurring method is not defined, so different user agents might employ different effects. As an example, the following styles are rendered as shown in Figure 11-34:

```
p.cl1 {color: black; text-shadow: gray 2px 2px 4px;}
p.cl2 {color: white; text-shadow: 0 0 4px black;}
p.cl3 {color: black;
       text-shadow: 1em 0.5em 5px red,
                    -0.5em -1em hsla(100,75%,25%,0.33);}
```

Figure 11-34. Dropping shadows all over

> **WARNING**
>
> Note that large numbers of text shadows, or text shadows with very large blur values, can create performance slowdowns, particularly when animated in low-power and CPU-constrained situations such as mobile devices. Authors are advised to test thoroughly before deploying public designs that use text shadows.

Emphasizing Text

Another way to call out text is by adding emphasis marks to the text, where each character is given a mark. This is more common in ideographic languages like Chinese or Mongolian, but these marks can be added to any language's text with CSS. There are three text emphasis properties similar to those we saw for text decorations, and then a shorthand that conflates two of them.

Setting emphasis style

The most important of the three properties sets the type of emphasis mark, allowing you to pick from a list of common mark types, or supply your own mark as a text string.

TEXT-EMPHASIS-STYLE

Values	none | [[filled | open] ‖ [dot | circle | double-circle | triangle | sesame]] | *<string>*
Initial value	`none`
Applies to	Text
Computed value	As declared, or `none` if nothing is declared
Inherited	Yes
Animatable	No
Note	As of mid-2022, most browsers only supported this as `-webkit-text-emphasis-style`, except Firefox, which only supported `text-emphasis-style`

By default, text has no emphasis marks, or `none`. Alternatively, emphasis marks can be one of five shapes: `dot`, `circle`, `double-circle`, `triangle`, or `sesame`. Those shapes can be set as `filled`, which is the default; or `open`, which renders them as unfilled outlines. These are summarized in ???, and examples are shown in Figure 11-35.

1. The predefined emphasis marks

Shape	filled	open
sesame	(U+FE45)	(U+FE46)
dot	• (U+2022)	◦ (U+25E6)
circle	● (U+25CF)	○ (U+25CB)
double-circle	(U+25C9)	(U+25CE)
triangle	▲ (U+25B2)	(U+25B3)

The sesame is the most common mark used in vertical writing modes; the circle is the usual default in horizontal writing modes.

In cases where the emphasis marks will not fit into the current text line's height, they will cause the height of that line of text to be increased until they fit without overlapping other lines. Unlike text decorations and text shadows, text emphasis marks *do* affect the line height.

If none of the pre-defined marks work in your specific situation, you can supply your own character as a string (a single character in single or double quotes). However, be careful: if the string is more than a single character, it may be reduced to the first character in the string by the browser. Thus, `text-emphasis-style: 'cool'` may see the browser only use the `c` as a mark, as shown in Figure 11-35. Furthermore, the string symbols may or may not be rotated to match writing direction in vertical languages.

	sesame	dot	circle	double-circle	triangle	"cool"
filled	horizontal language marks 세로 언어 마크	horizontal language marks 세로 언어 마크	horizontal language marks 세로 언어 마크	horizontal language marks 세로 언어 마크	horizontal language marks 세로 언어 마크	horizontal language marks 세로 언어 마크
open	horizontal language marks 세로 언어 마크	horizontal language marks 세로 언어 마크	horizontal language marks 세로 언어 마크	horizontal language marks 세로 언어 마크	horizontal language marks 세로 언어 마크	horizontal language marks 세로 언어 마크

Figure 11-35. Various emphasis marks

Some examples of setting emphasis marks:

```
h1 em {text-emphasis-style: triangle;}
strong a:any-link {text-emphasis-style: filled sesame;}
strong.callout {text-emphasis-style: open double-circle;}
```

A key difference between text emphasis and text decoration is that unlike decoration, emphasis is inherited. In other words, if you set a style of `filled sesame` on a paragraph, and that paragraph has child elements like links, those child elements will inherit the `filled sesame` value.

Another difference is that every glyph (character or other symbol) gets its own mark, and these marks are centered on the glyph. Thus, in proportional fonts like those seen in Figure 11-35, the marks will have different separations between them depending on which two glyphs are next to each other.

The CSS specification recommends that emphasis marks be half the size of the text's font size, as if they were given `font-size: 50%`. They should otherwise use the same text styles as the text; thus, if the text is boldfaced, the emphasis marks should be as well. They should also use the text's color, unless

overridden with the next property we’ll cover.

Changing emphasis color

If you have a scenario where you wish to have the emphasis marks be a different color than the text they’re marking, then `text-emphasis-color` is here for you.

TEXT-EMPHASIS-COLOR

Values	*<color>*
Initial value	`currentColor`
Applies to	Text
Computed value	The computed color
Inherited	Yes
Animatable	No
Note	As of mid-2022, most browsers only supported this as `-webkit-text-emphasis-color`, except Firefox, which only supported `text-emphasis-color`

The default value, as is often the case with color-related properties, is `currentColor`. That ensures emphasis marks will match the color of the text by default. To change it, you can do things like the following:

```
strong {text-emphasis-style: filled triangle;}
p.one strong {text-emphasis-color: gray;}
p.two strong {text-emphasis-color: hsl(0 0% 50%);}
/* these will yield the same visual result */
```

Placing emphasis marks

Thus far, we’ve seen emphasis marks in specific positions: above each glyph in horizontal text, and to the right of each glyph in vertical text. These are the default CSS values, but not always the preferred placement. `text-emphasis-position` allows you to change where marks are placed.

TEXT-EMPHASIS-POSITION

Values	[over | under] && [right | left]
Initial value	`over right`
Applies to	Text
Computed value	As declared
Inherited	Yes
Animatable	No
Note	As of mid-2022, most browsers only supported this in the form `-webkit-text-emphasis-position`, except Firefox, which only supported `text-emphasis-position`

The values `over` and `under` are only applied when the typographic mode is horizontal. Similarly, `right` and `left` are only used when the typographic mode is vertical.

This can be important in some Eastern languages. For example, Chinese, Japanese, Korean, and Mongolian all prefer to have marks to the right when the text is written vertically. They diverge on horizontal text: Chinese prefers marks below the text, and the rest prefer above the text, when it's horizontal. Thus you might write something like this in a style sheet:

```
:lang(cn) {text-emphasis-position: under right;}
```

This would override the default `over right` in cases where the text is marked as being Chinese, applying `under right` instead.

The text-emphasis shorthand

There is a shorthand for the `text-emphasis` properties, but it only brings together style and color.

