sitepoint

CSS MASTER

THIRD EDITION

BY TIFFANY B. BROWN

ORGANIZED, EFFICIENT, POWERFUL—CSS DONE RIGHT!

CSS Master, 3rd Edition

Copyright © 2021 SitePoint Pty. Ltd.

Ebook ISBN: 978-1-925836-43-1

- **Technical Editor:** Rachel Andrew
- **Product Manager:** Simon Mackie
- **Editor:** Ralph Mason
- **Cover Designer:** Alex Walker

Notice of Rights

Notice of Liability

Trademark Notice

Published by SitePoint Pty. Ltd.

10-12 Gwynne St, Richmond, VIC, 3121
Australia
Web: www.sitepoint.com
Email: books@sitepoint.com

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit http://www.sitepoint.com/ to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, design, and more.

About the Author

Tiffany B. Brown is a freelance web developer based in Los Angeles, California. She has worked on the Web for nearly two decades with a career that includes media companies, marketing agencies, and government.

Brown was also part of the Digital Service Team at the United States Department of Veterans Affairs, the United States Digital Service, and the Opera Software Developer Relations team.

Brown is also a co-author of SitePoint's *Jump Start HTML 5*, and has contributed to Dev.Opera, A List Apart, SitePoint.com, and Smashing Magazine.

Preface

CSS has grown from a language for formatting documents into a robust language for designing web applications. Its syntax is easy to learn, making CSS a great entry point for those new to programming. Indeed, it's often the second language that developers learn, right behind HTML.

However, the simplicity of CSS syntax is deceptive. It belies the complexity of the box model, stacking contexts, specificity, and the cascade. It's tough to develop interfaces that work across a variety of screen sizes and with an assortment of input mechanisms. CSS mastery lies in understanding these concepts and how to mitigate them.

Mastering CSS development also means learning how to work with tools such as linters and optimizers. Linters inspect your code for potential trouble spots. Optimizers improve CSS quality, and reduce the number of bytes delivered to the browser. And, of course, there's the question of CSS architecture: which selectors to use, how to modularize files, and how to prevent selector creep.

CSS has also grown in its capabilities. Until recently, we had to use clunky methods such as float, or weighty JavaScript libraries, to create the kinds of layouts that are now possible with the Flexible Box, Multicolumn, and Grid layout modules. Three-dimensional effects were impossible—or required images—before the arrival of CSS transforms. Creating slide shows is now trivial thanks to Scroll Snap. We even have support for variables.

What's Changed in This Edition?

As with previous editions, writing this edition required careful consideration of what to include and what to exclude. The third edition restores and expands the "Selectors" chapter from the first edition. The "Layouts" chapter now includes a section on CSS Shapes, and a more comprehensive look at Flexible Box (aka Flexbox) layout.

This edition also adds two entirely new chapters. One covers the `scroll-behavior` property and the ins-and-outs of CSS Scroll Snap. The second covers CSS visual effects: blend modes, filter effects, clipping and masking.

But *CSS Master* isn't a comprehensive guide to CSS. CSS is a dense and ever-expanding topic with lots of nooks and crannies. Trying to cover it all is a massive task. Instead, my hope is that you'll come away from this book with a better sense of how CSS works—particularly its trickier bits—and how to write it well.

Who Should Read This Book?

This book is for intermediate-level CSS developers, as it assumes a fair amount of experience with HTML and CSS. No time is spent covering the basics of CSS syntax. Coverage of CSS concepts such as the box model and positioning are included to illuminate concepts for the experienced developer, but this coverage is not meant as an introduction for beginners. Experience with JavaScript is helpful, but not necessary.

Conventions Used

Code Samples

Code in this book is displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park.
The birds were singing and the kids were all back at school.
</p>
```

You'll notice that we've used certain layout styles throughout this book to signify different types of information. Look out for the following items.

Tips, Notes, and Warnings

Hey, You!

Tips provide helpful little pointers.

Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always ...

... pay attention to these important points.

Watch Out!

Warnings highlight any gotchas that are likely to trip you up along the way.

Supplementary Materials

- https://www.sitepoint.com/community/ are SitePoint's forums, for help on any tricky problems.
- This book's code archive is available on GitHub.
- **books@sitepoint.com** is our email address, should you need to contact us to report a problem, or for any other reason.

Chapter 1: Selectors

Understanding selectors is key to writing maintainable, scalable CSS. Selectors are the mechanism by which CSS rules are matched to elements. There are various ways to do this, and you're probably familiar with most of them. Element type, class name, ID, and attribute selectors are all well supported and widely used.

In this chapter, we'll firstly review the types of selectors. Then we'll look at the current browser landscape for CSS selectors, with a focus on newer selectors defined by the Selectors Level 3 and Selectors Level 4 specifications.

This chapter stops short of being a comprehensive look at all selectors, as that could take up a whole book in itself. Instead, we'll focus on selectors with good browser support that are likely to be useful in your current work. Some material may be old hat, but it's included for context.

Types of Selectors

Selectors can be grouped into four basic types: simple, compound, combinator, and complex.

Simple selectors are the oldest form of CSS selector, and may be the type used most often. Simple selectors specify a single condition for matching elements. The universal selector (`*`) is a simple selector. So are type (or element) selectors such as `p` and pseudo-element selectors such as `::first-letter`. Attribute selectors such as `[hidden]`, class selectors such `.message-error`, and ID selectors such as `#masthead` also fall into this category.

Compound selectors, such as `p:last-child` or `.message.error`, are a sequence of simple selectors that reflect a set of simultaneous conditions to meet when applying rules to an element. In other words, `.message.error` will match `<div class="message error">`, but not `<div class="message">` or `<div class="error">`.

Combinator selectors express a relationship between elements. There are four:

- the **descendant combinator**, as in `article p`
- the **child combinator** (`>`), as in `.sidebar > h2`
- the **adjacent sibling combinator** (`+`), as in `ul + p`
- the **general sibling combinator** (`~`), as in `p ~ figure`

Rules are applied to the right-most element in a combinator selector when it fits the condition indicated by the combinator. We'll discuss combinator selectors in detail later in the chapter.

Lastly, there are complex selectors. **Complex selectors** consist of one or more compound selectors separated by a combinator. The selector `ul:not(.square) > a[rel=external]` is an example of a complex selector.

Selectors can be grouped into what's known as a **selector list** by separating them with a comma. Selector lists apply styles to elements that match *any* of the selectors in the list. For example, the rule `article, div { padding: 20px; }` adds 20 pixels of padding to *both* `<article>` and `<div>` elements.

Knowing what kind of selectors you're working with will help you grasp one of the more confusing aspects of CSS: **specificity**. Keeping specificity low increases the reusability of your CSS rules. A selector such as `#menu > .pop-open` means that you can only use the `.pop-open` pattern when it's a direct descendant of `#menu`, even if there are similar interactions elsewhere in your project.

We'll return to specificity in Chapter 2, "CSS Architecture and Organization". For the rest of this chapter, however, we'll discuss specific groups of selectors: combinators, attribute selectors, pseudo-elements, and pseudo-classes.

Combinators

As we saw above, a combinator is a character sequence that expresses a relationship between the selectors on either side of it. Using a combinator creates a complex selector. Using complex selectors can, in some cases, be the most concise way to define styles.

In the previous section, we listed the four combinators: descendant (via whitespace), child (>), adjacent sibling (+), and general sibling (~).

Let’s illustrate each of these combinators. We’ll use them to add styles to the HTML form shown below.

The form pictured above was created using the following chunk of HTML:

```
<form method="GET" action="/processor">
    <h1>Buy Tickets to the Web Developer Gala</h1>
    <p>Tickets are $10 each. Dinner packages are an extra $5.
All fields are required.</p>
    <fieldset>
        <legend>Tickets and Add-ons</legend>

        <p>
            <label for="quantity">Number of Tickets</label>
            <span class="help">Limit 8</span>
            <input type="number" value="1" name="quantity"
id="quantity" step="1" min="1" max="8">
    </p>

        <p>
            <label for="quantity">Dinner Packages</label>
            <span class="help">Serves 2</span>
            <input type="number" value="1" name="quantity"
id="quantity" step="1" min="1" max="8">
        </p>

    </fieldset>
    <fieldset>
        <legend>Payment</legend>
        <p>
            <label for="ccn">Credit card number</label>
            <span class="help">No spaces or dashes, please.
</span>
            <input type="text" id="ccn" name="ccn"
placeholder="372000000000008" maxlength="16" size="16">
        </p>
        <p>
            <label for="expiration">Expiration date</label>
            <span class="help"><abbr title="Two-digit
month">MM</abbr>/<abbr title="Four-digit Year">YYYY</abbr>
</span>
            <input type="text" id="expiration"
name="expiration" placeholder="01/2018" maxlength="7"
size="7">
        </p>

    </fieldset>
    <fieldset>
        <legend>Billing Address</legend>
        <p>
            <label for="name">Name</label>
            <input type="text" id="name" name="name"
placeholder="ex: John Q. Public" size="40">
        </p>
        <p>
            <label for="street_address">Street
Address</label>
            <input type="text" id="name" name="name"
placeholder="ex: 12345 Main Street, Apt 23" size="40">
        </p>

        <p>
            <label for="city">City</label>
            <input type="text" id="city" name="city"
placeholder="ex: Anytown">
        </p>

        <p>
            <label for="state">State</label>
            <input type="text" id="state" name="state"
placeholder="CA" maxlength="2" pattern="[A-W]{2}" size="2">
        </p>

        <p>
            <label for="zip">ZIP</label>
            <input type="text" id="zip" name="zip"
placeholder="12345" maxlength="5" pattern="0-9{5}" size="5">
        </p>
    </fieldset>

    <button type="submit">Buy Tickets!</button>
</form>
```

The Descendant Combinator

You’re probably quite familiar with the descendant combinator. It’s been around since the early days of CSS (though it lacked a proper name until CSS2.1). It’s widely used and widely supported.

The descendant combinator is simply a whitespace character. It separates the parent selector from its descendant, following the pattern `A B`, where `B` is an element contained by `A`. Let’s add some CSS to our markup from above and see how this works:

```
form h1 {
    color: hsl(231, 48%, 48%);
}
```

We’ve just changed the color of our form title, the result of which can be seen below.

Let’s add some more CSS, this time to increase the size of our pricing message (“Tickets are $10 each”). We’ll also make it hot pink:

```
form p {
    font-size: 36px;
    color: #c2185b;
}
```

There’s a problem with this selector, however, as you can see in the image below. Our selector is too broad.

We’ve actually increased the size of the text in *all* of our form’s paragraphs, which isn’t what we want. How can we fix this? Let’s try the child combinator.

The Child Combinator

In contrast to the descendant combinator, the child combinator (`>`) selects only the *immediate children* of an element. It follows the pattern `A > B`, matching any element `B` where `A` is the immediate ancestor.

If elements were people, to use an analogy, the child combinator would match the child of the mother element. But the descendant combinator would also match her grandchildren, and great-grandchildren. Let’s modify our previous selector to use the child combinator:

```
form > p {
    font-size: 36px;
}
```

Now only the direct children of `form` are affected, as shown in the image below.

The Adjacent Sibling Combinator

With the adjacent sibling combinator (`+`), we can select elements that follow each other and have the same parent. It uses the pattern `A + B`. Styles are applied to `B` elements that are *immediately* preceded by `A` elements.

Let’s go back to our example. Notice that, in the Billing Address section, our labels and inputs sit next to each other. That means we can use the adjacent sibling combinator to make them sit on separate lines:

```
label + input {
    display: block;
    clear: both;
}
```

You can see the results in the image below.

You can see in the image above that some of our labels remain on the same line as their input fields. In those instances, there's a `<span>` element between the `<label>` and `<input>` elements, meaning they're not adjacent siblings. To match sibling elements that aren't adjacent, we'll have to use the general sibling combinator (as we'll see in the next section).

Let's look at another example that combines the universal selector (`*`) with a type selector:

```
* + fieldset {
    margin: 5em 0;
}
```

This example adds a `5em` margin to the top and bottom of every `<fieldset>` element, as shown in the image below.

Since we're using the universal selector, there's no need to worry about whether the previous element is another `<fieldset>` or `<p>` element.

More Uses of the Adjacent Sibling Selector

Heydon Pickering explores more clever uses of the adjacent sibling selector in his article "Axiomatic CSS and Lobotomized Owls".

The General Sibling Combinator

With the general sibling combinator (`~`) we can select elements that share the same parent without considering whether they're adjacent. Given the pattern `A ~ B`, this selector matches all `B` elements that are preceded by an `A` element.

Let's look at the Number of Tickets field again. Its markup looks like this:

```
<p>
    <label for="quantity">Number of Tickets</label>
    <span class="help">Limit 8</span>
    <input type="number" value="1" name="quantity"
id="quantity" step="1" min="1" max="8">
</p>
```

Our `<input>` element follows the `<label>` element, but there's a `<span>` element in between. The adjacent sibling combinator will fail to work here. Let's change our adjacent sibling combinator to a general sibling combinator:

```
label ~ input {
    display: block;
}
```

Now all of our `<input>` elements sit on a separate line from their `<label>` elements, as seen in the following image.

Because the general sibling combinator matches any subsequent sibling, you'll want to use it judiciously. Consider the markup and CSS below:

```
<!DOCTYPE html>
   <html lang="en-US">
   <head>
        <meta charset="utf-8">
        <title>In This Essay, I Will</title>
        <style>
        h1 ~ p {
          background: yellow
        }
        h2 + p {
          outline: 5px dotted #009;
        }
        </style>
    </head>
    <body>
        <h1>In This Essay, I Will</h1>

        <p>Lorem ipsum dolor sit amet, consectetur adipiscing
elit. Fusce odio leo, sollicitudin vel mattis eget.…</p>

        <p>Nulla sit amet neque eleifend diam aliquam
rhoncus. Donec id congue est. Aliquam sagittis euismod
tristique.…</p>

        <h2>Show how the general sibling combinator
works</h2>

        <p>Proin condimentum elit sapien, ut tempor nisl
porta quis. …</p>
    </body>
</html>
```

Here we've used the general sibling combinator with an `<h1>` element. As a result, *every* paragraph element that follows an `<h1>` element has a yellow background. This includes the paragraph that follows the `<h2>` heading, as shown below.

If you have control over the document's markup, I'd recommend using a class selector instead of the general sibling combinator. The general sibling combinator makes it too easy to accidentally style more elements than you intended to.

Attribute Selectors

Introduced with the CSS Level 2 Specification, attribute selectors make it possible to style elements based on the presence of an attribute, such as `[controls]` for a media player, or `[disabled]` for a form field.

You can also use attribute selectors to match elements based on the presence of an attribute and its value. For example, to style submit buttons, you might use the following CSS:

```
[type=submit] {
   background: seagreen;
   border: 0;
   border-radius: 1000px;
   color: #fff;
   font-size: 18pt;
   padding: 10px 20px;
}
```

There are also several attribute selectors for partially matching attribute values, as well as substrings. Being able to target partially matching attribute values is one of my favorite features of CSS selectors. When used thoughtfully, they can reduce the number of rules and declarations you need to write. We'll look at them shortly. Most of the attribute selectors we'll cover are old hat. I've included them, however, for context and completeness.

A Note About Quotes

Quoting the values of attribute selectors is optional in most cases. Both `[type=checkbox]` and `[type="checkbox"]` are valid and well-supported syntaxes for attribute selectors. Use quotes when the attribute's value contains spaces or punctuation characters, such as `[class="piechart animated"]`, `[data-action="modal:close"]` or `[id='section-2.2']`.

Matching Space-separated Attribute Values

Although we can select elements based on an attribute value, as discussed above, a selector such as `[rel=external]` won't match `<a href="/" rel="external citation">`, because the `rel` value isn't exactly `external`. Instead, we need to use a selector that can accommodate space-separated values, which takes the form of `[att~=val]`.

The space-separated attribute value selector matches elements with the attribute (`att`) and a list of values, one of which is `val`. This can be any attribute that accepts space-separated values, including `class` or `data-*`.

Space-separated lists of attributes are admittedly uncommon. They're sometimes used with the `rel` attribute and [microformats]() to describe relationships between people and documents. As an example, we might mark up external links like so:

```
<nav>
   <a href="http://bob.example.com/" rel="external
friend">Bob</a>
   <a href="http://maria.example.com/" rel="external
acquaintance">Mar&iacute;a</a>
   <a href="http://ifeoma.example.com/" rel="external
colleague">Ifeoma</a>
</nav>
```

We can then use this presence-based attribute selector to match links that contain `friend` as one of its attribute values:

```
[rel~=friend] {
    font-size: 2em;
    background: #eee;
    padding: 4px;
    text-decoration: none;
    border-bottom: 3px solid #ccc;
}
[rel~=friend]:link, [rel~=friend]:visited {
    color: #34444C;
}
[rel~=friend]:hover{
    background: #ffeb3b;
    border-color: #ffc107;
}
```

The result of this is shown in the image below.

Matching Hyphenated Attribute Values

One of the more interesting tasks we can do with attribute selectors is to use `[attr|=val]` to match the first part of an attribute value before the first hyphen. For example, `[lang|=en]` would match an element like `<p lang="en-US">`.

The main purpose of this selector is for working with languages and language codes, such as `en-US` and `es-MX`.

Let's say we have some markup like this:

```
<p lang="fr-FR"><q>Tout le monde</q></p>
<p><q>All the world</q>, or <q>Everyone</q></p>
```

We can italicize our French text and add language-appropriate angle quotes (« and ») to either side of it:

```
[lang|="fr"] {
    font-style: italic;
}
[lang|="fr"] q:before{
    content: '\00AB'; /* Left angle quote */
}
[lang|="fr"] q:after{
    content: '\00BB';  /* Right angle quote */
}
```

What's cool about this selector is that it works even if there's no hyphen. So the styles above would also apply to `<p lang="fr">`. And we can further limit the scope of these selectors by adding an element selector, such as `p[lang|="fr"]`.

This selector isn't limited to language codes. We can use it with any hyphenated attribute value. Consider the following markup:

```
<article class="promo">
    <h3>U.S. Meets Climate Goals 5 Years Early</h3>
    <p>Lorem ipsum dolor sit amet, consectetur adipisicing
....</p>
</article>

<article class="promo-entertainment">
    <h3>Prince-Bythewood, Duvernay Among Nominees At
Oscars</h3>
    <p>Lorem ipsum dolor sit amet, consectetur adipisicing
....</p>
</article>

<article class="promo-sports">
    <h3>New York Knicks win NBA title</h3>
    <p>Lorem ipsum dolor sit amet, consectetur adipisicing
....</p>
</article>

<article class="promo-business">
    <h3>GrubDash to Hire 3,000 Drivers, Offer Benefits</h3>
    <p>Lorem ipsum dolor sit amet, consectetur adipisicing
....</p>
</article>
```

These are all article promos or teasers. They share some of the same visual characteristics and behavior, along with classes prefixed with `promo`. Here, too, we can use the hyphenated attribute selector to match these class names:

```
[class|="promo"] {
    border-top: 5px solid #4caf50;
    color: #555;
    line-height: 1.3;
    padding-top: .5em;
}
[class|="promo"] h3 {
    color: #000;
    font-size: 1.2em;
    margin:0;
}
[class|="promo"] p {
    margin: 0 0 1em;
}
```

Follow this up with specific border colors for each section type, and you'll achieve something along the lines of the layout pictured below.

We can also use this selector with ID names. For example, `[id|=global]` would match `#global-footer`, `#global-menu`, and so on.

Matching Attribute Values by Substring

We can also select elements when the attribute values match a particular substring using `[att^=val]`, `[att$=val]` and `[att*=val]`.

The `^=` selector matches a substring at the *beginning* of an attribute value. For example, think about links using `tel:` (non-standard) or `mailto:`. Since they behave differently from other hyperlinks, it makes sense to style them differently just as a hint to the user. Take a “Call this business” link:

```
<a href="tel:+14045555555">Call this business</a>
```

We can select this and other `tel:` links by using `[href^="tel:"]`. Let’s add some declarations:

```
[href^="tel:"] {
    background: #2196f3 url(../../Images/phone-icon.svg) 10px
center / 20px auto no-repeat;
    border-radius: 100px;
    padding: .5em 1em .5em 2em;
}
```

You can see the result in the image below.

The `$=` selector matches a substring at the *end* of an attribute value. If, for example, we wanted to give a special color to PDF file links, we could use `a[href$=".pdf"]`:

```
a[href$=".pdf"] {
    color: #e91e63;
}
```

This selector would also be handy for matching elements whose attribute values end with the same suffix. For example, you could match both `<aside class="sports-sidebar">` and `<aside class="arts-sidebar">` with `[class$=sidebar]`.

The `*=` selector matches a substring *in any position* within the attribute value. Using the selector `[class*=sidebar]`, we could select an element with a class of `sports-sidebar-a`, *along with* elements with the classes `sports-sidebar` and `arts-sidebar`.

Matching Attribute Values by Case

CSS is, for the most part, a case-insensitive language. Both `color: tomato` and `COLOR: TOMATO` do the same thing. Both `p {…}` and `P {…}` will style paragraphs in HTML, whether the HTML uses `<p>` or `<P>`. The same applies with attribute names, where `[href]` and `[HREF]` will both match `href="…"` and `HREF="…"`.

However, the same doesn't apply to attribute *values*. Letter case matters with these. In the following markup, the ID attribute for our `<div>` tag mixes uppercase and lowercase letters:

```
<div id="MixedCaseIDExample">
    The identifier for this tag mixes uppercase and lowercase
letters.
</div>
```

To style the `<div>`, we might use its ID selector—that is, `#MixedCaseIDExample`. But we'd have to use it exactly as it appears in the HTML. Using `#mixedcaseidexample`, for example, wouldn't cut it.

But there is an alternative. We could instead use case-insensitive attribute matching. It's a feature defined by the Selectors Level 4 specification.

Case-insensitive attribute matching uses the `i` flag to indicate that these styles should be applied to any case combination:

```
[id=mixedcaseidexample i] {
  color: blue;
}
```

Now our selector will match the ID attribute whether its value is `mixedcaseidexample`, `MixedCaseIDExample`, or `mIxEdCaSeIdExAmPlE`.

In some cases, you may want to enforce case-sensitive value matching. To enforce case-sensitive matching, use the `s` flag:

```
[id="mixedcaseidexample" s] {
  color: orange;
}
```

The `s` flag matches `#mixedcaseidexample`, but not `#MixedCaseIDExample` or `#mIxEdCaSeIdExAmPlE`.

Pseudo-classes and Pseudo-elements

Most of the new selectors added in CSS3 and CSS4 are not attribute selectors at all. They're pseudo-classes and pseudo-elements.

Though you've probably used pseudo-classes and pseudo-elements in your CSS, you may not have thought about what they are or how they differ from each other.

Pseudo-classes let us style objects based on information—such as their state—that's distinct from the document tree, or that can't be expressed using simple selectors. For example, an element can only have a hover or focus state once the user interacts with it. With the `:hover` and `:focus` pseudo-classes, we can define styles for those states. Otherwise, we'd have to rely on scripting to add and remove class names.

Pseudo-elements, on the other hand, let us style elements that aren't directly present in the document tree. HTML doesn't define a `firstletter` element, so we need another way to select it. The `::first-letter` pseudo-element gives us that capability.

Beware of Universal Selection

Using pseudo-classes and pseudo-elements without a simple selector is the equivalent of using them with the universal selector. For a selector such as `:not([type=radio])`, every element that lacks a `type` attribute and value of `radio` will match—including `<html>` and `<body>`. To prevent this, use

`:not()` as part of a compound selector, such as with a class name or element, as in `p:not(.error)`.

In the same way, using class names, IDs and attribute selectors on their own applies them universally. For example, `.warning` and `[type=radio]` are the same as `*.warning` and `*[type=radio]`.

Pseudo-elements

The CSS Pseudo-elements Module Level 4 specification clarifies behavior for existing pseudo-elements and defines several new ones. We'll focus on the ones that currently have browser support:

- `::after` inserts additional generated content after the content of an element
- `::before` inserts additional generated content before the content of an element
- `::first-letter` selects the first letter of an element
- `::first-line` selects the first line of an element
- `::marker` styles bullets and numbers for list items and the `<summary>` element
- `::placeholder` styles placeholder text for form controls using the `placeholder` attribute
- `::selection` styles text selected by the cursor

Of these, `::first-letter`, `::first-line`, `::selection`, `::marker` and `::placeholder` affect content that's part of the document source. The `::before` and `::after` pseudo-elements, on the other hand, inject content into a document. Let's look at each of these pseudo-elements more closely.

`::before` and `::after`

Most pseudo-elements allow us to select content that's already part of the document source—that is, the HTML you authored—but that's not specified by the language. But `::before` and `::after` work differently. These pseudo-elements add generated content to the document tree. This content doesn't exist in the HTML source, but it's available visually.

Why would you want to use generated content? You might, for example, want to indicate which form fields are required by adding content after their label:

```
/* Apply to the label element associated with a required
field */
.required::after {
    content: ' (Required) ';
    color: #c00;
    font-size: .8em;
}
```

Required form fields use the `required` HTML property. Since that information is already available to the DOM, using `::before` or `::after` to add helper text is supplemental. It isn't critical content, so it's okay that it's not part of the document source.

Generated Content and Accessibility

Some screen reader and browser combinations recognize and read generated content, but most don't. You can't be sure that content generated using `::before` or `::after` will be available to assistive technology users. You can read more about this in Leonie Watson's piece "Accessibility support for CSS generated content".

Another use case for `::before` or `::after` is adding a prefix or suffix to content. For example, the form mentioned above might include helper text, as shown here:

```
<form method="post" action="/save">
    <fieldset>
        <legend>Change Your Password</legend>
        <p>
            <label for="password">Enter a new
password</label>
            <input type="password" id="password"
```

```
name="password">
        </p>
        <p>
            <label for="password2">Retype your
password</label>
            <input type="password" id="password2"
name="password2">
        </p>
        <p class="helptext">Longer passwords are stronger.
</p>
        <p><button type="submit">Save changes</button></p>
    </fieldset>
</form>
```

Let’s enclose our helper text in red parentheses using `::before` and `::after`:

```
.helptext::before,
.helptext::after {
    color: hsl(340, 82%, 52%);
}
.helptext::before {
    content: '( ';
}
.helptext::after {
    content: ')';
}
```

The result is shown below.

Both `::before` and `::after` behave similarly to other descendant elements. They inherit the inheritable properties of their parent, and are contained within it. They also interact with other element boxes as though they were true elements.

One Pseudo-element per Selector

Currently, only one pseudo-element is allowed per selector. A selector such as `p::first-line::before` is invalid and unsupported.

This means that we can use `::before` and `::after` with CSS Grid and Flexbox. One use case is decorated headings, such as the one shown below.

The CSS required to create this heading is as follows:

```
h1 {
    display: grid;
    grid-template-columns: 1fr auto 1fr;
    gap: 3rem;
}
h1::before,
h1::after {
    content: '\00a0';
    background: url('decoration.svg') repeat-x center / 50%
auto;
}
```

You can read more about CSS Grid and Flexbox layout in Chapter 5, “Layouts”.

Creating Typographic Effects with `::first-letter`

While the `::before` and `::after` pseudo-elements inject content, `::first-letter` works with content that exists as part of the document source. With `::first-letter`, we can create initial letter effects, such as drop caps, as you might see in a magazine or book layout.

Initial and Drop Caps

An **initial capital** is an uppercase letter at the start of a block of text that’s set in a larger font size than the rest of the body copy. A **drop capital** (or drop cap) is similar to an initial capital, but is inset into the first paragraph by at least two lines.

This CSS snippet adds an initial capital letter to every `<p>` element in our document:

```
p::first-letter {
    font-family: serif;
    font-weight: bold;
    font-size: 3em;
    font-style: italic;
    color: #3f51b5;
}
```

The result is shown below.

As you may have noticed from the image above, `::first-letter` will affect the `line-height` of the first line if you've set a unitless `line-height` for the element. In this case, each `<p>` element inherits a `line-height` value of 1.5 from the `<body>` element. There are three ways to mitigate this:

- Decrease the value of `line-height` for the `::first-letter` pseudo-element. A value of `.5` seems to work well most of the time, depending on the font.
- Set a `line-height` with units on the `::first-letter` pseudo-element.
- Set a `line-height` with units on either the `<body>` or the `::first-letter` parent.

The first option preserves the vertical rhythm that comes with using a unitless `line-height`. The second option limits the side effects of using a fixed `line-height` just to those pseudo-elements. Option three, however, introduces a high likelihood that you'll create a side effect that requires more CSS to undo.

Why Unitless?

The Mozilla Developer Network entry for `line-height` explains why unitless values are the way to go.

In this case, let's decrease the `line-height` value for `p::first-letter` to `.5` (and rewrite our file properties to use the `font` shorthand):

```
p::first-letter {
    font: normal 10rem / 1 'Alfa Slab One', serif;
    color: hsl(291, 64%, 30%);
    display: inline-block;
    padding-right: .25rem;
}
```

This change produces the result shown in the image below.

Notice here that we adjusted the bottom margin of each `<p>` element to compensate for the reduced `line-height` of `p::first-letter`.

Creating a drop capital requires a few more lines of CSS. Unlike an initial capital, the text adjacent to the drop capital letter wraps around it. This means that we need to add `float:left;` to our rule set:

```
p::first-letter {
    float: left;    /* Makes the text wrap around the drop
cap */
    font: normal 10rem / 1 'Alfa Slab One', serif;
    color: hsl(200, 18%, 46%);
    margin-bottom: .4rem;
    padding: 1rem 1.4rem 0 0;
    text-shadow: 2px 2px 0px hsl(200, 100%, 10%);
}
```

Floating an element, or in this case a pseudo-element, causes the remaining text to flow around it, as illustrated below.

Be aware that `::first-letter` can be difficult to style with pixel-perfect accuracy across browsers, unless you use `px` or `rem` units for size, margin, and line height.

Sometimes the first letter of a text element is actually punctuation—such as in a news story that begins with a quote:

```
<p>&#8220;Lorem ipsum dolor sit amet, consectetur adipiscing
elit.&#8221; Fusce odio leo, sollicitudin vel mattis eget, …
</p>
```

In this case, the styles defined for `::first-letter` affect both the opening punctuation mark and the first letter, as shown below. All browsers handle this in the same way.

However, this isn’t necessarily how it works when the punctuation mark is generated by an element. Consider the following markup:

```
<p><q>Lorem ipsum dolor sit amet, consectetur adipiscing
elit.</q> Fusce odio leo, sollicitudin vel mattis eget,
iaculis sit …</p>
```

Current browsers typically render the `<q>` element with language-appropriate quotation marks before and after the enclosed text. Safari, Chrome, and Edge ignore the opening quotation mark. However, Firefox versions 90 and under apply `::first-letter` styles to the opening quotation mark, not the first letter.

In Chrome-based browsers and Safari, neither the opening quotation mark for the `<q>` element nor the first letter of the paragraph are restyled. The image below shows how this looks in Chrome.

According to the CSS Pseudo-elements Module Level 4 specification, punctuation that immediately precedes or succeeds the first letter or character should be included. However, the specification is unclear about whether this also applies to generated punctuation.

In Firefox 90 and earlier, some punctuation characters cause Firefox to ignore a `::first-letter` rule altogether. These include, but are not limited to, the dollar sign ($), caret (^), back tick (`) and tilde (~) characters. Firefox, to date, also doesn't apply `::first-letter` styles to emoji.

This is true whether the first character is set using `::before` and the `content` property, or included in the document source. There's no fix for this. Avoid using these characters as the first character if you're also using `::first-letter`.

The `::first-letter` pseudo-element doesn't apply to elements such as `<a>`, `<b>`, or `<code>`. Nor does it apply to parent elements with a `display` property value of `inline`.

Creating Typographic Effects with `::first-line`

The `::first-line` pseudo-element works similarly to `::first-letter`, but affects the entire first line of an element. For example, the first line of every paragraph could have larger text and a different color from the rest of the paragraph:

```
p::first-line {
    font: bold 1.5em serif;
    font-style: italic;
    color: #673ab7;
}
```

You can see the result in the image below.

Notice that the first line of each paragraph is affected, rather than the first sentence. The font size and element width determine how many characters fit on this first line.

It’s possible to force the end of a first line by using a `<br>` or `<hr>` element, as shown below.

Unfortunately, this is far from perfect. If your element is only wide enough to accommodate 72 characters, adding a `<br>` element after the 80th character won’t affect the `::first-line` pseudo-element. You’ll end up with an oddly placed line break.

Similarly, using a non-breaking space (`&nbsp;`) to prevent a line break between words won't affect `::first-line`. Instead, the word that sits before the `&nbsp;` will be forced onto the same line as the text that comes after it.

Generated content that's added using `::before` will become part of the first line, as shown in the image below.

If the generated text is long enough, it will fill the entire first line. However, if we add a `display: block` declaration—such as `p::before {content: '!!!'; display: block;}`—that content will become the entire first line.

Unfortunately, versions of Firefox 90 and below handle this differently. Firefox correctly inserts the value of the `content` property, but adding `display: block` causes the `::first-line` rule to fail completely.

Not all properties are compatible with `::first-line`. Only the following are supported:

- `background` and the `background-` prefixed properties
- `color`
- `font` and the `font-` prefixed group of properties
- `letter-spacing`
- `line-height`
- `opacity`
- `text-decoration`, including expanded properties such as `text-decoration-line`
- `text-transform`
- `text-shadow`
- `word-spacing`
- `vertical-align`

User Interface Fun with `::selection`

The `::selection` pseudo-element is one of the so-called “highlight pseudo-elements” defined by the CSS Pseudo-Elements Module Level 4 specification. Formerly part of the Selectors Level 3 specification, it’s the only highlight pseudo-element that’s currently supported by browsers. Three other properties—`::target-text`, `::spelling-error` and `::grammar-error`—are still in flux.

With `::selection`, we can apply CSS styles to content that users have highlighted with their mouse. By default, the background and text color of highlighted content is determined by the user’s system settings. Developers, however, can change what that highlight looks like—as shown below, where the selection color has been set to green.

Not every CSS property can be used with `::selection`. Only a few properties are permitted by the specification:

- `color`
- `background-color`
- `text-decoration`, and related properties such as `text-decoration-style`
- `text-shadow`
- `stroke-color`
- `fill-color`
- `stroke-width`

To date, only `text-shadow`, `color` and `background-color` have been implemented in browsers. Let's look at an example:

```
::selection {
    background: #9f0; /* lime green */
    color: #600;
}
```

This CSS adds a lime-green background to any element the user highlights, and changes the text color to a deep red. The example works in every browser that supports `::selection`, and you can see the effect in the image below.

Color Combinations

When selecting foreground and background colors to use with `::selection`, keep accessibility in mind. Some color combinations fail to generate enough contrast to be read by low-vision users. Other color combinations may be illegible for color-blind users. Be sure to use a contrast checker and color-blindness simulator before selecting your final colors.

Custom List and Summary Icons with `::marker`

`::marker` is a pseudo-element that represents a bullet or number indicator of elements with a `display` value of `list-item`. In most current browser versions, the default user-agent stylesheet applies `display: list-item` to `<li>` and `<summary>` elements.

Any element with a `list-item` display value will generate a marker box that can be selected and styled using `::marker`. Using a `display` value other than `list-item`—such as `display: inline` or `display: grid`—removes the marker box and the ability to use `::marker`.

Browser support for `::marker`

Safari 14.2 and below, and Chrome/Edge versions 88 and below, don't support `::marker` when used with the `<summary>` element. Instead, use the `::-webkit-details-marker` pseudo-element. Keep in mind, however, that `::-webkit-details-marker` is limited to `color` and `font` properties.

With `::marker`, we can do things like define custom bullet content for unordered lists, or change the size and color of the numbers in an ordered list:

```
ol ::marker {
  color: blue;
  font-size: 4rem;
}
```

You can see the effect of this rule in the image below.

Only a small subset of CSS properties can be used with `::marker`, as outlined in the CSS Lists and Counters Module Level 3 specification:

- `color`
- `content`
- `direction`
- `font`, along with its longhand properties such as `font-size` and `font-weight`
- `white-space`
- animation and transition properties, such as `animation-transition` and `transition-delay`
- `text-combine-upright`
- `unicode-bidi`

Future versions of the specification may expand this list. To date, we're limited to the above properties.

Further Safari Limitations

Safari versions 14.2 and below only partially support `::marker`. They render color and font styles, but not the value of the `content` property.

Because of these limitations, `li::before` can be a more flexible option for adding custom bullets or numbers. Using `::before` gives you more control over things like horizontal spacing between bullets and content, and vertical alignment. It's also well-supported in older browsers.

In browsers that support both, you may instead choose to use both `::marker` and `::before`:

```
li::marker {
  content: '✕';
}
li::before {
  content: '\00a0'; /* Unicode for a non-breaking space */
  display: inline-block;
  padding: 0 10px;
}
```

In the preceding example, we've used `::marker` to set the content and color of list item bullets, and `::before` to manage the spacing between the markers and each list item's contents. You can see the results below.

For the most part, list style properties interact with the `::marker` pseudo-element. Adding a `list-style: upper-roman` declaration, for example, sets the numeric markers for an unordered list. You can then use `::marker` to change the size and color:

```
ul {
    list-style: upper-roman;
}
ul ::marker {
    font-size: 4rem;
    color: green;
}
```

But there’s an exception: if you set the `content` property of a list item using `::marker`, most browsers will render that value instead of the value of `list-style` or `list-style-type`:

```
ul ::marker {
    content: '◈';        /* Rendered in browsers that support
::marker */
    font-size: 2.3rem;
    font-weight: bold;
    color: #090;
}
ul {
    list-style: '→';      /* Visible when ::marker is
unsupported */
}
```

The image below shows how, in Firefox 90, `::marker` takes precedence over `list-style` when both are defined and supported.

Styling Input `::placeholder` Values

Text-based form inputs have a `placeholder` attribute that lets us add a hint about what kind of data the field expects:

```
<form>
    <p>
        <label for="subscriber_email">Your email
address</label>
        <input type="email" name="subscriber_email"
id="subscriber_email" placeholder="yourname@example.com">
```

```
    </p>
</form>
```

Most browsers display the value of the `placeholder` attribute within the form control as black text with a reduced opacity, as pictured below.

We can change the appearance of this text using the `::placeholder` pseudo-element selector. Let’s change the color and size of our placeholder text:

```
::placeholder {
    color: rgba(0, 0, 100, 1);
    font-weight: bold;
}
```

Now this is what we see.

Placeholder Text Can Be Harmful

Placeholder text can be confusing for many users, particularly those with cognitive impairments. Consider using descriptive text positioned near the form control instead. Placeholder text is also not a replacement for the `<label>` element. Use labels with your form controls, even if you use the `placeholder` attribute.

`::placeholder` supports the same subset of CSS properties as `::first-line`. When changing the appearance of `::placeholder` text, choose colors and text sizes that create sufficient contrast. Firefox includes tools to check for basic accessibility blunders such as poor contrast between text and background colors.

Later in this chapter, we'll discuss the `:placeholder-shown` pseudo-class, which applies to the form control itself.

Pseudo-classes

As mentioned earlier in this chapter, pseudo-classes help us define styles for documents based on information that can't be gleaned from the document tree or that can't be targeted using simple selectors. Among them are logical and linguistic pseudo-classes such as `:not()` and `:lang()`, as well as user-triggered pseudo-classes such as `:hover` and `:focus`, and form state pseudo-classes such as `:checked` and `:invalid`.

Styling the `:root` of a Document

One pseudo-class you often see in CSS snippets and demos is the `:root` pseudo-class. The `:root` pseudo-class matches the root element of the document. When working with HTML, this matches the `<html>` element. For SVG documents, it's the `<svg>` element.

You might choose `:root` over `html` if you need to define a set of custom properties (variables) for a stylesheet that will be shared across HTML and SVG documents. The following example uses `:root` and custom properties to define a color palette:

```
:root {
  --color-primary: blue;
  --color-secondary: magenta;
  --color-tertiary: yellowgreen;
}
```

Linking this stylesheet from an SVG or HTML document makes these properties available for use with either in a way that using `html` as a selector doesn’t.

Highlighting Page Fragments with `:target`

A fragment identifier is the part of a URL starting with a `#` character—such as `#top` or `#footnote1`. You may have used them to create in-page navigation links—sometimes called “jump links”. With the `:target` pseudo-class, we can highlight the portion of the document that corresponds to that fragment.

Say, for example, you have a series of comments in a discussion board thread:

```
<section id="comments">
    <h2>Comments</h2>
    <article class="comment" id="comment-1146937891">...
</article>
    <article class="comment" id="comment-1146937892">...
</article>
    <article class="comment" id="comment-1146937893">...
</article>
</section>
```

With some CSS and other fancy bits, it might look a little like what’s pictured below.

Each comment in the code above has a fragment identifier, which means we can link directly to it with an anchor link such as `<a href="#comment-1146937891">` or `<a href="http://example.com/post/#comment-1146937891">`. Then all we need to do is specify a style for this comment using the `:target` pseudo-class:

```
.comment:target {
    background: #ffeb3b;
    border-color: #ffc107
}
```

When someone clicks a link to an `<article>` element with a class of `comment`, the browser will take them to that comment and give it a yellow background, as shown below.

You can use any combination of CSS with `:target`, but be cautious about using properties that can show or hide content. Adjusting the `z-index` property, for example, can hide content on the page, but still expose it to assistive technology. That may not be what you want.

Styling Elements That Have No Children Using `:empty`

Sometimes WYSIWYG (what you see is what you get) editors add empty `<p>` elements to your content. These empty elements may affect your document layout if your stylesheet also uses `p` as a type selector. It's possible to visually exclude these elements, however, with the `:empty` pseudo-class:

```
p:empty {
  display: none;
}
```

Earlier versions of the selectors specifications defined `:empty` elements as elements devoid of any element or text nodes—including space or newline characters. This means that for most current implementations, `p:empty` matches `<p></p>`, but not `<p> </p>`.

Perhaps unexpectedly, `:empty` will *always* match `<img>` and `<input>` elements when used with the universal selector (again, `:empty` is the same as `*:empty`). For `<input>` elements, this is true even if the field contains text.

In the meantime, you can use the `:placeholder-shown` pseudo-class to select blank form control fields. We'll discuss this selector later in the chapter.

Concise and Resilient Selectors with `:is()`

The `:is()` pseudo-class is one of three logical pseudo-classes available in CSS—the other two being `:not()` and `:where()` (which we'll discuss in the next sections).

Can I `:has()` a Parent Selector?

CSS Selectors Level 4 also defines a fourth logical pseudo-class, `:has()`. Unfortunately, `:has()` lacks browser support, so we won't cover it in this chapter. It’s sometimes referred to as the elusive “parent selector”, a long wished for but difficult to implement concept. In early 2021, Igalia announced an intent to prototype support for `:has()`. Full browser support may take some time. In the meantime, parent selection remains the preserve of JavaScript.

You can use `:is()` to create more concise and resilient selectors. It’s a functional pseudo-class that accepts a selector list as its argument. Here’s an example:

```
article :is( h1, h2, h3, h4 ) {
    font-family: 'EB Garamond', serif;
    font-style: italic 45deg;
}
```

In this case, our selector matches `<h1>`,`<h2>`,`<h3>`, or `<h4>` elements that are the descendants of an `<article>`. It’s the equivalent of writing the following:

```
article h1,
article h2,
article h3,
article h4 {
    font-family: 'EB Garamond', serif;
    font-style: italic 45deg;
}
```

That’s a much longer selector list! Using `:is()` significantly reduces the length of selectors.

Before `:is()` Was

WebKit implemented an earlier version of the `:is()` pseudo-class as `:-webkit-any()`. The main difference was that `:-webkit-any()` didn’t support a selector list argument. Earlier versions of the Selectors Level 4 specification also defined a `:matches()` pseudo-class. It’s been replaced by `:is()`.

The `:is()` function accepts what’s known as a **forgiving selector list**. Consider the following CSS:

```
:is( :blank, :placeholder-shown ) {
    font: inherit;
    padding: 1rem;
    color: #003a;
}
```

The selector above matches input elements that are blank *or* that have visible placeholder text. Here's the catch: most browsers don't yet support the `:blank` pseudo-class. Despite this, our declarations will still be applied to elements that match `:placeholder-shown`. A forgiving selector lists tells browsers to ignore selectors that the browser doesn't understand.

Forgiving selector lists are a newer CSS concept. Earlier CSS specifications defined how a browser should treat a selector list that it can't fully parse, whether the error is due to a lack of browser support or a typo. As explained in the CSS 2.1 Specification:

> CSS 2.1 gives a special meaning to the comma (`,`) in selectors. However, since it is not known if the comma may acquire other meanings in future updates of CSS, the whole statement should be ignored if there is an error anywhere in the selector, even though the rest of the selector may look reasonable in CSS 2.1.

In other words, if any item in a standard selector list isn't supported, the browser ignores the entire rule. Using `:is()`, on the other hand, lets the browser ignore selectors that it doesn't understand.

Browser Support

Yes, `:is()` lets us write resilient selectors, but that resiliency still requires browser support. If the browser doesn't support `:is()`, the original parsing rules still apply. The entire rule will fail.

Negating Selectors with `:not()`

The `:not()` pseudo-class is the opposite of `:is()`. It returns all elements *except* for those that match the selector argument. For example, `p:not(.message)` matches every `<p>` element that *doesn't* have a class of `message`.

Here’s an example of a form that uses textual input types and radio buttons:

```
<form method="post" action="#">
      <h1>Join the Cool Kids Club</h1>
      <p>
          <label for="name">Name:</label>
          <input type="text" id="name" name="name" required>
      </p>

      <p>
          <label for="email">Email:</label>
          <input type="email" id="email" name="email"
required>
      </p>
      <fieldset>
      <legend>Receive a digest?</legend>
      <p>
          <input type="radio" id="daily" name="digest">
          <label for="daily" class="label-
radio">Daily</label>
          <input type="radio" id="weekly" name="digest">
          <label for="weekly" class="label-
radio">Weekly</label>
      </p>
      </fieldset>
      <button type="submit">Buy Tickets!</button>
</form>
```

In the HTML, labels associated with a radio type have a `.label-radio` class. We can use `:not()` to target those elements without a `.label-radio` class:

```
label:not( .label-radio ) {
    font-weight: bold;
    display: block;
}
```

The end result is shown below.

Here’s a slightly trickier example. Let’s create styles for textual inputs. These include input types such as number, email, and text along with password and URL. Let’s do this by excluding radio button, checkbox, and range inputs:

```
input:not( [type=radio], [type=checkbox], [type=range] ) {
    ...
}
```

As with `:is()`, the `:not()` pseudo-class accepts either a single selector or a selector list as an argument. It will match any and all of the supported selectors in the list.

Chrome and Edge versions 87 and below, and Firefox versions 83 and below, implement an earlier definition of `:not()` that doesn't accept selector lists. Instead, those browsers accept a single selector argument. For those browsers, we'll need a different approach.

Your instinct might be to rewrite the preceding example like so:

```
input:not( [type=radio] ),
input:not( [type=checkbox] ),
input:not( [type=range] ) {
    ...
}
```

Unfortunately, this won't work. Each selector overrides the previous one. It's the equivalent of typing:

```
input:not( [type=radio] ){ ... }
input:not( [type=checkbox] ) { ... }
input:not( [type=range] ) { ... }
```

Instead, you'll need to use the following selector:

```
input:not( [type=radio] ):not( [type=checkbox] ):not(
[type=range] ) {
    ...
}
```

Each instance of `:not()` in this selector further filters the list, achieving our desired result.

Pseudo-elements aren't valid arguments for `:is()` and `:not()`. A selector such as `:is(::first-letter)` or `:is(::marker, ::-webkit-details-marker)` won't match any elements, and browsers will ignore the rules associated with that selector.

Adjusting Selector Specificity with `:where()`

The CSS Selectors Level 4 specification calls `:where()` the “specificity-adjustment pseudo-class”. It’s also a functional pseudo-class that accepts a selector or a selector list as its argument. Using `:where()` limits the impact of a selector’s specificity without changing it.

Consider this CSS snippet:

```
a:not( :hover ) { /* Specificity of 0,1,1 */
    text-decoration: underline 2px;
}
nav a { /* Specificity of 0,0,2. This rule does not take
effect */
    text-decoration: hsl( 340, 82%, 52% ) wavy underline
1.5px;
}
```

In this example, our first rule has a more specific selector than our second. As a result, the second rule is never applied, and the links don’t get a wavy, pink underline.

One way to resolve this would be to change `nav a` to `nav a:not(:hover)`. Doing so would increase the specificity of that rule, which may not be what you want. Let's try `:where()` instead:

```
a:where( :not( :hover ) ) { /* Retains specificity of 0,1,1
but with an adjustment */
    text-decoration: underline 2px;
}
nav a { /* Rule takes effect. Still has a specificity of
0,0,2 */
    text-decoration: red wavy underline 1.5px;
}
```

Adding `:where()` says to the browser: "Apply this style to `<a>` elements only where they don't have a hover state." Now our navigation links have squiggly underlines.

Again, using `:where()` doesn't modify the specificity *value* of a selector. In fact, its specificity value is zero. Instead, think of it as a way to clarify your intent.

Selecting Elements by Their Index

CSS also provides selectors for matching elements based on their position in the document subtree. These are known as **child–indexed pseudo-classes**, because they rely on the position or order of the element rather than its type, attributes, or ID. There are five:

- `:first-child`
- `:last-child`
- `:only-child`
- `:nth-child()`
- `:nth-last-child()`

`:first-child` and `:last-child`

As you've probably guessed from the names, the `:first-child` and `:last-child` pseudo-classes make it possible to select elements that are the first child or last child of a node (element). As with other pseudo-classes, `:first-child` and `:last-child` have the fewest side effects when they're part of a compound selector.

Let's take a look at the HTML and CSS below:

```
<!DOCTYPE html>
    <html lang="en-US">
    <head>
    <meta charset="utf-8">
    <title>:first-child and :last-child</title>
    <style type="text/css">
    body {
        font: 16px / 1.5 sans-serif;
    }
    :first-child {
        color: #e91e63; /* hot pink */
    }
    :last-child {
         color: #4caf50; /* green */
    }
    </style>
</head>
<body>
    <h2>List of fruits</h2>
    <ul>
        <li>Apples</li>
        <li>Bananas</li>
        <li>Blueberries</li>
        <li>Oranges</li>
        <li>Strawberries</li>
    </ul>
</body>
</html>
```

This code produces the result shown below. Using `:first-child` by itself matches more elements than we want.

Because `:first-child` is unqualified, both the `<h2>` element and first `<li>` element are hot pink. After all, `<h2>` is the first child of `<body>`, and the Apples `<li>` is the first child of the `<ul>` element. But why are the remaining `<li>` elements green? Well, that's because `:last-child` is also unqualified, and `<ul>` is the last child of body. It's effectively the same as typing `*:first-child` and `*:last-child`.

If we qualify `:first-child` and `:last-child` by adding a simple selector, it all makes more sense. Let's limit our selection to list items. Change `:first-child` to `li:first-child` and `:last-child` to `li:last-child`. The result is shown below.

`:only-child`

The `:only-child` pseudo-class matches elements if they're the only child of another element. In the following example, we have two parent `<div>` elements and their child elements. The first `<div>` contains one item, while the second contains three:

```
<div>
   <span class="fruit">Apple</span>
</div>

<div>
   <span class="fruit">Banana</span>
   <span class="vegetable">Spinach</span>
   <span class="vegetable">Okra</span>
</div>
```

Using `.fruit:only-child {color: #9c27b0; /* violet */}` will match `<span class="fruit">Apple</span>`, since it's the only child of the first `<div>`. None of the items in the second `<div>` match, however, because there are three siblings. You can see what this looks like below.

`:nth-child()` and `:nth-last-child()`

The ability to select the first and last children of a document is fine. But what if we want to select odd or even elements instead? Perhaps we'd like to pick the sixth element in a document subtree, or apply styles to every third element. This is where the `:nth-child()` and the `:nth-last-child()` pseudo-classes come into play.

Like `:not()`, `:nth-child()` and :`nth-last-child()` are also functional pseudo-classes. They accept a single argument, which should be one of the following:

- the `odd`keyword
- the `even` keyword
- an integer (such as 2 or 8)
- an argument in the form `An+B`(where `A` is a step interval, `B` is the offset, and `n` is a variable representing a positive integer). This `An+B` syntax is described in CSS Syntax Module Level 3.

That last item has a degree of complexity. We'll come back to it shortly.

The difference between `:nth-child()` and `:nth-last-child()` is the starting point. `:nth-child()` counts forward and `:nth-last-child()` counts backward. CSS indexes use counting numbers and start with one rather than zero.

Both `:nth-child()` and `:nth-last-child()` are useful for alternating patterns. Creating zebra-striped table row colors is a perfect use case. The CSS that follows gives even-numbered table rows a light, bluish-gray background:

```
tr:nth-child(even) {
    background: rgba(96, 125, 139, 0.1);
}
```

Here's the result seen in the browser.

Changing `:nth-child()` to `:nth-last-child()` inverts this banding, since the counting begins from the bottom.

How about trying some complex examples using more complex arguments? We’ll start with the document pictured below, which contains 20 `<div>` items.

With `:nth-child()` and `:nth-last-child()`, we can select a single child at a particular position. We can select all of the children after a particular position, or we can select elements by multiples, with an offset. Let’s change the background color of the sixth item:

```
.item:nth-child(6) {
    background: #e91e63; /* red */
}
```

This gives us the result below.

But what if we want to select every third element? Here's where the `An+B` syntax comes in:

```
.item:nth-child(3n) {
    background: #e91e63; /* red */
}
```

Again, `A` is a step interval. It's kind of a multiplier for `n`, which starts at 0. So if `A` equals 3, then `3n` would match every third element (the 3rd, 6th, 9th elements, and so on). That's exactly what happens, as you can see below.

Counting with `n`

We noted earlier that selectors count *elements* from 1. However, `n` is a variable that represents any number *from zero*. The `3n` in our `.item:nth-child(3n)` selector above produces `3 x 0`, `3 x 1`, `3 x 2` and so on. Of course, `3 x 0` equals zero, so we don't see any visual styling based on this, as there's no element zero. It's important to note that `n` starts at zero, because, as we'll see below when we introduce `+`, `n+8` will produce results starting from 8 (because `0 + 8` equals 8).

Matters become even more interesting when we use `:nth-child()` and `:nth-last-child()` to select all elements after a certain point. Let's try selecting all but the first seven elements:

```
.item:nth-child(n+8) {
    background: #e91e63;
}
```

Here, there's no step value. As a result, `n+8` matches every element `n` beginning with the eighth element, as shown below.

Negative Offsets

Negative offset and range values are also valid. Using `:nth-child(-n+8)` would invert our selection, and match the first eight elements.

We can also use the offset and step values to select every third element, starting with the fifth:

```
.item:nth-child(3n+5) {
    background: #e91e63;
}
```

You can see the results of this selector below.

Selecting Elements of a Particular Type by Their Index

The pseudo-classes discussed in the previous section match elements if they occupy the given position in a document subtree. For instance, `p:nth-last-child(2)` selects every `<p>` element that's the next-to-last element of its parent.

In this section, we'll discuss **typed child-indexed pseudo-classes**. These pseudo-classes also match elements based on the value of their indexes, but matches are limited to elements of a particular type or tag name—such as selecting the fifth `<p>` element, or all even-indexed `<h2>` elements.

There are five such pseudo-classes with names that mirror those of their untyped counterparts:

- `:first-of-type`
- `:last-of-type`
- `:only-of-type`
- `:nth-of-type()`
- `:nth-last-of-type()`

The difference between these and child-indexed pseudo-classes is a subtle one. Where `p:nth-child(5)` matches the fifth item only if it's a `<p>` element, `p:nth-of-type(5)` matches all `<p>` elements, then finds the fifth `<p>` element among those.

Let's start with a slightly different document. It still has 20 items, but some of them are `<p>` elements and some of them are `<div>` elements. The `<p>` elements have rounded corners, as can be seen below.

Using `:first-of-type`, `:last-of-type`, and `:only-type`

With `:first-of-type`, we can select the first element that matches a selector. How about we give our first `<p>` element a lime-green background:

```
p:first-of-type {
    background: #cddc39; /* lime green */
}
```

This will match every `<p>` element that’s the first `<p>` element of its parent.

The `:last-of-type` pseudo-class works similarly, matching the last such element of its parent.

However, `:only-of-type` will match an element if it's the only child element of that type of its parent. In the image below, we're using p:only-of-type to match the only child that's a paragraph element.

Let's look at another example of using `:first-of-type`, but this time with a pseudo-element. Remember the `::first-letter` pseudo-element from earlier in this chapter? Well, as you saw, it created an initial capital for every element it was applied to. We'll now go one step further and limit this initial capital to the first paragraph instead:

```
p:first-of-type::first-letter {
    font: bold italic 3em / .5 serif;
    color: #3f51b5;
}
```

Now our paragraph will have an initial capital, even if it's preceded by a headline.

Using `:nth-of-type()` and `:nth-last-of-type()`

The `:nth-of-type()` and `:nth-last-of-type()` are also functional pseudo-classes. They accept the same arguments as `:nth-child()` and `:nth-last-child()`. But like `:first-of-type` and `:last-of-type`, the indexes resolve to elements of the same type. For example, to select every odd-numbered `<p>` element, we can use the `odd` keyword with `:nth-of-type()`:

```
p:nth-of-type(odd) {
    background: #cddc39;
    color: #121212;
}
```

As you can see from the image below, this only matches odd-numbered *paragraph* elements, even though there are other element types in between them.

Similarly, using `:nth-last-of-type(even)` selects even-numbered `<p>` elements, but the count begins from the last `<p>` element in the document—in this case, item 18.

Using `*-of-type` Pseudo-classes with Non-element Selectors

The “of-type” selectors are designed to be used with element selectors—such as `p:first-of-type`. You might be tempted to use “of-type” selectors to target the first instance of some other kind of hook, such as a class—as in `.item:first-of-type`. But this can lead to unexpected results. In the markup that follows, we have three list items and a paragraph element, all of which have a class attribute value of `item`:

```
<ul>
  <li class="item">Lorem ipsum dolor sit amet, consectetur
adipiscing elit.</li>
  <li class="item">Pellentesque sodales at risus vel
fermentum.</li>
  <li class="item">Fusce et eros iaculis, accumsan ligula ac
felis. </li>
</ul>

<p class="item">Duis nec ex at arcu rhoncus rhoncus sit amet
at est. Donec condimentum accumsan justo. Ut convallis
faucibus sollicitudin.</p>
```

Let’s say we want to target just the first element with a class of `item`. If we add a rule `.item:first-of-type {background: magenta;}`, you might expect just the first list item to have a magenta background. That isn’t what happens, however, as you can see in the image below.

Instead, our paragraph element also has a magenta background. Because it's the first paragraph type element in the document, it also matches the `.item:first-of-type` selector.

The Selectors Level 4 specification adds a new argument syntax for `:nth-of-type()`/`:nth-last-of-type()` to bring its behavior more into line with developer expectations: the `of [S]` syntax, where `[S]` is a non-element selector.

To use our previous markup example, we could select the first instance of an element with the `item` class using the following CSS:

```
:nth-of-type(1 of .item) {
   background: magenta;
}
```

This matches the first element with an `item` class attribute value. To date, however, Safari is the only browser that supports this syntax.

Styling Form Fields Based on Input

Let's take a look at some pseudo-classes that are specific to form fields and form field input. These pseudo-classes can be used to style fields based on the validity of user input, whether the field is required or currently enabled.

All of the pseudo-classes that follow are specific to forms. As a result, there's less of a need to limit the scope with a selector. Using `:enabled` won't introduce side effects for `<span>` elements. Limiting the scope is helpful, however, when you want to style various types of form controls differently.

`:enabled` and `:disabled`

As their name suggests, these pseudo-classes match elements that have (or lack) the HTML5 `disabled` attribute. This can be elements such as `<input>`, `<select>`, `<button>` or `<fieldset>`:

```
<button type="submit" disabled>Save draft</button>
```

Form elements are enabled by default. That is, they only become disabled if the `disabled` attribute is set. Using `input:enabled` will match every input element that doesn't have a `disabled` attribute. Conversely, `button:disabled` will match all button elements with a `disabled` attribute:

```
button:disabled {
    opacity: .5;
}
```

The image below shows the `:enabled` and `:disabled` states for our `<button>` element.

`:required` and `:optional`

Required and optional states are determined by the presence or absence of the `required` attribute on the field. Remember that, in HTML5, the presence or absence of the attribute determines its value. In other words, `required="false"` has the same effect as `required="true"`, `required="required"` and `required`. For example:

```
<p>
    <label for="email">Email:</label>
    <input type="email" id="email" name="email"
placeholder="example: jane.doe@example.com" required>
</p>
```

Most browsers only indicate whether a field is required once the form is submitted. With the `:required` pseudo-class, we can indicate to the user that the field is required before submission. For example, the following CSS will add a yellow border to our email field:

```
input:required {
    border: 1px solid #ffc107;
}
```

The `:optional` class works similarly, by matching elements that don't have a `required` attribute. For example, take the following CSS:

```
select:optional {
    border: 1px solid #ccc;
}
```

This produces the following result in Firefox 86.

`:checked`

Unlike the other form-related pseudo-classes we've covered, `:checked` only applies to radio and checkbox form controls. As the name indicates, this pseudo-class lets us define separate styles for selected inputs.

In order to create custom radio button and checkbox inputs that work well across browsers, we'll need to be a little bit clever with our selectors. Let's combine an adjacent sibling combinator, a pseudo-element, and `:checked` to create custom radio button and checkbox controls. For example, to change the style of a label when its associated radio button is checked, we could use the following CSS:

```
[type=radio]:checked + label {
    font-weight: bold;
    font-size: 1.1rem;
}
```

This makes the label bold and increases its size when its associated control is checked. We can improve this, though, by using the `::before` pseudo-element with our `<label>` element to inject a custom control:

```
[type=radio] {
  /*
    appearance: none removes default browser styles for radio
buttons
    and other elements. Safari supports this property with a
-webkit-
    prefix.
  */
  -webkit-appearance: none;
          appearance: none;
}
[type=radio] + label::before {
    background: #fff;
    content: '';
    display: inline-block;
    border: 1px solid #444;
    border-radius: 1000px;
    height: 1.2rem;
    margin-right: 1em;
    vertical-align: middle;
    width: 1.2rem;
}
[type=radio]:checked + label::before {
    background: #4caf50;
}
```

This gives us the customized controls you see below.

In order for this technique to work, of course, our HTML needs to be structured appropriately:

- The `<label>` element must be immediately adjacent to its `<input>` control.
- The form control must have an `id` attribute in addition to the `name` attribute (for example, `<input type="radio" id="chocolate" name="flavor">`).
- The label must have a `for` attribute, and its value must match the ID of the form control (for example, `<label for="chocolate">Chocolate</label>`).

Associating the `<label>` using `for` with the input ensures that the form input will be selected when the user clicks or taps the label or its child pseudo-element (`::before`).

`:indeterminate`

The `:indeterminate` pseudo-class lets you set styles for elements that are in an indeterminate state. Only three types of elements can have an indeterminate state:

- `<progress>` elements, when it's unclear how much work remains (such as when waiting for a server response)
- grouped `input[type=radio]` form controls, before the user selects an option
- `input[type=checkbox]` controls, when the `indeterminate` attribute is set to `true` (which can only be done via DOM scripting)

Indeterminate Checkboxes

CSS-Tricks.com provides a useful overview of indeterminate checkbox controls, including when and why you might use them.

Let's look at an example using the `<progress>` element:

```
<div>
    <label for="upload">Uploading progress</label>
    <progress max="100" id="upload" aria-
describedby="progress-text"></progress>
    <span id="progress-text">0 of <i>unknown</i> bytes.
</span>
</div>
```

Notice here that we haven't included a `value` attribute. For most WebKit- and Blink-based browsers, the presence or absence of the `value` attribute determines whether a `<progress>` element has an indeterminate state. Firefox, on the other hand, sets an indeterminate state for `<progress>` elements when the `value` attribute is empty.

Unfortunately, `<progress>` elements still require vendor-prefixed pseudo-elements. Here's our CSS:

```
progress {
    background: #ccc;
    box-shadow: 0 0 8px 0px #000a;
    border-radius: 1000rem;
    display: block;
    overflow: hidden;
    width: 100%;
}
/* Firefox progress bars */
progress:indeterminate::-moz-progress-bar {
    background: repeating-linear-gradient(-45deg, #999,  #999
1rem, #eee 1rem, #eee 2rem);
}
/* WebKit and Blink progress bars */
progress:indeterminate::-webkit-progress-bar {
    background: repeating-linear-gradient(-45deg, #999,  #999
1rem, #eee 1rem, #eee 2rem);
}
/* Perhaps someday we'll be able to do this */
progress:indeterminate {
    background: repeating-linear-gradient(-45deg, #999,  #999
1rem, #eee 1rem, #eee 2rem);
}
```

This CSS gives us the progress bar shown below.

When the `value` of the `progress` element changes, it will no longer have an `:indeterminate` state.

`:in-range` and `:out-of-range`

The `:in-range` and `:out-of-range` pseudo-classes can be used with range, number, and date input form controls. Using `:in-range` and `:out-of-range` requires setting `min` and/or `max` attribute values for the control. Here's an example using the number input type:

```
<p>
    <label for="picknum">Enter a number from 1-100</label>
    <input type="number" min="1" max="100" id="picknum"
```

```
name="picknum" step="1">
</p>
```

Let’s add a little bit of CSS to change styles if the values are within or outside of our range of one to 100:

```
:out-of-range {
    background: #ffeb3b;
}
:in-range {
    background: #fff;
}
```

If the user enters -3 or 101, the background color of `#picknum` will change to yellow, as defined in our `:out-of-range` rule.

Otherwise, it will remain white, as defined in our `:in-range` rule.

`:valid` and `:invalid`

With the `:valid` and `:invalid` pseudo-classes, we can set styles based on whether or not the form input meets our requirements. This will depend on the validation constraints imposed by the type or pattern attribute value. For example, an `<input>` with `type="email"` will be invalid if the user input is “foo 123”, as shown below.

A form control will have an invalid state under the following conditions:

- when a required field is empty
- when the user's input doesn't match the type or pattern constraints—such as `abc` entered in an `input[type=number]` field
- when the field's input falls outside of the range of its `min` and `max` attribute values

Optional fields with empty values are valid by default. Obviously, if user input satisfies the constraints of the field, it exists in a valid state.

Form controls can have multiple states at once. So you may find yourself managing specificity (discussed in the next section) and cascade conflicts. A way to mitigate this is by limiting which pseudo-classes you use in your projects. For example, don't bother defining an `:optional` rule set if you'll also define a `:valid` rule set.

It's also possible, however, to chain pseudo-classes. For example, we can mix the `:focus` and `:invalid` pseudo-classes to style an element only if it has focus: `input:focus:invalid`. By chaining pseudo-classes, we can style an element that has more than one state.

`:placeholder-shown`

Where `::placeholder` matches the placeholder text, the `:placeholder-shown` pseudo-class matches *elements* that currently have a visible placeholder. Placeholder text is typically visible when the form control is empty—that is, before the user has entered any information in the field. Any property that can be used with `<input>` elements can also be used with `:placeholder-shown`.

Remember that `:invalid` matches form controls that have a `required` attribute and no user data. But we can exclude fields for which no data has been entered by combining `:invalid` with `:not()` and `:placeholder-shown`:

```
input:not(:placeholder-shown):invalid {
    background-color: rgba(195, 4, 4, .25);
    border-color: rgba(195, 4, 4, 1);
    outline-color:  rgba(195,4,4, 1);
}
```

The image below shows the results. Both form fields are required, but only the field with invalid user-entered data is highlighted.

Our first field is visually marked invalid because the user has entered an invalid email address. However, the second field hasn't changed, because the user hasn't entered data.

As mentioned earlier in this section, placeholder text can introduce usability challenges. For that reason, it's best avoided. Removing the attribute, however, prevents us from using the `:placeholder-shown` pseudo-class.

But there's a simple fix. Set the value of the `placeholder` attribute to a whitespace character: `placeholder=" "`. This lets us avoid the usability issues associated with using placeholder text, but still takes advantage of the `:placeholder-shown` selector.

Conclusion

You've made it to the end of the chapter! I know that was a lot to take in. You should now have a good understanding of:

- what kinds of selectors are available for matching elements
- the difference between pseudo-elements and pseudo-classes
- how to use newer pseudo-classes introduced by the Selectors Level 3 and 4 specifications

In the next chapter, we'll address some golden rules for writing maintainable, scalable CSS.

Chapter 2: CSS Architecture and Organization

If you've ever worked on a CSS codebase of any size—or even a small codebase with multiple developers—you'll have realized how difficult it is to create CSS that's predictable, reusable, and maintainable without being bloated. With added developers often comes added complexity: longer selectors, colliding selectors, and larger CSS files.

In this chapter, we'll explore CSS architecture and organization. First up: file organization. We'll take a look at strategies for managing CSS across projects, or as part of your own CSS framework.

Then we'll look at *specificity*. It's a frequent pain point for CSS development, especially for teams. **Specificity** is the means by which browsers decide which declarations to apply. If you've ever wondered why all of the buttons on your site are green when you wanted some of them to be orange, this section is for you. We'll discuss how to calculate selector specificity, and choose selectors that maximize reusability while minimizing the number of characters you'll need.

Finally, we'll discuss some guidelines and methodologies for writing CSS. These rules make it easier to avoid selector-naming collisions and overly specific selectors—the kinds of issues that arise when working within teams.

File Organization

Part of a good CSS architecture is file organization. A monolithic file is fine for solo developers or small projects. For large projects—sites with multiple layouts and content types—it's smarter to use a modular approach and split your CSS across multiple files.

Splitting your CSS across files makes it easier to parcel tasks out to teams. One developer can work on form input components, while another can focus on a card pattern or media object component. Teams can split work sensibly and increase overall productivity.

So what might a good file structure that splits the CSS across files look like? Here's a structure that's similar to one I've used in projects:

- `typography.css`: font faces, weights, line heights, sizes, and styles for headings and body text
- `forms.css`: styles for form controls and labels
- `lists.css`: list-specific styles
- `tables.css`: table-specific styles
- `accordion.css`: styles for the accordion component
- `cards.css`: styles for the card component

CSS frameworks such as Bootstrap, Bulma, and UIkit use a similar approach. They all become quite granular, with separate files for progress bars, range inputs, close buttons, and tooltips. That granularity allows developers to include only the components they need for a project.

The details of how you split your CSS will depend on your own preferences and practices. If your workflow includes a preprocessor such as Sass or Less, these might be partials with a `.scss` or `.less` extension. You may also add a `_config.scss` or `_config.less` file that contains color and font variables.

Or perhaps you have a more component-centric workflow, as with the component library tool Fractal, or JavaScript frameworks like React and Vue.js. You might instead opt for a single `base.css` or `global.css` file that smooths out browser differences, and use a separate CSS file for each pattern or component.

Something to avoid: organizing your CSS by page or view. Page-centric approaches encourage repetitious code and design inconsistencies. You probably don't need both `.contact-page label` and `.home-page label` rule sets. Instead, try to find common patterns or components in your site's design and build your CSS around them.

Using multiple files during site development doesn't necessarily mean you'll use multiple files in production. In most cases, you'll want to optimize CSS delivery by concatenating files, and separating critical from non-critical CSS. We discuss optimization techniques in Chapter 3, "Debugging and Optimization".

File organization is just one aspect of CSS architecture. Despite its position in this chapter, it's actually the least important aspect. In my experience, most CSS architecture problems arise from selector choice and specificity. We'll discuss how to avoid these issues in the next section.

Specificity

Developers who come to CSS from more traditional programming languages sometimes note that *CSS has a global scope*. In other words, using `button` as a selector applies those declarations to every `<button>` element, whether that was intended or not.

A Quick CSS Vocabulary Review

A CSS **declaration** consists of a property paired with a value. **Properties** are features that you can modify, such as `box-sizing` or `display`. A **value** is the modification or settings for a property (such as `border-box` or `none`). Both `box-sizing: border-box` and `display: none` are examples of declarations. A set of declarations enclosed by curly braces (`{` and `}`) is a **declaration block**. Together, declaration blocks and their selectors are called **rules** or **rule sets**.

The "global" nature of CSS is really an issue of *specificity* and the cascade in *Cascading* Style Sheets. Although it may seem arbitrary at first, CSS has well-defined rules for determining what declarations to apply. Understanding specificity may be what separates CSS developers from CSS masters.

Calculating exact specificity values can seem tricky at first. As explained in the Selectors Level 4 specification, you need to:

- count the number of ID selectors in the selector (= A)
- count the number of class selectors, attribute selectors, and pseudo-classes in the selector (= B)
- count the number of type selectors and pseudo-elements in the selector (= C)
- ignore the universal selector

We then need to combine A, B, and C to get a final specificity value. Take the following rule:

```
input {
    font-size: 16px;
}
```

The selector for this rule, `input`, is a “type” or “element” selector. Since there’s only one type selector, the specificity value for this rule is 0,0,1. What if we add an attribute selector, as shown below?

```
input[type=text] {
    font-size: 16px;
}
```

Adding an attribute selector raises the value for this rule to 0,1,1. Let’s add a pseudo-class:

```
input[type=text]:placeholder-shown {
    font-size: 16px;
}
```

Now our selector’s specificity is 0,2,1. Adding an ID selector, as shown below, increases the specificity value to 1,2,1:

```
#contact input[type=text]:placeholder-shown {
    font-size: 16px;
}
```

Think of specificity as a score or rank that determines which style declarations get applied to an element. The universal selector (`*`) has a low degree of specificity. ID selectors have a high degree. Descendant selectors such as `p img`, and child selectors such as `.panel > h2`, are more specific than type selectors such as `p`, `img`, or `h1`. Class names fall somewhere in the middle.

Higher-specificity selectors are higher-priority selectors. Declarations associated with higher-specificity selectors are the declarations that the browser will ultimately apply.

Calculating Specificity

Keegan Street’s Specificity Calculator and Polypane’s CSS Specificity calculator are helpful for calculating selector specificity. Polypane’s calculator also supports selector lists.

However, when two selectors are equally specific, the cascade kicks in, and the last rule wins. Here's an example:

```
a:link {
    color: #369;
}
a.external {
    color: #f60;
}
```

Both `a:link` and `a.external` have a specificity value of 0,1,1: zero ID selectors, one class or pseudo-class, and one type (or element) selector. However, the `a.external` rule set follows the `a:link` rule set. As a result, `a.external` takes precedence. Most of our links will be cadet blue, but those with `class="external"` will be orange.

Complex and combinator selectors, of course, give us higher-specificity values. Consider the following CSS:

```
ul#story-list > .book-review {
    color: #0c0;
}
#story-list > .book-review {
    color: #f60;
}
```

Although these rule sets look similar, they aren't the same. The first selector, `ul#story-list > .bookreview`, contains a type selector (`ul`), an ID selector (`#story-list`), and a class selector (`.bookreview`). It has a specificity value of 1,1,1. The second selector, `#story-list > .book-review`, only contains an ID and a class selector. Its specificity value is 1,1,0. Even though our `#story-list > .book-review` rule follows `ul#story-list > .bookreview`, the higher specificity of the former means that those elements with a `.book-review` class will be green rather than orange.

Although most pseudo-classes increase the specificity of a selector, `:not()` and `:is()` work a bit differently. They don't change the specificity value of a selector. Instead, the specificity value of these selectors gets replaced by the value of the most specific selector in their arguments. In other words, the specificity value `:not( [type=text] )` is the same as `[type=text]`: 0,1,0.

Understanding the Impact of `!important`

The `!important` keyword upends some of these rules. When a declaration contains an `!important` keyword, that declaration takes precedence, regardless of specificity or cascade rules. Consider the following CSS:

```
body {
    background: pink !important;
}
html body {
    background: yellow;
}
```

Although `html body` has a higher level of specificity (0,0,2) than `body` (0,0,1), the `!important` keyword means that our document will have a pink background instead of a yellow one.

The Cascade in Cascading Style Sheets

Read more about how declarations are sorted in the “Cascade Sorting Order” section of the CSS Cascading and Inheritance Level 3 specification.

Overriding an `!important` declaration requires another `!important` declaration paired with a selector of equal or higher specificity. In other words, if we really wanted a yellow background, we’d need to add an `!important` keyword to the `html body` rule set.

Removing the `body` rule altogether is also an option. If you have complete control over your stylesheets and markup, doing so would save a few bytes. If you’re using a component library, or customizing a theme, you may prefer to override the rule instead. That way, your changes won’t be undone by a new release of the component or theme.

Choosing Low-specificity Selectors

Err on the side of using low-specificity selectors. They make it easier to reuse your CSS, and extend patterns in new ways.

Consider the following:

```
button[type=button] {
    background: #333;
    border: 3px solid #333;
    border-radius: 100px;
    color: white;
    line-height: 1;
    font-size: 2rem;
    font-family: inherit;
    padding: .5rem 1rem;
}
```

This gives us a charcoal-gray button with white text and rounded ends, as shown in the following image.

Let’s add some styles for a close button. We’ll use a `.close` class, as shown below:

```
button[type=button] {
    background: #333;
    border: 3px solid #333;
    border-radius: 100px;
    color: white;
    line-height: 1;
    font-size: 6rem;
    font-family: inherit;
    padding: .5rem 1rem;
}
.close {
    width: 9rem;
    height: 9rem;
    background: #c00;
    border: 0;
    border-bottom: 5px solid #c00;
    font-size: 3rem;
    line-height: 0;
    padding: 0;
}
```

Now we have two charcoal-gray buttons with white text and rounded ends.

What’s happening? Our `button[type=button]` selector has a specificity of 0,1,1. However, `.close` is a class selector. Its specificity is only 0,1,0. As a result, most of our `.close` rules don't get applied to `<button type="button" class="close">`.

We can ensure that our `.close` styles are applied by either:

- changing `.close` to `button[type=button].close`
- making `button[type=button]` less specific

The second option adds fewer bytes, so let's use that:

```
[type=button] {
    background: #333;
    border: 3px solid #333;
    border-radius: 100px;
    color: white;
    line-height: 1;
    font-size: 6rem;
    padding: .5rem;
}
.close {
    width: 9rem;
    height: 9rem;
    background: #c00;
    border: 0;
    border-bottom: 5px solid #c00;
    font-size: 3rem;
    line-height: 0;
}
```

Changing the specificity of our selector leaves us with our intended result.

Avoid Chaining Selectors

Another way to minimize specificity is to avoid chaining selectors. Selectors such as `.message.warning` have a specificity of 0,2,0. Higher specificity means they're hard to override. What's more, chaining classes may cause side effects. Here's an example:

```
.message {
    background: #eee;
    border: 2px solid #333;
    border-radius: 1em;
    padding: 1em;
}
```

```
.message.error {
    background: #f30;
    color: #fff;
}
.error {
    background: #ff0;
    border-color: #fc0;
}
```

Using `<p class="message">` with this CSS gives us a nice gray box with a dark gray border.

Using `<p class="message error">`, however, gives us the background of `.message.error` and the border of `.error`, as shown below.

The only way to override a chained selector is to use an even more specific selector. To be rid of the yellow border, we'd need to add a class name or type selector to the chain, such as `.message.warning.exception` or `div.message.warning`. It's more expedient to create a new class instead.

If you do find yourself chaining selectors, go back to the drawing board. Either the design has inconsistencies, or you're chaining prematurely in an attempt to prevent problems you don't yet have. The maintenance headaches you'll prevent and the flexibility you'll gain are worth it.

Avoid Using ID Selectors

HTML allows an identifier (that is, an `id` attribute) to be used once per document. As a result, rule sets that use ID selectors are hard to repurpose. Doing so typically involves using a list of ID selectors—for example, `#sidebar-feature, #sidebar-sports`, and so on.

Identifiers also have a high degree of specificity, and require longer selectors to override declarations. In the example that follows, we need to use `#sidebar.sports` and `#sidebar.local` to undo the background color of `#sidebar`:

```
/* Avoid doing this in your CSS */
#sidebar {
    float: right;
    width: 25%;
    background: #eee;
}
#sidebar.sports  {
    background: #d5e3ff;
}
#sidebar.local {
    background: #ffcccc;
}
```

Instead, we can use a simple `.sidebar` class selector:

```
.sidebar {
    float: right;
    width: 25%;
    background: #eee;
}
.sports  {
    background: #d5e3ff;
}
.local {
    background: #ffcccc;
}
```

Not only does this save a few bytes, but our `.sports` and `.local` rules can now be used with other elements.

Using an attribute selector such as `[id=sidebar]` avoids the higher specificity of an identifier. Though it lacks the reusability of a class selector, the low specificity means we can avoid selector chaining.

A Case for Higher Specificity

In some circumstances, you may *want* the higher specificity of an ID selector. For example, a network of media sites might wish to use the same navigation bar across all of its web properties. This component must be consistent across sites in the network, and should be hard to restyle. Using an ID selector reduces the chances of those styles being overridden accidentally. You can also achieve this using the `!important` keyword.

Let's discuss a selector such as `#content article.sports table#stats tr:nth-child(even) td:last-child`. Not only is it absurdly long, but with a specificity of 2,3,4, it's also not reusable. How many *possible* instances of this selector can there be in your markup?

Let's make this better. We can immediately trim our selector to `#stats tr:nth-child(even) td:last-child`. It's specific enough to do the job. An even simpler approach is to use a class name such as `.stats`. It's a much shorter selector, and those styles aren't limited to `#stats` tables.

Minimizing Nesting When Using a Preprocessor

Overly long, highly specific selectors are often caused by nested rule sets. Both Sass and Less support nested rule set syntax, which is useful for grouping related styles and saving keystrokes. Take, for example, the following CSS:

```
article {
    margin: 2em auto;
}
article p {
    margin: 0 0 1em;
    font-family: 'Droid Serif','Liberation Serif',serif;
}
```

In both Less and Sass, we can rewrite this to take advantage of nesting:

```
article {
    margin: 2em auto;
    p {
        margin: 0 0 1em;
        font-family: 'Droid Serif','Liberation Serif',serif;
    }
}
```

This gives us a descendant selector, and the output will match the standard CSS above.

It's also possible to nest a rule set inside a nested rule set. Take a look at this example:

```
nav {
    > ul {
        height: 1em;
        overflow: hidden;
        position: relative;

        &::after {
            content: ' ';
            display: block;
            clear: both;
        }
    }
}
```

Here, we've nested styles for `::after` inside a declaration block for `ul`, which itself is nested inside a `nav` declaration block. When compiled, we end up with the following CSS:

```
nav > ul {
    height: 1em;
    overflow: hidden;
    position: relative;
}
nav > ul::after {
    content: ' ';
    display: block;
    clear: both;
}
```

So far, so good. Our selectors aren't terribly long or specific. Now let's look at a more complex example of nesting:

```
article {
    color: #222;
    margin: 1em auto;
    width: 80%;

    &.news {
        h1 {
            color: #369;
            font-size: 2em;

            &[lang]{
                font-style: italic;
            }
        }
    }
}
```

That doesn't seem too egregious, right? Our `[lang]` selector is only four levels deep. Well, take a look at our compiled CSS output:

```
article {
    color: #222;
    margin: 1em auto;
    width: 80%;
}
article.news h1 {
    color: #369;
    font-size: 2em;
}
article.news h1 [lang] {
    font-style: italic;
}
```

Uh-oh! Now we have a couple of high-specificity selectors: `article.news h1` and `article.news h1[lang]`. They use more characters than necessary, and require longer and more specific selectors to override them. Mistakes like this can swell the size of our CSS when repeated across a codebase.

Neither Less nor Sass has a hard limit on how deeply rules can be nested. A good rule of thumb is to avoid nesting your rules by more than three levels. Less nesting results in lower specificity and CSS that's easier to maintain.

Using Type and Attribute Selectors with Caution

It's good to keep specificity low, but be careful about the selectors you use to accomplish that. Type and attribute selectors can be the most bothersome.

Type selectors are element selectors such as `p`, `button`, and `h1`. **Attribute selectors** include those such as `[type=checkbox]`. Style declarations applied to these selectors will be applied to every such element across the site. Let's look at another example using buttons, this time with an element selector:

```
button {
    background: #FFC107;
    border: 1px outset #FF9800;
    display: block;
    font: bold 16px / 1.5 sans-serif;
    margin: 1rem auto;
    width: 50%;
    padding: .5rem;
}
```

This seems innocuous enough. But what if we want to create a button that's styled differently? Let's create a `.tab-rounded` button for tab panels:

```
<div class="tabs">
    <button type="button" role="tab" aria-selected="true"
aria-controls="tab1" id="tab-id" class="tab-rounded">Tab
1</button>
</div>
<div tabindex="0" role="tabpanel" id="tab1" aria-
labelledby="tab-id">
    <p>This is the tab 1 text</p>
</div>
```

Now we need to write CSS to override every line that we don't want our rounded button to inherit from the `button` rule set:

```
.tab-rounded {
    background: inherit;
    border: 1px solid #333;
    border-bottom: 0;  /* We only need borders on three sides
for tabs */
    border-radius: 3px 3px 0 0;
    diplay: inline-block;
    font-weight: normal;
    margin: .5rem 0 0;
    width: auto;
    padding: 1rem 2rem;
}
```

We'd still need many of these declarations to override browser defaults, but what if we assign our `button` styles to a `.default` class instead? We can then drop the `display`, `font-weight` and `width` declarations from our `.tab-rounded` rule set. That's a 21% reduction in size:

```
.default {
    background: #FFC107;
    border: 1px outset #FF9800;
    display: block;
    font: bold 16px / 1.5 sans-serif;
    margin: 1rem auto;
    width: 50%;
    padding: .5rem;
}
.tab-rounded {
    background: inherit;
    border: 1px solid #333;
    border-bottom: 0;
    border-radius: 3px 3px 0 0;
    padding: 1rem 2rem;
}
```

Just as importantly, avoiding type and attribute selectors reduces the risk of styling conflicts. A developer working on one module or document won't inadvertently add a rule that creates a side effect in another module or document.

Choosing What to Name Things

When choosing class-name selectors, use *semantic* class names.

When we use the word **semantic**, we mean *meaningful*. Class names should describe what the rule does or the type of content it affects. Ideally, we want names that will endure changes in the design requirements. Naming things is harder than it looks.

Here are examples of what not to do: `.red-text`, `.blue-button`, `.border-4px`, `.margin10px`. What's wrong with these? They're too tightly coupled to the existing design choices. Using `class="red-text"` to mark up an error message does work. But what happens if the design changes and error messages become black text inside orange boxes? Now your class name is inaccurate, making it tougher for you and your colleagues to understand what's happening in the code.

A better choice in this case is to use a class name such as `.alert`, `.error`, or `.message-error`. These names indicate how the class should be used and the kind of content (error messages) they affect.

Recommended Reading

Philip Walton discusses these and other rules in his article "CSS Architecture". I also recommend Harry Roberts' site CSS Guidelines and Nicolas Gallagher's post "About HTML Semantics and Front-end Architecture" for more thoughts on CSS architecture. For a different take, read "CSS Utility Classes and 'Separation of Concerns'" by Adam Wathan, the creator of Tailwind CSS.

We'll now look at two methodologies for naming things in CSS. Both methods were created to improve the development process for large sites and large teams, but they work just as well for teams of one. It's up to you whether you choose one or the other, neither, or a mix of both. The point of introducing them is to help you to think through approaches for writing your own CSS.

Block-Element-Modifier (BEM)

BEM, or Block-Element-Modifier, is a methodology, a naming system, and a suite of related tools. Created at Yandex, BEM was designed for rapid development by sizable development teams. In this section, we'll focus on the concept and the naming system.

BEM methodology encourages designers and developers to think of a website as a collection of reusable component *blocks* that can be mixed and matched to create interfaces. A **block** is simply a section of a document, such as a header, footer, or sidebar, as illustrated below.

Perhaps confusingly, "block" here refers to the segments of HTML that make up a page or application.

Blocks can contain other blocks. For example, a header block might also contain logo, navigation, and search form blocks, as seen below. A footer block might contain a site map block.

More granular than a block is an **element**. As the BEM documentation explains:

> An element is a part of a block that performs a certain function. Elements are context-dependent: they only make sense in the context of the block they belong to.

A search form block, for example, contains a text input element and a submit button element, as illustrated below. (To clarify, we're using "element" as a shorthand for "design element" rather than in the sense of HTML elements.)

A main content block, on the other hand, might have an article list block. This article list block might contain a series of article promo blocks. And each article promo block might contain image, excerpt, and “Read more” elements, as presented below.

Together, blocks and elements form the basis of the BEM naming convention. According to the rules of BEM:

- block names must be unique within a project
- element names must be unique within a block
- variations of a block—such as a search box with a dark background—should add a modifier to the class name

In a BEM naming system, block and element names are separated by a double underscore (as in `.block__element`). Block and element names are separated from modifier names by a double hyphen (for example, `.block--modifier` or `.block__element--modifier`).

Here’s what BEM looks like using a search form example:

```
<form class="search">
    <div class="search__wrapper">
        <label for="s" class="search__label">Search for:
</label>
        <input type="text" id="s" class="search__input">
        <button type="submit"
class="search__submit">Search</button>
    </div>
</form>
```

A variation of this form with a dark background might use the following markup:

```
<form class="search search--inverse">
    <div class="search__wrapper search__wrapper--inverse">
        <label for="s" class="search__label search_label--
inverse">Search for: </label>
        <input type="text"  id="s" class="search__input
search__input--inverse">
        <button type="submit" class="search__submit
search__submit--inverse">Search</button>
    </div>
</form>
```

Our CSS might look like this:

```
.search {
    color: #333;
}
.search--inverse {
    color: #fff;
    background: #333;
}
.search__submit {
    background: #333;
    border: 0;
    color: #fff;
    height: 2rem;
    display: inline-block;
}
.search__submit--inverse {
    color: #333;
    background: #ccc;
}
```

In both our markup and CSS, `search--inverse` and `search__label--inverse` are *additional* class names. They're not replacements for `search` and `search__label`. Class names are the only type of selector used in a BEM system. Child and descendant selectors may be used, but descendants should also be class names. Element and ID selectors are verboten. Enforcing block and element name uniqueness also prevents naming collisions, which can become a problem among teams.

There are several advantages to this approach:

- it's easy for new team members to read the markup and CSS, and understand its behavior
- adding more developers increases team productivity
- consistent naming reduces the possibility of class-name collisions and side effects
- the CSS is independent of the markup
- the CSS is highly reusable

Learning More about BEM

There's a lot more to BEM than can comfortably fit in a section of a chapter. The BEM site describes this methodology in much greater detail, and also features tools and tutorials to get you started. The Get BEM website is another fantastic resource.

Atomic CSS

Atomic CSS takes a markedly different approach from BEM. Named and explained by Thierry Koblentz of Yahoo in his 2013 piece "Challenging CSS Best Practices", Atomic CSS uses a tight library of class names. These class names are often abbreviated and divorced from the content they affect. In an Atomic CSS system, you can tell what the class name does—but there's no relationship between class names (at least, not those used in the stylesheet) and content types. Today, Tailwind CSS and Tachyons use a similar approach.

Let's illustrate with an example. Below is a set of rules in what we might call a conventional CSS architecture. These rule sets use class names that describe the content to which they apply—a global message box, and styles for "success", "warning", and "error" message boxes:

```
.msg {
    background-color: #a6d5fa;
    border: 2px solid #2196f3;
    border-radius: 10px;
    font-family: sans-serif;
    padding: 10px;
}
.msg-success {
    background-color: #aedbaf;
    border: 2px solid #4caf50;
}
.msg-warning {
    background-color: #ffe8a5;
    border-color:  #ffc107;
}
.msg-error {
    background-color: #faaaa4;
    border-color: #f44336;
}
```

To create an error message box, we'd need to add both the `msg` and `msg-error` class names to the element's `class` attribute:

```
<p class="msg msg-error">An error occurred.</p>
```

Let's contrast this with an atomic system, where each declaration becomes its own class:

```
.bg-a {
    background-color: #a6d5fa;
}
.bg-b {
    background-color: #aedbaf;
}
.bg-c {
    background-color: #ffe8a5;
}
.bg-d {
    background-color: #faaaa4;
}
.bc-a{
    border-color: #2196f3;
}
.bc-b {
    border-color: #4caf50;
}
.bc-c {
    border-color:  #ffc107;
}
.bc-d {
    border-color:  #f44336;
}
.br-1x {
    border-radius: 10px;
}
.bw-2x {
    border-width: 2px;
}
.bss {
    border-style: solid;
}
.sans {
    font-style: sans-serif;
}
.p-1x {
    padding: 10px;
}
```

That’s a lot more CSS. Let’s now recreate our error message component. Using Atomic CSS, our markup becomes:

```
<p class="bw-2 bss p-1x sans br-1x bg-d bc-d">
    An error occurred.
</p>
```

Our markup is also more verbose. But what happens when we create a warning message component?

```
<p class="bw-2 bss p-1x sans br-1x bg-c bc-c">
    Warning: The price for that item has changed.
</p>
```

Two class names changed: `bg-d` and `bc-d` were replaced with `bg-c` and `bc-c`. We've reused five rules. Now, let's create a button:

```
<button type="button" class="p-1x sans bg-a br-
1x">Save</button>
```

Hey now! Here we've reused four rules and avoided adding any more rules to our stylesheet. In a robust atomic CSS architecture, adding a new HTML component such as an article sidebar won't require adding more CSS (though, in reality, it might require adding a little bit more).

Atomic CSS comes with a few advantages:

- it keeps CSS trim by creating highly granular, highly reusable styles, instead of a rule set for every component
- it greatly reduces specificity conflicts by using a system of low-specificity selectors
- it allows for rapid HTML component development once the initial rule sets are defined

Atomic CSS is a bit like using utility classes in your CSS, but taken to the extreme.

The Case Against Atomic CSS

Atomic CSS runs counter to much of the popular advice about writing CSS. It feels almost as wrong as sticking `style` attributes everywhere. Indeed, one of the main criticisms of the Atomic CSS methodology is that it blurs the line between content and presentation. If `class="fl m-1x"` floats an element to the left and adds a 10px margin, what do we do when we no longer want that element to float left?

One answer, of course, is to remove the `fl` class from our element. But now we're changing HTML. The whole point of CSS is to separate markup from presentation. Still, updating the HTML may be a small price to pay for rapid component development and trimmer CSS.

Another Answer: Change the CSS

We could also solve this problem by removing the `.fl {float: left;}` rule from our stylesheet, although that would affect every element with a class name of `fl`.

In Koblentz's original post, he used class names such as `.M-10` for `margin: 10px` and `.P-10` for `padding: 10px`. I hope the issue with such a naming convention is obvious. Changing to a margin of `5px` or `20px` means we'd need to update our CSS *and* our HTML, or have class names that fail to accurately describe their effect.

Using class names such as `p-1x`, as done in this section, resolves that issue. The `1x` part of the class name indicates a ratio rather than a defined number of pixels. If the base padding is `5px` (that is, `.p-1x { padding: 5px; }`), then `.p-2x` would set `10px` of padding. Yes, that's less descriptive of what the class name does, but it also means that we can change our CSS without updating our HTML, and without creating a misleading class name.

An atomic CSS architecture doesn't prevent us from using class names that describe the content. You can still add `.button__close` or `.accordion__trigger` to your code. Such class names are preferable for JavaScript and DOM manipulation.

Know When to Go Your Own Way

In practice, your CSS will include a mix of approaches. You may have class names that describe content or components in addition to utility class names that describe color and layout.

If you don't have full control over the markup, as with some CMS products, neither of these approaches may be useful. You may even need to use long and specific selectors to reach your design goals.

Conclusion

After reading this chapter, you should now know:

- how to organize your CSS for easier development and maintenance
- how browsers determine which CSS rules to use
- why class selectors are the most flexible selector for writing scalable, maintainable CSS
- the basics of BEM and Atomic CSS, and the pros and cons of each

In the next chapter, you'll learn what to do when you notice a bug in your CSS. We'll also discuss a few tools for making your CSS files smaller.

Chapter 3: Debugging and Optimization

On your road to becoming a CSS master, you'll need to know how to troubleshoot and optimize your CSS. How do you diagnose and fix rendering problems? How do you ensure that your CSS creates no performance lags for end users?

It's also important to ensure code quality. Were you a little too verbose with your comments? Are there too many unused selectors? Are your selectors overly specific in a way that could affect performance?

Knowing which tools to use will help you ensure that your front end works well. In this chapter, we'll look at some browser and command-line tools to help you analyze and troubleshoot your CSS.

Browser-based Developer Tools

Most desktop browsers include an element inspector feature that you can use to troubleshoot your CSS. Start using this feature by right-clicking anywhere in the browser viewport and selecting **Inspect** or **Inspect Element** from the menu. Mac user? Press the `Ctrl` key while clicking the element you'd like to inspect. The image below indicates what you can expect to see in Chrome.

You can also press `Ctrl` + `Shift` + `I` (Windows/Linux) or `Cmd` + `Option` + `I` (macOS) to open the developer tools panel. Or use the the application's menu:

- Google Chrome and Microsoft Edge: **Tools** > **Developer Tools** (**More tools** and **Developer tools** on Linux)
- Firefox: **Tools** > **Web Developer**
- Safari: **Develop** > **Show Web Inspector**

In Safari, you may have to enable the **Develop** menu first by going to **Safari** > **Preferences** > **Advanced** and checking the box next to **Show Develop menu in menu bar**.

After opening the developer tools interface, you may need to select the correct panel. In Firefox, this panel is named **Inspector**. In Chrome, Edge, and Safari, it's the **Elements** panel. You'll know you're in the right place when you see HTML on one side of the panel and CSS rules on the other.

Generated Markup

The markup you'll see in the HTML panel is a representation of the DOM. It's generated when the browser finishes parsing the document and may differ from your original markup. Using **View Source** reveals the original markup. Keep in mind that, for some JavaScript applications, there may not be much markup to view.

Using the Styles Panel

Sometimes an element isn't styled as expected. Maybe a typographical change failed to take, or there's less padding around a paragraph than you wanted. You can determine which rules are affecting an element by using the **Styles** panel of the Web Inspector.

Browsers are fairly consistent in how they organize the **Styles** panel. Declarations set using the `style` attribute are listed first, whether they were added to the HTML, or programmatically using JavaScript.

Inline styles are followed by a list of style rules applied via author stylesheets —those written by you or your colleagues. Styles in this list are grouped by media query and/or filename.

Authored style rules precede user agent styles. **User agent styles** are the browser's default styles. They also have an impact on your site's look and feel. In Firefox, you may have to select the **Show Browser Styles** option in order to view user agent styles. You can find this setting in the **Settings** panel. Press the `F1` key when the Developer Tools panel is open.

Properties and values are grouped by selector. The checkbox next to each property lets you enable and disable specific declarations. Click on a property or value to change it.

Identifying Cascade and Inheritance Problems

As you inspect styles, you may notice that some properties appear crossed out. These properties have been overridden either by a cascading rule, a conflicting rule, or a more specific selector, as depicted below.

In the image above, the `background`, `border`, and `font-size` declarations of the `[type=button]` block are displayed with a line through them. These declarations were overridden by those in the `.close` block, which comes after the `[type=button]` block in our CSS file.

Spotting Invalid or Unsupported Properties and Values

You can also use the element inspector to spot invalid or unsupported properties or property values. In Chromium-based browsers, including Edge, invalid or unsupported CSS declarations both have a line through them and an adjacent warning icon, which can be seen below.

Firefox also strikes through unsupported properties and values, but places the warning icon to the right of the declaration.

In the screenshot below, Safari strikes through unsupported rules with a red line, and highlights them with a yellow background and warning icon.

When it comes to basic debugging and identifying inheritance conflicts, it doesn’t matter which browser you choose. Familiarize yourself with all of them, however, for those rare occasions when you need to diagnose a browser-specific issue.

Debugging Flexbox and Grid Layouts

As you work with Grid and Flexbox, you may wonder why your layout is a little off, or why a particular property isn’t working the way you’d expect. Chrome, Edge and Firefox all include Grid and Flexbox inspectors with their developer tools to help you diagnose issues with both modules.

Flexbox Inspection

The Flexbox inspector is only available in Chrome/Edge versions 90 and later.

Launching the Grid or Flexbox inspector works similarly in Chrome, Edge, and Firefox. First, open the browser's developer tools and locate the **Elements** (Chrome/Edge) or **Inspector** panel (Firefox). Next, locate the grid or flex container in the document tree. You'll see a small label next to the element—*grid* if the element is a grid container, and *flex* if it's a flex container. Click that label to display the Grid or Flexbox overlay. The following image shows what the Grid overlay looks like in Edge 92.

For Grid containers, the overlay shows the number for each grid line (the dividing lines of a grid) and highlights each grid gap (the space between each item in a grid). One feature of Grid is the ability to indicate how many columns or rows an element should span by indicating a starting and an ending grid line number. Using the inspector lets you see how grid lines are numbered.

For Flex containers, the overlay highlights the size of the `gap` property, and any space created by the use of `justify-content`, `align-content`, or `align-items`. The following image shows the Firefox Flexbox overlay.

The Flexbox overlays for Chrome and Edge work similarly. Chrome and Edge versions 92 and above also include a feature for tinkering with the `align-*` and `justify-*` properties of Grid and Flexbox. The image below shows the Flex inspector in Edge 92.

Firefox, on the other hand, includes a feature that can help you understand why a flex item may be larger or smaller than you expected. The Firefox flex inspector indicates when a flex item is set to grow or shrink.

We’ll revisit Grid and Flexbox debugging tools in Chapter 5, “Layouts”.

Debugging Responsive Layouts

On-device testing is ideal. During development, however, it’s helpful to simulate mobile device viewports and touch features. All major desktop browsers include a mode for responsive debugging.

Firefox

Firefox calls its responsive debugging feature **Responsive Design Mode**. Look for the phone and tablet icon. You’ll find it on the right side of the developer tools panel.

While in Responsive Design Mode, you can test a range of viewport dimensions. You can also simulate the viewport and pixel density of several Android and iOS devices by selecting an option from the device menu.

You can also switch the viewport’s orientation, adjust its dimensions, or change its pixel density ratio from this menu.

Touch events are disabled when you first enter Responsive Design Mode. You can enable them by clicking the touch icon, or by selecting a specific device from the device menu.

You can also use the Firefox Responsive Design Mode to simulate slow connections. Responsive Design Mode includes the ability to mimic GPRS and LTE mobile speeds as well as Wi-Fi connections with its throttling feature.

Chrome and Microsoft Edge Chromium

In Chrome and Edge versions 79 and above, responsive mode is named **Device Mode**. To use it, click the device icon (pictured below) in the upper-left corner, next to the **Select an element** icon.

Like Firefox, Chrome’s and Edge’s Device Mode lets you mimic several kinds of Android and iOS devices, including older devices such as the Galaxy S5 and iPhone 8. In both browsers, DOM touch events are available to your code when using Device Mode.

In Chrome, the Device Mode’s basic throttling feature (shown below) approximates performance on low-tier and mid-tier devices. You can also use it to simulate being offline.

To mimic network speeds, use the throttling menu found in the developer tools Network panel.

Safari

Safari’s Responsive Design Mode is best for testing layouts in iOS device viewports. To enter Responsive Design Mode, select **Develop** > **Enter**

Responsive Design Mode, or `Cmd` + `Ctrl` + `R`.

Unfortunately, Safari's developer tools are limited by comparison. You can't, for example, mimic slow network speeds. Nor can you simulate touch events. To test touch events, you'll need to use a physical iOS device, or a remote device testing service.

Responsive Design–focused Browsers

Blisk and Polypane are two newer, commercial services that have responsive design and device emulation at their core. Both use Chromium under the hood, so you can't rely on them for debugging issues with non-Chromium browsers. They are, however, perfect for testing and debugging responsive layouts.

Both browsers let you view your layout at multiple viewports in the same window at the same time, so that you don’t have to spend time resizing browser windows.

Debugging for UI Responsiveness

Some CSS properties and values trigger operations called “reflows” and “repaints”, which can have a negative effect on the responsiveness of the user interface. This is especially true for low-powered devices. Let’s look at how to measure UI performance using browser tools. First, however, let’s define “reflow” and “repaint”.

What Are Reflows and Repaints?

A **reflow** is any operation that changes the layout of part or all of a page. Examples include changing the dimensions of an element or updating its left position. They’re expensive, because they force the browser to recalculate the height, width, and position of elements in the document.

Repaints also force the browser to re-render part of the document. Changing the color of a button when in a `:hover` state is one example of a repaint. They’re a bit less troublesome than reflows, however, because they don’t affect the dimensions or positions of nodes.

Reflows and repaints are most often triggered by DOM operations—such as adding or removing elements. Changing the values of properties that affect the dimensions, visibility, or position of an element can also trigger reflows and repaints. CSS Triggers is a good (though dated) starting point for identifying which properties may be causing performance bottlenecks.

Page Loads

Page loads always trigger reflow and repaint operations as the browser parses the initial HTML, CSS, and JavaScript.

It’s difficult to completely banish repaints and reflows from a project. We can, however, identify them and reduce their impact using performance tools.

Performance Tools

Performance tools measure how well your front end behaves, capturing things like frame rate and asset load times. By recording page activity, we can determine which portions of our CSS may be causing performance bottlenecks.

In Microsoft Edge, Chrome and Firefox, you'll use the appropriately named **Performance** panel. Safari calls it **Timelines**.

There are two ways to trigger the profiling tool:

- manually by pressing the **Record** or start button
- programmatically using `console.profile( profileName )`, where `profileName` is the optional name for the profile.

To stop recording, press the stop button, or use `console.profileEnd( profileName )`.

For Testing Only

Both `console.profile()` and `console.profileEnd()` are experimental and non-standard. Don't use them in production!

Performance tools can be a bit befuddling. Each browser displays its data a little bit differently. To see what this looks like in practice, we'll compare two basic documents, examples A and B. In both cases, we're moving a series of `<div>` elements from an x-position of zero to an x-position of 1,000.

Both examples use CSS animations. In example A, however, we'll animate the `left` property. In example B, we'll translate our elements by 1,000 pixels and animate the `transform` property.

Our markup for both is the same:

```
<!DOCTYPE html>
    <html lang="en-US">
    <head>
        <meta charset="utf-8">
        <title>Performance example</title>
        <style type="text/css">
```

```
            /* CSS will go here */
        </style>
    </head>
    <body>
        <div></div>
        <div></div>
        <div></div>
        <div></div>
        <script type="text/javascript" src="toggle-move-
class.js"></script>
    </body>
</html>
```

The JavaScript for both documents is also the same. When the page loads, we’ll add a `running` class to the body that triggers our animation:

```
function startAnimation() {
    document.body.classList.add('running');
}
window.addEventListener( 'load', startAnimation );
```

Some of our CSS is common to both examples:

```
div {
    background: #36f;
    margin-bottom: 1em;
    width: 100px;
    height: 100px;
    /*
     * Shorthand for animation direction,
     * duration, timing function, iteration, and play state
     */
    animation: alternate 2s ease-in 2 paused;
}
.running div {
    animation-play-state: running;
}
```

For example A, we’ll animate the `left` property. Here’s our animation CSS:

```
@keyframes change_left {
    from {
        left: 0px;
    }
    to {
        left: 1000px;
    }
}
div {
    position: relative; /* Element must be positioned in
order for left to work */
    left: 0;
    animation-name: change_left;
}
```

In Safari, animating the `left` property generates lots of layout and rendering operations. It also uses quite a bit of CPU power.

In Chrome and Edge, the profile is similarly red. In this case, however, the uppermost row of red markings indicates dropped frames. Dropped frames can cause what experts call **jank**, or animations that aren't as smooth as they can be. The lower row of markings indicates shifts in page layout. These changes may cause performance lags or rendering glitches for devices with limited memory or slower processors. Animating the left property results in lots of dropped frames in Chrome and Edge, as shown below.

The reason for the style recalculations and repaints has to do with the property we're transitioning: `left`. The `left` property triggers a reflow whenever it's changed, even if that change is caused by an animation or transition.

Now, let's take a look at the CSS for example B:

```
@keyframes translate_left {
    from {
        transform: translateX(0);
    }
    to {
        transform: translate(1000px);
    }
}
div {
    transform: translateX(0);
    animation-name: translate_left;
}
```

This time we're animating the `transform` property from a start value of `translateX(0)` and an ending value of `translateX(1000px)`.

In most browsers, transforms don't trigger reflows. Animating `transform` vastly reduces the amount of layout and rendering operations in Safari. The image below shows the Safari timeline output for a transition of the transform property.

Chrome and Edge drop far fewer frames.

I haven’t yet mentioned Firefox, because Firefox handles both animations similarly. The following image shows the results of animating the `left` property. Firefox 86 managed an average frame rate of 52.58 frames per second.

The next image reflects the performance of animating a translation transform. Firefox 86 had a slightly higher frame rate of 56.56 frames per second.

In practice, choose the property that performs best across the broadest range of browsers.

Identifying Which Lines to Remove

As mentioned earlier in this chapter, properties that affect the size, geometry, or placement of objects can trigger reflow operations. This includes positional properties like `left`, `top`, `right`, and `bottom`, alignment-related properties, and `display`.

Once you know which properties *could* cause trouble, the next step is to test the hypothesis. Disable the property—either with a comment, or by adding a temporary `x-` prefix—and rerun the performance test.

Remember that performance is relative, not absolute or perfect. The goal is improvement: make it perform better than it did before. If a property or effect is unacceptably slow, eliminate it altogether.

More on Web Performance

If you'd like to learn more about how to use browser tools to analyze performance, check out the book *Jump Start Web Performance*, by Craig Buckler.

Minification with CSS Optimizer

Developer tools help you find and fix rendering issues, but what about efficiency? Are our file sizes as small as they can be? For that, we need minification tools.

Minification in the context of CSS means "removing excess characters". Consider, for example, this block of code:

```
h1 {
    font: 16px / 1.5 'Helvetica Neue', arial, sans-serif;
    width: 80%;
    margin: 10px auto 0px;
}
```

That's 98 bytes long, including line breaks and spaces. Let's look at a minified example:

```
h1{font:16px/1.5 'Helvetica Neue',arial,sans-
serif;width:80%;margin:10px auto 0}
```

Now our CSS is only 80 bytes long—an 18% reduction. Fewer bytes, of course, means faster download times and data transfer savings for you and your users.

In this section, we'll look at CSS Optimizer, or CSSO, a minification tool that runs on Node.js. It's available as a plugin for several build tools and workflows, but we'll focus on the command-line interface version.

Of course, using the command-line version of CSSO requires that you get comfortable using a command-line interface. Linux and macOS users can use the Terminal application (`Ctrl` + `Alt` + `T` for most Linux distributions, **Terminal.app** for macOS). If you're using Windows, try the command prompt. Go to the **Start** or **Windows** menu and type "cmd" in the search box. You can also use PowerShell.

Before you install CSSO, you'll need to install Node.js and npm. **Node.js** is a JavaScript runtime that lets you write JavaScript applications that run on your computer or on a server without a browser. **npm** is the package manager for Node.js. Package managers make it easy to install and update the code libraries used in your projects. npm is installed as part of the Node.js installation process, so you'll only need to install one package.

Installing CSSO with npm

Now you can install CSSO. In the command line, type the following:

```
npm install -g csso-cli
```

The `-g` flag installs CSSO globally, so that you can run it from any directory. npm will print a message to your terminal window when installation is complete. The image below shows CSSO being installed using npm on macOS.

Running CSSO with npx

If you do a lot of Node.js development, you may want to avoid installing packages globally. Global packages can cause trouble if multiple projects require different versions of a package. Luckily, npm includes a package **runner**—npx—that lets you run package binaries from a local directory or a central cache.

Because npx is a package *runner*, rather than an *installer*, you'll need to type `npx` and the full name of the package every time you want to run CSSO:

```
npx csso-cli
```

The first time you use this command, npx will alert you that it needs to install `csso-cli`. Confirm that you want to do so when prompted.

Now you’re ready to minify your CSS.

Using CSSO

To minify CSS files, run the `csso` command, passing the name of a file as an argument:

```
csso style.css
```

If you’re using npx, you’ll need to type a few more characters:

```
npx csso-cli style.css
```

This performs basic compression. CSSO strips unneeded whitespace, removes superfluous semicolons, and deletes comments from your CSS input file.

Once complete, CSSO prints the optimized CSS to standard output, meaning the current terminal or command prompt window. In most cases, however, we’ll want to save that output to a file. To do that, pass an output argument to `csso` using the `--output` or shorter `-o` flag. For example, if we wanted to save the minified version of `style.css` as `style.min.css`, we’d use the following:

```
csso style.css -o style.min.css
```

If you’ve chosen npx, use this:

```
npx csso-cli style.css -o style.min-css
```

By default, CSSO also restructures your CSS. It will, for example, merge the declaration blocks of duplicate selectors and remove some redundant properties. Consider the following CSS:

```
body {
    margin: 20px 30px;
    padding: 100px;
    margin-left: 0px;
}
h1 {
    font: 200 36px / 1.5 sans-serif;
}
h1 {
    color: #ff6600;
}
```

In this snippet, `margin-left` overrides the earlier `margin` declaration. We've also repeated `h1` as a selector for consecutive declaration blocks. After optimization and minification, we end up with this:

```
body{padding:100px;margin:20px 30px 20px 0}h1{font:200
36px/1.5 sans-serif;color:#f60}
```

CSSO has removed extraneous spaces, line breaks, and semicolons, and shortened `#ff6600` to `#f60`. CSSO also merged the `margin` and `margin-left` properties into one declaration (`margin: 20px 30px 20px 0`) and combined our separate `h1` selector blocks into one.

If you're skeptical about how CSSO will rewrite your CSS, you can disable its restructuring features. Use the `--no-restructure` flag. For example, running `csso style.css -o style.min.css --no-restructure` gives us the following:

```
body{margin:20px 30px;padding:100px;margin-left:0}h1{font:200
36px/1.5 sans-serif}h1{color:#f60}
```

Now our CSS is minified, but not optimized. Our `margin-left: 0` declaration remains. Disabling restructuring will keep your CSS files from being as small as they could be. Avoid disabling restructuring unless you encounter a problem.

Preprocessors and post-processors (such as Sass, Less and PostCSS) offer minification as part of their toolset. However, using CSSO may shave additional bytes from your files.

Enforcing Code Quality with stylelint

stylelint is a linting tool. A **linter** is an application that checks code for potential trouble spots, and enforces coding conventions according to a set of rules. You can, for instance, use linters to enforce tabs instead of spaces for indentation. stylelint can find problems such as duplicate selectors, invalid rules, or unnecessary specificity. These have the greatest impact on CSS maintainability.

Install stylelint as you would any other npm package:

```
npm install -g stylelint
```

Once installed, we'll need to configure stylelint. Start by installing the standard configuration:

```
npm install -g stylelint-config-standard
```

Next, create a file named `.stylelintrc` in your home directory. We'll use this file to configure stylelint. Placing it in your home directory makes it available to all of your projects.

Configuring stylelint for Each Project

You can also configure stylelint on a per-project basis by adding a `.stylelintrc` file to your project directory.

The `.stylelintrc` file can use JSON (JavaScript Object Notation) or YAML (YAML Ain't Markup Language) syntax. We'll use JSON here.

File Names and Syntax Highlighting

You may prefer to name your file `.stylelintrc.json` or `.stylelintrc.yaml` so that your text editor applies the appropriate syntax highlighting. Both file extensions are supported by stylelint.

Let's extend the standard configuration. Add the following to your `.stylelintrc` file:

```
{
    "extends": "stylelint-config-standard"
}
```

This is enough to start linting our CSS.

Using stylelint and Understanding Its Output

Type the following in the command line:

```
stylelint style.css
```

To recursively lint all CSS files in a directory, use the following:

```
stylelint "./css/**/*.css"
```

stylelint can also lint CSS that's embedded in HTML files using the `<style>` element. Pass the path to an HTML file as the argument. When stylelint finishes analyzing your CSS, you'll see output resembling what's shown below.

The first column indicates the line number and character position of the rule violation. For example, `4:2` indicates that our first rule violation occurs on line 4 of `style.css`, beginning with the second character.

Next, stylelint describes what it expected to see based on the rules defined in the standard configuration. `Expected "#FFFFFF" to be "#ffffff"` indicates that we should have used lowercase hexadecimal color notation. The last column indicates the name of the violated rule—`color-hex-case`.

Our second error occurs on the same line. We've also violated the `color-hex-length` rule by using a six-digit hexadecimal color value instead of a more concise, three-digit version.

As you can see from this output, the standard configuration is quite opinionated. It prefers an indentation of two spaces instead of tabs, and enforces a rule of one selector per line. Let's make a few changes.

Configuring stylelint's Rules

We can add or override stylelint rules by adding a `rules` property to `.stylelintrc`. Remember that our `.stylelintrc` file uses JSON. Each property needs to be enclosed in double straight quotation marks:

```
{
    "extends": "stylelint-config-standard",
    "rules": {}
}
```

Now let's decide what we'd like to change. We can view our current configuration using the following command:

```
stylelint --print-config ./
```

The `--print-config` flag requires a file path. We've installed stylelint and our configuration globally, so we can use any directory—in this case, our current path. Refer to the stylelint user guide for a complete list of rules and options.

Let's change our current indentation from spaces to tabs. We'll need to modify the `indentation` rule by adding it as a property of our `rules` object:

```
{
    "extends": "stylelint-config-standard",
    "rules": {
        "indentation": "tab"
    }
}
```

Tabs or Spaces?

“Tabs or spaces” is the subject of much developer debate. Spaces provide more consistent visual spacing across editors. Tabs, on the other hand, let individual developers adjust visual spacing to their needs while maintaining consistent spacing in the code. Choose either one, but stick to your choice.

Let’s also enforce single-line selector lists, with a space after each comma. In other words, we’ll allow selector lists formatted as follows:

```
h1, h2, h3, h4 {
    font-family: 'geomanistbold';
}
```

But let’s also disallow multi-line selector lists, such as the example below:

```
h1,
h2,
h3,
h4 {
    font-family: 'geomanistbold';
}
```

Choose Your Own Custom Rules

Whether to use single-line selector lists or multi-line selector lists is strictly a matter of preference. We’re enforcing a single-line selector list in this example to demonstrate how to customize a stylelint configuration.

For this, we’ll need to use a combination of four rules: `selector-list-comma-newline-after`, `selector-list-comma-newline-before`, `selector-list-comma-space-after` and `selector-list-comma-space-before`:

```
{
    "extends": "stylelint-config-standard",
    "rules": {
        "indentation": "tab",
        "selector-list-comma-newline-after": "never-multi-
line",
        "selector-list-comma-newline-before": "never-multi-
line",
        "selector-list-comma-space-before": "never",
        "selector-list-comma-space-after": "always"
    }
}
```

Now when we run `stylelint`, we’ll see the following output. Notice that stylelint no longer makes a fuss about our indentation.

In Chapter 2, I mentioned selector specificity and how it affects CSS maintenance. stylelint includes a few rules that let us enforce low specificity. We’ll use one of them here: `selector-max-specificity`. Let’s edit `.stylelintrc` again to limit specificity to a maximum of 0,2,1:

```
{
    "extends": "stylelint-config-standard",
    "rules": {
        "indentation": "tab",
        "selector-list-comma-newline-after": "never-multi-
line",
        "selector-list-comma-newline-before": "never-multi-
line",
        "selector-list-comma-space-before": "never",
        "selector-list-comma-space-after": "always",
        "selector-max-specificity": "0,2,1"
    }
}
```

With this rule, selectors such as `.subnav a:hover` won't trigger warnings or errors, but selectors such as `#menu` or `.features p:first-of-type::first-letter` will.

It's not necessary to use a preset configuration with stylelint. You could remove the `"extends"` property and set your own style preferences using `"rules"`. A custom configuration may be the wiser choice if you find yourself overriding more than a few standard rules.

Using stylelint with npx

As mentioned above, global npm packages can cause troubles if you work on multiple Node.js projects. The good news is that we can use stylelint with npx too. That said, the process of using stylelint with npx is a little bit different.

First, we'll need to install a local, directory-level version of `stylelint-config-standard`:

```
npm install stylelint-config-standard
```

Using `npm install` without the `-g` flag installs the `stylelint-config-standard` package in our current directory. It's only available within this directory, rather than system-wide. Now we can use `npx` to run stylelint:

```
npx stylelint style.css
```

Keep in mind that we only need to install `stylelint-config-standard` if we plan to use it in our `.stylelistrc` configuration. Otherwise, you can skip that step and just use whatever rules you've defined using the `rules` property.

Consider a Task Runner or Build Tool

Running these tools probably seems like a lot of extra work. To that end, consider adding a task runner or build system to your workflow. Popular ones include Grunt, Gulp, webpack and Broccoli.js. All four have robust documentation and sizable developer communities.

What's great about these task runners and build systems is that they automate concatenation and optimization tasks. They're not limited to CSS either. Most build tools also optimize JavaScript and images.

Because the configuration and build script files are typically JSON and JavaScript, you can easily reuse them across projects or share them with a team. Both CSSO and stylelint can be integrated with Grunt, Gulp, webpack, or Broccoli with the help of a plugin.

Above all, however, take a pragmatic approach to building your toolkit. Add tools that you think will enhance your workflow and improve the quality of your output.

Conclusion

In this chapter, we've looked at some tools to help you diagnose, debug, and optimize your CSS. In the next chapter, we'll look at how to work with variables in CSS.

Chapter 4: Custom Properties

For years, variables were one of the most commonly requested CSS features. It took years to work through the details of the syntax and decide how variables would fit into existing rules governing cascade and inheritance. Now they're available to developers in the form of CSS **custom properties**.

Custom properties make it easier to manage colors, fonts, size, and animation values, and ensure their consistency across a codebase. In this chapter, we'll look at:

- how to define properties and set default values for those properties
- how custom properties interact with cascade and inheritance rules
- how to use custom properties with media queries
- how to use custom properties and the HSL color space to generate color palettes

By the end, you should have a good grasp of how to use custom properties in your projects.

Defining a Custom Property

To define a custom property, select a name and prefix it with two hyphens. Any alphanumeric character can be part of the name. Hyphen (`-`) and underscore (`_`) characters are also allowed. A broad range of Unicode characters can be part of a custom property name. This includes emoji, but for the sake of clarity and readability, stick to alphanumeric names.

Here's an example:

```
--primarycolor: #0ad0f9ff; /* RGB alpha hexadecimal color
notation */
```

The `--` indicates to the CSS parser that this is a custom property. When used as a variable, the parsing engine replaces the property with its value.

Custom property names are *case-sensitive*. That means `--primaryColor` and `--primarycolor` are considered two distinct property names. That's a departure from traditional CSS, in which property and value case don't matter. It is, however, consistent with the rules for variable names in ECMAScript.

As with other properties, such as `display` or `font`, CSS custom properties must be defined within a declaration block. One common pattern is to define custom properties using the `:root` pseudo-element as a selector:

```
:root {
    --primarycolor: #0ad0f9ff;
}
```

`:root` is a pseudo-element that refers to the root element of the document. For HTML documents, that's the `<html>` element. For SVG documents, it's the `<svg>` element. Using `:root` makes properties immediately available throughout the document.

Using Custom Properties

To use a custom property as a variable, we need to use the `var()` function. For instance, if we wanted to use our `--primarycolor` custom property as a background color, we'd do the following:

```
body {
    background-color: var(--primarycolor);
}
```

Our custom property's value will become the computed value of the `background-color` property.

To date, custom properties can only be used as variables to set values for standard CSS properties. You can't, for example, store a property *name* as a variable and then reuse it. The following CSS won't work:

```
:root {
    --top-border: border-top; /* Can't set a property as
custom property's value */
    var(--top-border): 10px solid #bc84d8; /* Can't use a
variable as a property */
}
```

You also can’t store a property–value *pair* as a variable and reuse it. The following example is also invalid:

```
:root {
    --text-color: 'color: orange'; /* Invalid property value
*/
}
body {
    var(--text-color);             /* Invalid use of a
property */
}
```

Lastly, you can’t concatenate a variable as part of a value string:

```
:root {
    --base-font-size: 10;
}
body {
    font: var(--base-font-size)px / 1.25 sans-serif; /*
Invalid CSS syntax */
}
```

“Custom properties” is a future-proof name that accounts for how this feature might be used someday. This could change, however, should the CSS Extensions specification be implemented by browser vendors. That specification defines ways to extend CSS with custom selector combinations, functions, and at-rules.

We commonly call custom properties “variables”, and to date, that’s the only way we can use them. In theory, they’re not entirely interchangeable terms. In practice and for now, they are. I’ll mostly use *custom properties* in this chapter, since that’s their proper name. I’ll use *variables* when it makes the sentence clearer.

Setting a Fallback Value

The `var()` function accepts up to two arguments. The first argument should be a custom property name. The second argument is optional, but must be a declaration value. This declaration value functions as a fallback or default value that's applied when the custom property value isn't defined.

Let's take the following CSS:

```
.btn__call-to-action {
    background: var(--accent-color, deepskyblue);
}
```

If `--accent-color` is defined—let's say its value is `#f30`—then the fill color for any path with a `.btn__call-to-action` class attribute will have a red-orange fill. If it's not defined, the fill will be a deep sky blue.

Declaration values can also be nested. In other words, you can use a variable as the fallback value for the `var` function:

```
body {
    background-color: var(--books-bg, var(--arts-bg));
}
```

In the CSS above, if `--books-bg` is defined, the background color will be set to the value of the `--books-bg` property. If not, the background color will instead be whatever value was assigned to `--arts-bg`. If neither of those are defined, the background color will be the initial value for the property—in this case, `transparent`.

Something similar happens when a custom property has a value that's invalid for the property it's used with. Consider the following CSS:

```
:root {
    --text-primary: #600;
    --footer-link-hover: #0cg; /* Not a valid color value */
}
body {
    color: var(--text-primary);
}
a:link {
    color: blue;
}
a:hover {
    color: red;
}
footer a:hover {
    color: var(--footer-link-hover);
}
```

In this case, the value of the `--footer-link-hover` property is not a valid color. Instead, `footer a:hover` inherits its color from that of the `<body>` element.

Custom properties are resolved in the same way other CSS values are resolved. If the value is invalid or undefined, the CSS parser will use the inherited value if the property is inheritable (such as `color` or `font`), and the initial value if it's not (as with `background-color`).

Custom Properties and the Cascade

Custom properties also adhere to the rules of the cascade. Their values can be overridden by subsequent rules:

```
:root {
    --text-color: #190736; /* navy */
}
body {
    --text-color: #333;  /* dark gray */
}
body {
    color: var(--text-color);
}
```

In the example above, our body text would be dark gray. We can also reset values on a per-selector basis. Let's add a couple more rules to this CSS:

```
:root {
    --text-color: #190736; /* navy */
}
body {
    --text-color: #333;   /* dark gray */
}
p {
    --text-color: #f60;  /* orange */
}
body {
    color: var(--text-color);
}
p {
    color: var(--text-color)
}
```

In this case, any text that's wrapped in `<p>` element tags would be orange. But text within `<div>` or other elements would still be dark gray.

You can also set the value of a custom property using the `style` attribute—for example, `style="--brand-color: #9a09af"`.

Custom Properties and Color Palettes

Custom properties work especially well for managing HSL color palettes. **HSL** stands for *hue, saturation, lightness*. It's a light-based color model that's similar to RGB. We can use HSL values in CSS thanks to the `hsl()` and `hsla()` color functions. The `hsl()` function accepts three arguments: hue, saturation, and lightness. The `hlsa()` function also accepts a fourth argument, indicating the color's alpha transparency (a value between 0 and 1).

While an RGB system expresses color as proportions of red, green, and blue, HSL uses a color circle where hue is a degree position on that circle, and the tone or shade are defined using saturation and lightness values. Saturation can range from 0% to 100%, where 0% is gray and 100% is the full color. Lightness can also range from 0% to 100%, where 0% is black, 100% is white, and 50% is the normal color.

Chromatic Wheel by CrazyTerabyte from Openclipart.

In the HSL color system, the primary colors red, green, and blue are situated 120 degrees apart at 0 degrees/360 degrees, 120 degrees, and 240 degrees. Secondary colors—cyan, magenta, and yellow—are also 120 degrees apart, but sit opposite the primary colors, at 180 degrees, 300 degrees, and 60 degrees/420 degrees respectively. Tertiary, quaternary, and other colors fall in between at roughly ten-degree increments. Blue, written using HSL notation, would be `hsl(240, 100%, 50%)`.

HSL Argument Units

When you use a unitless value for the first argument of the `hsl()` and `hsla()` functions, browsers assume that it's an angle in degree units. You can, however, use any supported CSS angle unit. Blue can also be expressed as `hsl(240deg, 100%, 50%)`, `hsl(4.188rad, 100%, 50%)` or `hsla(0.66turn, 100% 50%)`.

Here's where it gets fun. We can set our hue values using a custom property, and set lighter and darker shades by adjusting the saturation and lightness value:

```
:root {
    --brand-hue:      270deg;  /* purple */
    --brand-hue-alt:  .25turn; /* green */

  /*
    hsl() and hsla() can accept comma-separated or space-
separated arguments,
    but older browsers (such as Internet Explorer 11) only
support
    comma-separated arguments.
  */

    --brand-primary:   hsl( var( --brand-hue ) 100% 50% );
    --brand-highlight: hsl( var( --brand-hue ) 100% 75% );
    --brand-lowlight:  hsl( var( --brand-hue ) 100% 25% );
    --brand-inactive:  hsl( var( --brand-hue )  50% 50% );

    --brand-secondary:     hsl( var( --brand-hue-alt ) 100%
50% );
    --brand-2nd-highlight: hsl( var( --brand-hue-alt ) 100%
75% );
    --brand-2nd-lowlight:  hsl( var( --brand-hue-alt ) 100%
25% );
    --brand-2nd-inactive:  hsl( var( --brand-hue-alt )  50%
50% );
}
```

The CSS above gives us the palette shown below.

This is a simple version, but you can also use custom properties to adjust saturation and lightness values.

Robust Palette Generation

Dieter Raber discusses a technique for robust palette generation in "Creating Color Themes With Custom Properties, HSL, and a Little calc()".

Another idea is to combine custom properties and the `calc()` function to generate a square color scheme from a base hue. Let's create a square color scheme in our next example. A **square color scheme** consists of four colors that are equidistant from each other on the color wheel—that is, 90 degrees apart:

```
:root {
    --base-hue: 310deg; /* Hot pink */
    --distance: 90deg;

    --color-a: hsl( var(--base-hue), 100%, 50% );
    --color-b: hsl( calc( var(--base-hue) + var( --distance )
), 100%, 50% );
    --color-c: hsl( calc( var(--base-hue) + ( var( --distance
) * 2 ) ), 100%, 50% );
    --color-d: hsl( calc( var(--base-hue) + ( var( --distance
) * 3 ) ), 100%, 50% );
}
```

This bit of CSS gives us the rather tropical-inspired color scheme shown below.

Custom properties also work well with media queries, as we’ll see in the next section.

Using Custom Properties and Media Queries

We can also use custom properties with media queries (we’ll take a deeper look at media queries in Chapter 10, “Applying CSS Conditionally”. For example, you can use custom properties to define light and dark color schemes:

```
:root {
    --background-primary: hsl(34, 78%, 91%);
    --text-primary: hsl(25, 76%, 10%);
    --button-primary-bg: hsl(214, 77%, 10%);
    --button-primary-fg: hsl(214, 77%, 98%);
}
@media screen and ( prefers-color-scheme: dark ) {
    :root {
      --background-primary: hsl(25, 76%, 10%);
      --text-primary: hsl(34, 78%, 91%);
      --button-primary-bg: hsl(214, 77%, 98%);
      --button-primary-fg: hsl(214, 77%, 10%);
  }
}
```

Similarly, we can use custom properties to change the base font size for screen versus print:

```
:root {