TEXT-EMPHASIS

Values	*<text-emphasis-style>* ‖ *<text-emphasis-color>*
Initial value	See individual properties
Applies to	Text
Computed value	See individual properties
Inherited	Yes
Animatable	No
Note	As of mid-2022, most browsers only supported this in the form `-webkit-text-emphasis-position`, except Firefox, which only supported `text-emphasis-position`

The reason `text-emphasis-position` is not included in the `text-emphasis` shorthand is so that it can (indeed must) be inherited separately. Therefore, the style and color of the marks can be changed via `text-emphasis` without overriding the position in the process.

As stated earlier, each character or ideogram or other glyph — what CSS calls a “typographic character unit” — gets its own emphasis mark. That was roughly correct, but there are exceptions. The following character units do *not* get emphasis marks:

1. Word separators such as spaces, or any other Unicode separator character.
2. Punctuation characters, such as commas, full stops, and parentheses.
3. Unicode symbols corresponding to control codes, or any unassigned characters.

Text drawing order

Browsers are supposed to use a specific order to draw the text decorations, shadows, and emphasis marks we’ve discussed previously, along with the text itself. These are drawn in the following order, from bottom-most (furthest away from the user) to top-most (closest to the user):

1. Shadows (`text-shadow`)
2. Underlines (`text-decoration`)
3. Overlines (`text-decoration`)
4. The actual text
5. Emphasis marks (`text-emphasis`)
6. Line-through (`text-decoration`)

Thus, the drop shadows of the text are placed behind everything else. Underlines and overlines go behind the text. Emphasis marks and line-throughs go on top of the text. Note that if you have top text-emphasis marks and an overline, the emphasis marks will be drawn on top of the overline, obfuscating the overline where they overlap.

Handling Whitespace

Now that we've covered a variety of ways to style, decorate, and otherwise enhance the text, let's talk about the property `white-space`, which affects the user agent's handling of space, newline, and tab characters within the document source.

WHITE-SPACE

Values	`normal` | `nowrap` | `pre` | `pre-wrap` | `pre-line` | `break-spaces`
Initial value	`normal`
Applies to	All elements
Computed value	As declared
Inherited	No
Animatable	No

By using the `white-space` property, you can affect how a browser treats the whitespace between words and lines of text. To a certain extent, default HTML handling already does this: it collapses any whitespace down to a single space. So given the following markup, the rendering in a web browser would show only one space between each word and ignore the line-feed in the elements:

```
<p>This    paragraph   has     many spaces        in it.</p>
```

You can explicitly set this default behavior with the following declaration:

```
p {white-space: normal;}
```

This rule tells the browser to do as browsers have always done: discard extra whitespace. Given this value, line-feed characters (carriage returns) are converted into spaces, and any sequence of more than one space in a row is converted to a single space.

Should you set `white-space` to `pre`, however, the whitespace in an affected element is treated as though the elements were HTML `pre` elements; whitespace is *not* ignored, as shown in Figure 11-36:

```
p {white-space: pre;}
```

```
<p>This    paragraph   has     many
    spaces        in it.</p>
```

This   paragraph   has   many
    spaces   in it.

Figure 11-36. Honoring the spaces in markup

With a `white-space` value of `pre`, the browser will pay attention to extra spaces and even carriage returns. In this respect, any element can be made to act like a `pre` element.

The opposite value is `nowrap`, which prevents text from wrapping within an element, except wherever you use a `br` element. When text can’t wrap and it gets too wide for its container, a horizontal scroll bar will appear by default (this can be changed using the `overflow` property). The effects of the following markup are shown in Figure 11-37:

```
<p style="white-space: nowrap;">This paragraph is not allowed to wrap,
which means that the only way to end a line is to insert a line-break
element.  If no such element is inserted, then the line will go forever,
forcing the user to scroll horizontally to read whatever can't be
initially displayed <br/>in the browser window.</p>
```

This paragraph is not allowed to wrap, which means that the only way to end a line is to insert a line-bre
in the browser window.

Figure 11-37. Suppressing line wrapping with the white-space property

If an element is set to `pre-wrap`, then text within that element has whitespace sequences preserved, but text lines are wrapped normally. With this value, generated line-breaks as well as those found in the source markup are both honored.

`pre-line` is the opposite of `pre-wrap` and causes whitespace sequences to collapse as in normal text but honors new lines.

`break-spaces` is similar to `pre-wrap`, except that all white space is preserved, even at the end of the line, with a line break opportunity after each white space character. These spaces take up space and do not hang, and thus affect the box’s intrinsic sizes (min-content size and max-content size).

Table 11-1 summarizes the behaviors of the various `white-space` properties.

Table 11-1. White-space properties

Value	Whitespace	Line feeds	Auto line wrapping	Trailing whitespace
`pre-line`	Collapsed	Honored	Allowed	Removed
`normal`	Collapsed	Ignored	Allowed	Removed
`nowrap`	Collapsed	Ignored	Prevented	Removed
`pre`	Preserved	Honored	Prevented	Preserved
`pre-wrap`	Preserved	Honored	Allowed	Hanging
`break-spaces`	Preserved	Honored	Allowed	Wrap

Consider the following markup, which has linefeed (e.g., return) characters to break lines, plus the end of each line has an extra several space characters which aren’t visible in the markup. The results are illustrated in Figure 11-38:

```
<p style="white-space: pre-wrap;">
This  paragraph      has  a  great   many   s p a c e s   within  its textual
  content,   but their    preservation     will    not    prevent   line
    wrapping or line breaking.
</p>
<p style="white-space: pre-line;">
This  paragraph      has  a  great   many   s p a c e s   within  its textual
  content,   but their collapse  will    not    prevent   line
    wrapping or line breaking.
<p style="white-space: break-spaces;">
This  paragraph      has  a  great   many   s p a c e s   within  its textual
  content,   but their preservation  will    not    prevent   line
    wrapping or line breaking.
</p>
```

This paragraph has a great many s p a c e s within its textual content, but their preservation will not prevent line wrapping or line breaking.

This paragraph has a great many s p a c e s within its textual content, but their collapse will not prevent line wrapping or line breaking.

This paragraph has a great many s p a c e s within its textual

content, but their preservation will not prevent line wrapping or line breaking.

Figure 11-38. Three different ways to handle whitespace

Notice how the third bit of text has a blank line between the first and second lines of text. This is because a line-wrap was performed between two adjacent blank spaces at the end of the line in the source markup. This didn't happen for `pre-wrap` or `pre-line`, because those `white-space` values don't allow hanging space to create line-wrap opportunities. `break-spaces` does.

White space impacts several properties, including `tab-size`, which has no effect when the `white-space` property value is set to a value in which white space is not maintained; and `overflow-wrap`, which only has an effect when `white-space` allows wrapping.

Setting Tab Sizes

Since whitespace is preserved in some values of `white-space`, it stands to reason that tabs (i.e., Unicode code point 0009) will be displayed as, well, tabs. But how many spaces should each tab equal? That's where `tab-size` comes in.

TAB-SIZE

Values	*<length>* | *<integer>*
Initial value	8
Applies to	Block elements
Computed value	The absolute-length equivalent of the specified value
Inherited	Yes
Animatable	Yes

By default, when white spaces are preserved, as with `white-space` values of `pre`, `pre-wrap`, and `break-spaces`, any tab character will be treated the same as eight spaces in a row, including any `letter-spacing` and `word-spacing`. You can alter that by using a different integer value. Thus, `tab-size: 4` will cause each tab to be rendered as if it were four spaces in a row. Negative values are not allowed for `tab-size`.

If a length value is supplied, then each tab is rendered using that length. For example, `tab-size: 10px` will cause a sequence of three tabs to be rendered as 30 pixels of whitespace. Some effects of `tab-size` are illustrated in Figure 11-39.

This sentence is preceded by three tabs, set to a length of `8`.

This sentence is preceded by three tabs, set to a length of `4`.

This sentence is preceded by three tabs, set to a length of `2`.

This sentence is preceded by three tabs, set to a length of `0`.

This sentence is preceded by three tabs, set to a length of `8`—but `white-space` is `normal`.

Figure 11-39. Differing tab lengths

Remember that `tab-size` is effectively ignored when the value of `white-space` causes whitespace to be collapsed (see Table 11-1). The value will still be computed in such cases, but there will be no visible effect no matter how many tabs appear in the source.

Wrapping and Hyphenation

Handling white space is all well and good, but it's a lot more common to want to influence how the visible characters are handled when it comes to line-wrapping. There are a few properties that can influence where line-wrapping is allowed, as well as enabling hyphenation support.

Hyphenation

Hyphens can be very useful when there are long words and short line lengths, such as blog posts on mobile devices and portions of *The Economist*. Authors can always insert their own hyphenation hints using the Unicode character U+00AD SOFT HYPHEN (or, in HTML, ``), but CSS also offers a way to enable hyphenation without littering up the document with hints.

HYPHENS

Values	`manual` | `auto` | `none`
Initial value	`manual`
Applies to	All elements
Computed value	As specified
Inherited	Yes
Animatable	No

With the default value of `manual`, hyphens are only inserted where there are manually-inserted hyphenation markers in the document, such as U+00AD or ``. Otherwise, no hyphenation occurs. The value `none`, on the other hand, suppresses any hyphenation, even if manual break markers are present; thus, U+00AD and `` are ignored.

> **TIP**
>
> The `<wbr>` element does not introduce a hyphen at the line break point. To make a hyphen appear only at the end of a line, use the soft hyphen character entity () instead.

The far more interesting (and potentially inconsistent) value is `auto`, which permits the browser to insert hyphens and break words at "appropriate" places inside words, even where no manually inserted hyphenation breaks exist. But what constitutes a "word"? And, under what circumstances is it appropriate to hyphenate a word? Both are language-dependent. User agents are supposed to prefer manually inserted hyphen breaks to automatically determined breaks, but there are no guarantees. An illustration of hyphenation, or the suppression thereof, in the following example is shown in Figure 11-40:

```
.cl01 {hyphens: auto;}
.cl02 {hyphens: manual;}
.cl03 {hyphens: none;}

<p class="cl01">Supercalifragilisticexpialidocious
  antidisestablishmentarianism.</p>
<p class="cl02">Supercalifragilisticexpialidocious
  antidisestablishmentarianism.</p>
<p class="cl02">Super&shy;cali&shy;fragi&shy;listic&shy;expi&shy;ali&shy;
docious anti&shy;dis&shy;establish&shy;ment&shy;arian&shy;ism.</p>
<p class="cl03">Super&shy;cali&shy;fragi&shy;listic&shy;expi&shy;ali&shy;
docious anti&shy;dis&shy;establish&shy;ment&shy;arian&shy;ism.</p>
```

12em wide | 10em wide | 8em wide

hyphens: auto
No soft hyphen entities

Supercalifragilisticexpialido-
cious antidisestablishmentari-
anism.

Supercalifragilisticexpi-
alidocious antidisestab-
lishmentarianism.

Supercalifragilistic-
expialidocious an-
tidisestablishmen-
tarianism.

hyphens: manual
No soft hyphen entities

Supercalifragilisticexpialidocious
antidisestablishmentarianism.

Supercalifragilisticexpialidocious
antidisestablishmentarianism.

Supercalifragilisticexpialidocious
antidisestablishmentarianism.

hyphens: manual
Soft hyphen entities

Supercalifragilisticexpiali-
docious antidisestablishment-
arianism.

Supercalifragilisticexpi-
alidocious antidis-
establishmentarianism.

Supercalifragilistic-
expialidocious anti-
disestablishment-
arianism.

hyphens: none
Soft hyphen entities

Supercalifragilisticexpialidocious
antidisestablishmentarianism.

Supercalifragilisticexpialidocious
antidisestablishmentarianism.

Supercalifragilisticexpialidocious
antidisestablishmentarianism.

Figure 11-40. Hyphenation results

Because hyphenation is language-dependent, and because the CSS specifications do not define precise (or even vague) rules regarding how user agents should carry out hyphenation, it may differ by browser.

If you do choose to hyphenate, be careful about the elements to which you apply the hyphenation. `hyphens` is an inherited property, so declaring `body {hyphens: auto;}` will apply hyphenation to everything in your document—including textareas, code samples, block quotes, and so on. Blocking automatic hyphenation at the level of those elements is probably a good idea, using rules something like this:

```
body {hyphens: auto;}
code, var, kbd, samp, tt, dir, listing, plaintext, xmp, abbr, acronym,
blockquote, q, textarea, input, option {hyphens: manual;}
```

It's usually a good idea to suppress hyphenation in code samples and code blocks is desirable, especially in languages that use hyphens in things like property and value names. (Ahem.) Similar logic holds for keyboard input text—you very likely don't want a stray dash getting into your Unix command-line examples! And so on down the line. If you decide that you want to hyphenate some of these elements, just remove them from the selector.

> **NOTE**
>
> It is strongly advised to set the `lang` attribute on HTML elements to enable hyphenation support and improve accessibility. As of mid-2022, `hyphens` is supported in Firefox for 30+ languages, Safari supports many European languages, but Chrome-related browsers only support English.

Hyphens can be suppressed by the effects of other properties, such as `word-break`, which affects how soft wrapping of text is calculated in various languages, determining whether line breaks appear where text would otherwise overflow its content box.

Word breaking

When a run of text is too long to fit into a single line, it is *soft wrapped*. This is in contrast to *hard wraps*, which are things like line-feed characters and `<br>` elements. Where the text is soft wrapped is determined by the user agent, but `word-break` lets authors influence that decision-making.

WORD-BREAK

Values	`normal` | `break-all` | `keep-all` | `break-word`
Initial value	`normal`
Applies to	Text
Computed value	As specified
Inherited	Yes
Animatable	No
Note	`break-word` is a legacy value and has been deprecated

The default value of `normal` means that text should be wrapped like it always has been. In practical terms, this means that text is broken between words, though the definition of a word varies by language. In Latin-derived languages like English, this is almost always a space between letter sequences (e.g., words) or at hyphens. In ideographic languages like Japanese, each symbol can be a complete word, so breaks can occur between any two symbols. In other ideographic languages, though, the soft-wrap points may be limited to appear between sequences of symbols that are not space-separated. Again, that's all by default, and is the way browsers have handled text for years

If you apply the value `break-all`, then soft wrapping can (and will) occur between any two characters, even if they are in the middle of a word. With this value, no hyphens are shown, even if the soft wrapping occurs at a hyphenation point (see `hyphens`, earlier). Note that values of the `line-break` property (described next) can affect the behavior of `break-all` in ideographic text.

`keep-all`, on the other hand, suppresses soft wrapping between characters, even in ideographic

languages where each symbol is a word. Thus, in Japanese, a sequence of symbols with no whitespace will not be soft wrapped, even if this means the text line will exceed the length of its element. (This behavior is similar to `white-space: pre`.)

Figure 11-41 shows a few examples of `word-break` values, and Table 11-2 summarizes the effects of each value.

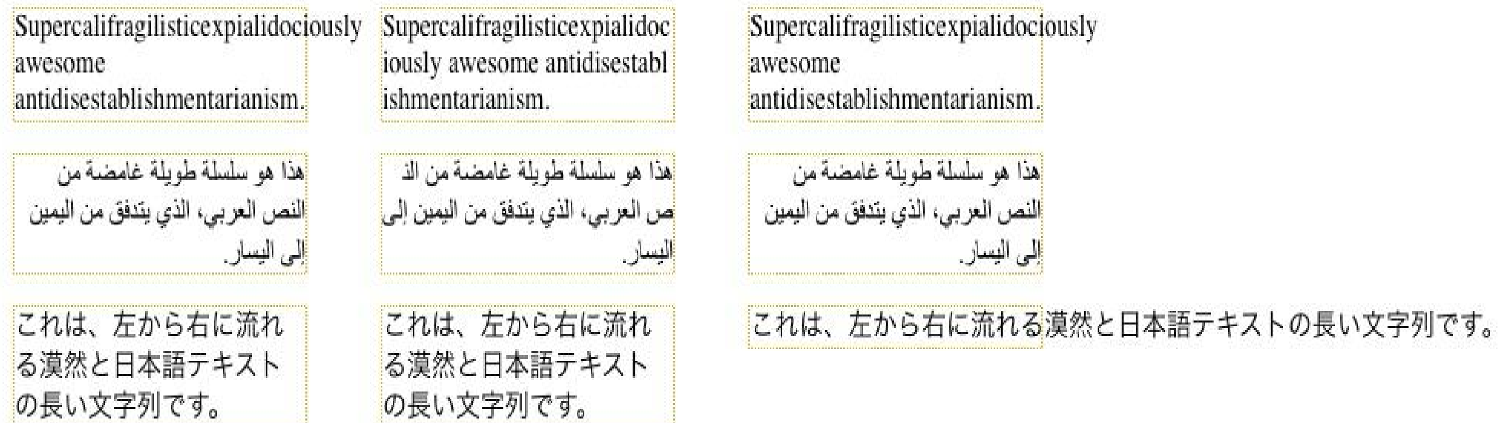


Figure 11-41. Altering word-breaking behavior

Table 11-2. Word-breaking behavior

Value	Non-CJK	CJK	Hyphenation permitted
`normal`	As usual	As usual	Yes
`break-all`	After any character	After any character	No
`keep-all`	As usual	Around sequences	Yes

As noted previously, the value `break-word` has been deprecated, although it supported by all known browsers as of mid-2022. When used, it has the same effect as `{word-break: normal; overflow-wrap: anywhere;}`, even if `overflow-wrap` has a different value. (We'll cover `overflow-wrap` in an upcoming section.)

Line breaking

If your interests run to CJK text, then in addition to `word-break` you will also want to get to know `line-break`.

LINE-BREAK

Values	`auto` | `loose` | `normal` | `strict` | `anywhere`
Initial value	`auto`
Applies to	All elements
Computed value	As specified
Inherited	Yes
Animatable	Yes

As we just saw, `word-break` can affect how lines of text are soft wrapped in CJK text. The `line-break` property also affects such soft wrapping, specifically how wrapping is handled around CJK-specific symbols and around non-CJK punctuation (such as exclamation points, hyphens, and ellipses) that appears in text declared to be CJK.

In other words, `line-break` applies to certain CJK characters all the time, regardless of the content's declared language. If you throw some CJK characters into a paragraph of English text, `line-break` will still apply to them, but not to anything else in the text. Conversely, if you declare content to be in a CJK language, `line-break` will continue to apply to those CJK characters *plus* a number of non-CJK characters within the CJK text. These include punctuation marks, currency symbols, and a few other symbols.

There is no authoritative list of which characters are affected and which are not, but the specification provides a list of recommended symbols and behaviors around those symbols.

The default value `auto` allows user agents to soft wrap text as they like, and more importantly lets UAs vary the line breaking they do based on the situation. For example, the UA can use looser line-breaking rules for short lines of text and stricter rules for long lines. In effect, `auto` allows the user agent to switch between the `loose`, `normal`, and `strict` values as needed, possibly even on a line-by-line basis within a single element.

You can perhaps infer that those other values have the following general meanings:

`loose`

: This value imposes the "least restrictive" rules for wrapping text, and is meant for use when line lengths are short, such as in newspapers.

`normal`

: This value imposes the "most common" rules for wrapping text. What exactly "most common" means is not precisely defined, though there is the aforementioned list of recommended behaviors.

`strict`

This value imposes the “most stringent” rules for wrapping text. Again, this is not precisely defined.

anywhere

This value creates a line-breaking opportunity around every typographic unit, including white space and punctuation marks. A soft wrap can even happen in the middle of a word, and hyphenation is not applied in such circumstances.

Wrapping Text

After all that information about hyphenation and soft wrapping, what happens when text overflows its container anyway? That’s what `overflow-wrap` addresses.

Originally called `word-wrap`, the `overflow-wrap` property applies to inline elements, setting whether the browser should insert line breaks within otherwise unbreakable strings in order to prevent text from overflowing its line box. In contrast to `word-break`, `overflow-wrap` will only create a break if an entire word cannot be placed on its own line without overflowing.

OVERFLOW-WRAP

Values	`normal` | `break-word` | `anywhere`
Initial value	`normal`
Applies to	All elements
Computed value	As specified
Inherited	Yes
Animatable	Yes

This property is less straightforward than it first appears, because its primary effect is to change how word wrapping and minimum-content sizing (which we haven’t even had a chance to discuss yet) interact in trying to avoid overflow at the ends of text lines.

NOTE

Note that `overflow-wrap` can only operate if the value of `white-space` allows line wrapping. If it does not (e.g., with the value `pre`), then `overflow-wrap` has no effect.

If the default value of `normal` is in effect, then wrapping happens as normal; which is to say, word wrapping between words or as directed by the language. If a word is longer than the width of the element containing it, then it will “spill out” of the element box, just like on the classic CSS IS AWESOME coffee mug. (Google it if you haven’t seen it before. It’s worth the chuckle.)

If the value `break-word` is applied, then wrapping can happen in the middle of words, with no hyphen placed at the site of the wrapping, but this will happen so that line lengths will be as wide as the element's width. In other words, if the `width` property of the element is given the value `min-content`, the "minimum content" calculations will assume that content strings must be as long as possible.

By contrast, when `anywhere` is set, the "minimum content" calculations will take line-wrapping opportunities into account. This means, in effect, that the minimum content width will be the width of the widest character in the element's content. Only when two skinny characters are next to each other will they have a chance to be on the same line together, and in a monospace font, every line of text will be a single character. Figure 11-42 illustrates the difference between these three values.

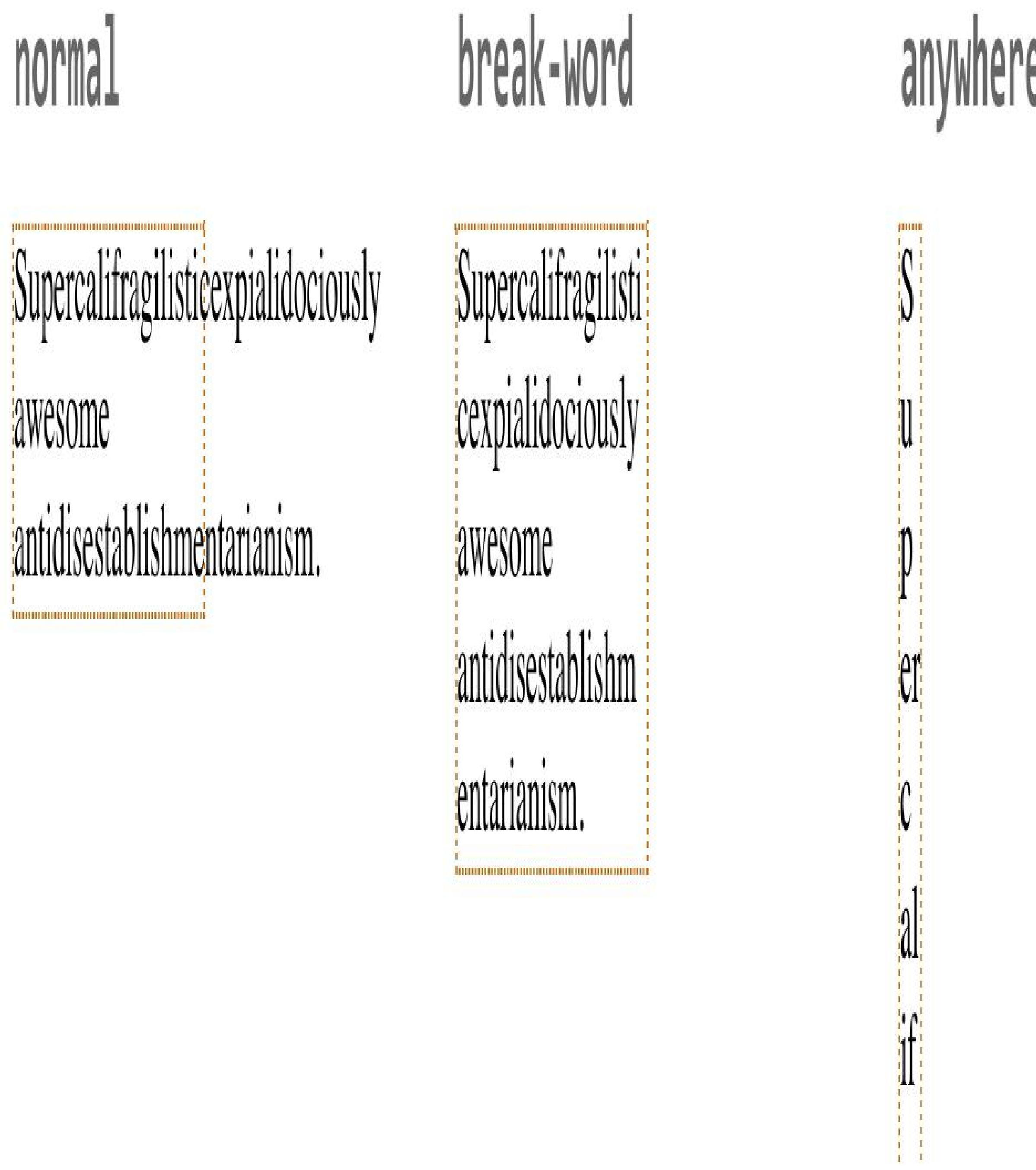


Figure 11-42. Overflow wrapping for `width: min-content`

If the value of `width` is something other than `min-content`, then `break-word` and `anywhere` will have the same results. Really, the only difference between the two values is with `anywhere`, soft wrap opportunities introduced by the word break are considered when calculating min-content intrinsic sizes. With `break-word`, they are not considered.

While `overflow-wrap: break-word` may appear very similar to `word-break: break-all`, they are not the same thing. To see why, compare the second box in Figure 11-42 to the top middle box in Figure 11-41. As it shows, `overflow-wrap` only kicks in if content actually overflows; thus, when there is an opportunity to use whitespace in the source to wrap lines, `overflow-wrap` will take it. By contrast, `word-break: break-all` will cause wrapping when content reaches the wrapping edge, regardless of any whitespace that comes earlier in the line.

Once upon a time there was a property called `word-wrap` that did exactly what `overflow-wrap` does. The two are so identical that the specification explicitly states that user agents "must treat `word-wrap` as an alternate name for the `overflow-wrap` property, as if it were a shorthand of `overflow-wrap`."

WARNING

As of mid-2022, WebKit browsers did not support the `anywhere` value for `overflow-wrap`.

Writing Modes

Earlier, we discussed inline direction, we introduced the topic of reading direction. We've already seen numerous benefits of including the `lang` attribute in your HTML, from being able to style based on language selectors, to allowing the user agent to hyphenate. Generally, you should let the user agent handle the direction of text based on the laguage attribute, but CSS does provide properties for the rare occassions when an override is necessary.

Setting Writing Modes

The property used for specifying one of five available writing modes is, of all things, `writing-mode`. This property sets the block flow direction of the element, which determines how boxes are stacked together.

WRITING-MODE

Values	`horizontal-tb` | `vertical-rl` | `vertical-lr` | `sideways-rl` | `sideways-lr`
Initial value	`horizontal-tb`
Applies to	All elements except table row groups, table column groups, table rows, table columns, ruby base containers, and ruby annotation containers
Computed value	As specified
Inherited	Yes
Animatable	Yes

The default value, `horizontal-tb` , means “a horizontal inline direction, and a top-to-bottom block direction.” This covers all Western and some Middle Eastern languages, which may differ in the direction of their horizontal writing. The other two values offer a vertical inline direction, and either a right-to-left or left-to-right block direction.

`sideways-rl` and `sideways-lr` take horizontal text and turn its flow “sideways,” with the direction the text runs either going right to left (for `sidewyas-rl`) or left to right (for `sideways-lr`). The difference between these values and the vertical values is that the text is turned whichever way is necessary to make the text read naturally.

All five values are illustrated in Figure 11-43.

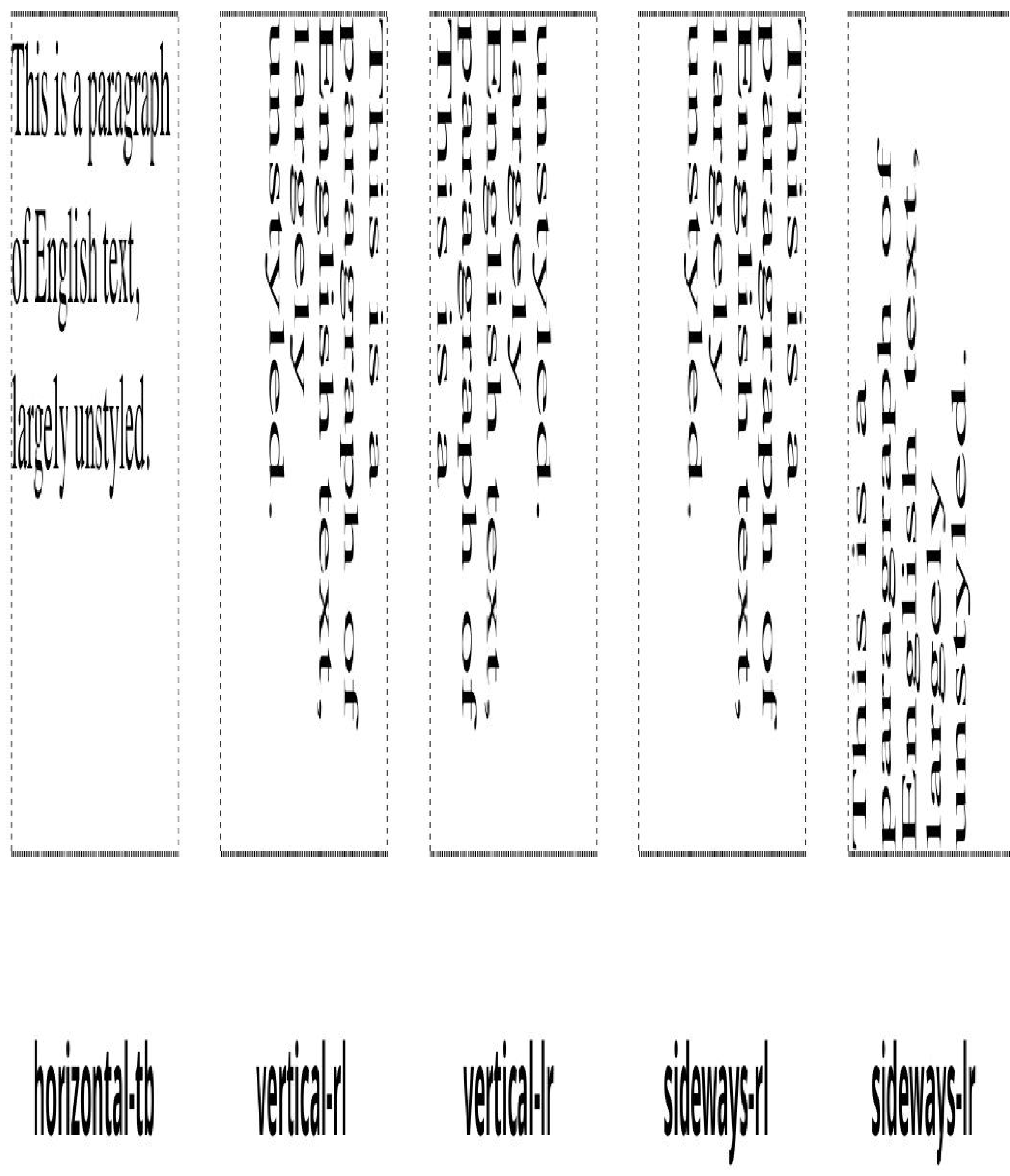


Figure 11-43. Writing modes

Notice how the lines are strung together in the two `vertical-` examples. If you tilt your head to the

right, the text in `vertical-rl` is at least readable. The text in `vertical-lr`, on the other hand, is difficult to read because it appears to flow from bottom to top, at least when arranging English text. This is not a problem in languages which actually use `vertical-lr` flow, such as forms of Japanese.

In vertical writing modes, the block direction is horizontal, which means vertical alignment of inline elements actually causes them to move horizontally. This is illustrated in Figure 11-44.

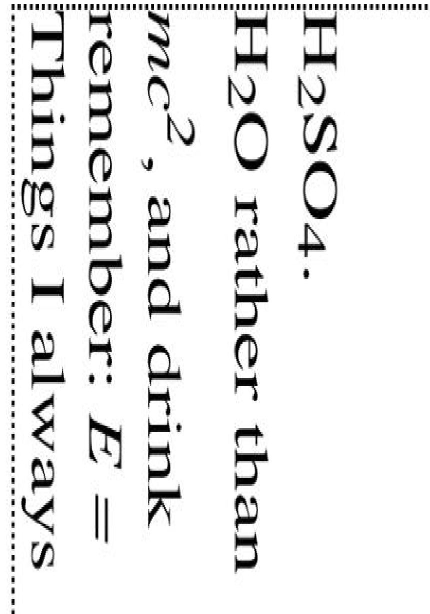


Figure 11-44. Writing modes and "vertical" alignment

All the super- and subscript elements cause horizontal shifts, both of themselves and the placement of the lines they occupy, even through the property used to move them is `vertical-align`. As described earlier, the vertical displacement is with respect to the line box, where the box's baseline is defined as horizontal—even when it's being drawn vertically.

Confused? It's OK. Writing modes are likely to confuse you, because it's such a different way of thinking *and* because old assumptions in the CSS specification clash with the new capabilities. If there had been vertical writing modes from the outset, `vertical-align` would likely have a different name —`inline-align` or something like that. (Maybe one day that will happen.)

Changing Text Orientation

Once you've settled on a writing mode, you may decide you want to change the orientation of characters within those lines of text. There are many reasons you might want to do this, not least of which are situations where different writing systems are commingled, such as Japanese text with English words or numbers mixed in. In these cases, `text-orientation` is the answer.

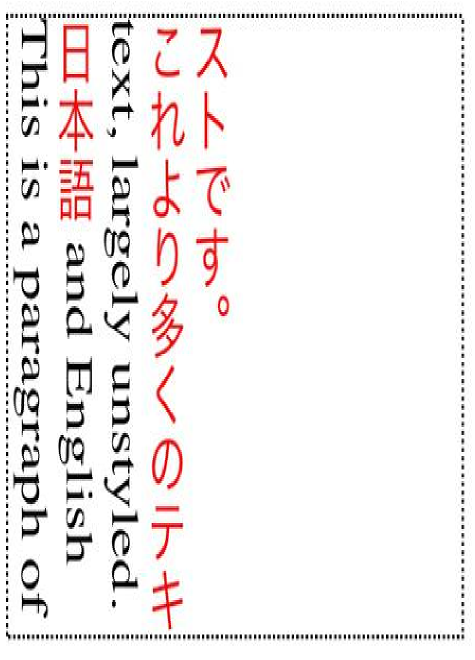
TEXT-ORIENTATION

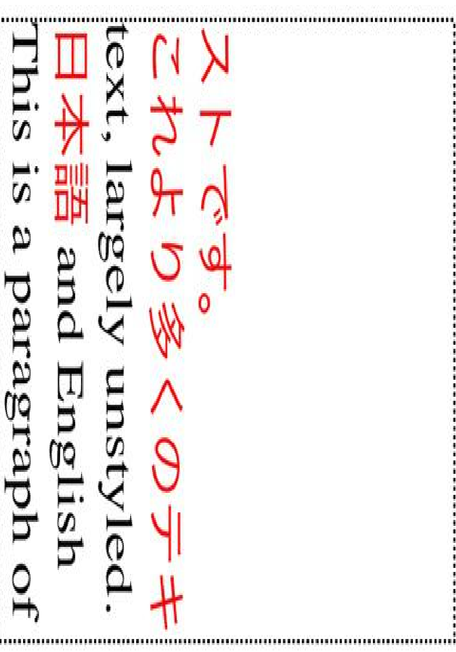
Values	`mixed` | `upright` | `sideways`
Initial value	`mixed`
Applies to	All elements except table row groups, table rows, table column groups, and table columns
Computed value	As specified
Inherited	Yes
Animatable	Yes

The effect of `text-orientation` is to affect how characters are oriented. What that means is best illustrated by the following styles, rendered in Figure 11-45:

```
.verts {writing-mode: vertical-lr;}
#one {text-orientation: mixed;}
#two {text-orientation: upright;}
#thr {text-orientation: sideways;}
```

This is a paragraph of 日本語 and English text, largely unstyled. これより多くのテキストです。





mixed upright sideways

Figure 11-45. Text orientation

Across the top of Figure 11-45 is a basically unstyled paragraph of mixed Japanese and English text. Below that, three copies of that paragraph, using the writing mode `vertical-lr`. In the first of the

three, `text-orientation: mixed` , writes the horizontal-script characters (the English) sideways, and the vertical-script characters (the Japanese) upright. In the second, all characters are `upright`, including the English characters. In the third, all characters are `sideways`, including the Japanese characters.

> **WARNING**
>
> As of mid-2022, `sideways` was not supported by Chromium browsers.

Combining characters

Only relevant to vertical writing modes, the `text-combine-upright` property enables displaying a supset of characters upright within vertical text. This can be useful when mixing languages or pieces of languages, such as embedding Arabic numerals in CJK text, but may have other applications.

TEXT-COMBINE-UPRIGHT

Values	`none` | `all` | [digits <*integer*>?]
Initial value	`none`
Applies to	non-replaced inline elements
Computed value	specified keyword, plus integer if *digits*
Inherited	Yes
Animatable	No
Note	For <*integer*> values, only the numbers 2, 3, and 4 are valid

Essentially, this property lets you say whether characters may sit next to each other horizontally while being part of a vertical line of text. Your choices are whether to allow this for all characters, or only for a few numeric digits.

Here's how it works: as a line of vertical text is laid out, the browser can consider whether the width of two characters, sitting next to each other, are less than or equal to the value of `1em` for the text. If so, they may be placed next to each other, effectively putting two characters into the space of one. If not, the first character is placed alone, and the process continues.

As of mid-2022, this can lead to characters being very, very squished. For an example, consider the following markup and CSS:

```
<div lang="zh-Hant">
<p>      </p>
<p class="combine">      </p>
```

```
  <p>117023       </p>
  <p class="combine">117023       </p>
  <p class="combine">  <span>117</span>  <span>0</span>  <span>23</span></p>
  <p>  <span class="combine">117</span>  <span
     class="combine">0</span>  <span class="combine">23</span></p>
  </div>

  p {writing-mode: vertical-rl;}
  .combine {text-combine-upright: all;}
```

All of the paragraphs are written using `writing-mode: vertical-rl`, but some are set to `text-combine-upright: all`, and others are not. The last paragraph is not set to `all`, but the `<span>` elements within it have been. The result is what's shown in Figure 11-46.

这是一些文本

这是117一些0文本23日

这是117一些0文本23日

这是117一些0文本23日

Figure 11-46. Various types of upright combination.

Lest you think a bug is at work there, the results were consistent across browsers (as of mid-2022).The second and fourth columns have every single character, whether Chinese ideographs or Arabic numerals, squished horizontally to fit on a single line.

A way around this is to break up the text with child elements, as shown in the fifth and sixth columns. In the first, numbers are surrounded with `<span>` elements, which break up the fitting process. This works as long as no run of text has too many characters; beyond two or three symbols, and the text becomes progressively more difficult to comprehend.

The sixth column shows a way to hack around the problem: only apply `text-combine-upright: all` to the `<span>` elements, which are already used to wrap the Arabic numerals, by giving each `<span>` a `class` value of `combine`. In that case, the `.combine` rule will only apply to the `<span>` elements, not all the text in the paragraph.

This is what the `digits` value is supposed to make possible without the need for all the extra markup. Theoretically, you could get the same result as that shown in the sixth column of Figure 11-46 by applying the following CSS to the paragraph that has no `<span>` elements in it:

```
p {writing-mode: vertical-rl; text-upright-combine: digits 4;}
```

Sadly, as of mid-2022, no browser supported this behavior, unless you count Internet Explorer 11 using the alternate property name `-ms-text-combine-horizontal`.

Declaring Direction

Harking back to the days of CSS2, there are a pair of properties that can be used to affect the direction of text by changing the inline baseline direction: `direction` and `unicode-bidi`. These should generally not be used, but are covered here in case you come across them in legacy code.

> **WARNING**
>
> The CSS specification explicitly warns *against* using `direction` and `unicode-bidi` in CSS when applied to HTML documents. To quote: “Because HTML [user agents] can turn off CSS styling, we recommend… the HTML `dir` attribute and `<bdo>` element to ensure correct bidirectional layout in the absence of a style sheet.” The properties are covered here because they may appear in legacy stylesheets.

DIRECTION

Values	`ltr` | `rtl`
Initial value	`ltr`
Applies to	All elements
Computed value	As specified
Inherited	Yes
Animatable	Yes

The `direction` property affects the writing direction of text in a block-level element, the direction of table column layout, the direction in which content horizontally overflows its element box, and the position of the last line of a fully justified element. For inline elements, direction applies only if the property `unicode-bidi` is set to either `embed` or `bidi-override` (See the following description of `unicode-bidi`).

Although `ltr` is the default, it is expected that if a browser is displaying right-to-left text, the value will be changed to `rtl`. Thus, a browser might carry an internal rule stating something like the following:

```
*:lang(ar), *:lang(he) {direction: rtl;}
```

The real rule would be longer and encompass all right-to-left languages, not just Arabic and Hebrew, but it illustrates the point.

While CSS attempts to address writing direction, Unicode has a much more robust method for handling directionality. With the property `unicode-bidi`, CSS authors can take advantage of some of Unicode's capabilities.

UNICODE-BIDI

Values	`normal` | `embed` | `bidi-override`
Initial value	`normal`
Applies to	All elements
Computed value	As specified
Inherited	No
Animatable	Yes

Here we'll simply quote the value descriptions from the CSS 2.1 specification, which do a good job of capturing the essence of each value:

`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 "left-to-right embedding" character (U+202A; for `direction: ltr`) or a "right-to-left embedding" character (U+202B; for `direction: rtl`) at the start of the element and a "pop directional formatting" character (U+202C) at the end of the element.

`bidi-override`

This creates an override for inline-level elements. For block-level elements, this creates an override for inline-level descendants not within another block. 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 "left-to-right override" character (U+202D; for `direction: ltr`) or "right-to-left override" character (U+202E; for `direction: rtl`) at the start of the element and a "pop directional formatting" character (U+202C) at the end of the element.

Summary

Even without altering the font face, there are many ways to change the appearance of text. There are classic effects such as underlining, but CSS also enables you to draw lines over text or through it, change the amount of space between words and letters, indent the first line of a paragraph (or other block-level element), align text in various ways, exert influence over the hyphenation and line breaking of text, and much more. You can even alter the amount of space between lines of text. There is also support in CSS for languages other than those that are written left-to-right, top-to-bottom. Given that so much of the web is text, the strength of these properties makes a great deal of sense.

About the Authors

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 also the founder of Complex Spiral Consulting, which counts among its clients America Online; Apple Computer, Inc.; Wells Fargo Bank; and Macromedia, which described Eric as "a critical partner in our efforts to transform Macromedia Dreamweaver MX 2004 into a revolutionary tool for CSS-based design."

Beginning in early 1994, Eric was the visual designer and campus web coordinator for the Case Western Reserve University website, where he also authored a widely acclaimed series of three HTML tutorials and was project coordinator for the online version of the *Encyclopedia of Cleveland History* and the *Dictionary of Cleveland Biography*, the first encyclopedia of urban history published fully and freely on the web.

Author of *Eric Meyer on CSS* and *More Eric Meyer on CSS* (New Riders), *CSS: The Definitive Guide* (O'Reilly), and *CSS2.0 Programmer's Reference* (Osborne/McGraw-Hill), as well as numerous articles for the O'Reilly Network, Web Techniques, and Web Review, Eric also created the CSS Browser Compatibility Charts and coordinated the authoring and creation of the W3C's official CSS Test Suite. He has lectured to a wide variety of organizations, including Los Alamos National Laboratory, the New York Public Library, Cornell University, and the University of Northern Iowa. Eric has also delivered addresses and technical presentations at numerous conferences, among them An Event Apart (which he cofounded), the IW3C2 WWW series, Web Design World, CMP, SXSW, the User Interface conference series, and The Other Dreamweaver Conference.

In his personal time, Eric acts as list chaperone of the highly active css-discuss mailing list, which he cofounded with John Allsopp of Western Civilisation, and which is now supported by *evolt.org*. Eric lives in Cleveland, Ohio, which is a much nicer city than you've been led to believe. For nine years he was the host of "Your Father's Oldsmobile," a big-band radio show heard weekly on WRUW 91.1 FM in Cleveland.

You can find more detailed information on Eric's personal web page.

How does someone get to be the author of *Flexbox in CSS*, *Transitions and Animations in CSS*, and *Mobile HTML5* (O'Reilly), and coauthor of *CSS3 for the Real World* (SitePoint)? For **Estelle Weyl**, the journey was not a direct one. She started out as an architect, used her master's degree in health and social behavior from the Harvard School of Public Health to lead teen health programs, and then began dabbling in website development. By the time Y2K rolled around, she had become somewhat known as a web standardista at *http://www.standardista.com*.

Today, she writes a technical blog that pulls in millions of visitors, and speaks about CSS3, HTML5, JavaScript, accessibility, and mobile web development at conferences around the world. In addition to sharing esoteric programming tidbits with her reading public, Estelle has consulted for Kodak Gallery, SurveyMonkey, Visa, Samsung, Yahoo!, and Apple, among others. She is currently the Open Web Evangelist for Instart Logic, a platform that helps make web application delivery fast and secure.

When not coding, she spends her time doing construction, striving to remove the last remnants of communal hippiedom from her 1960s-throwback home. Basically, it's just one more way Estelle is

working to bring the world into the 21st century.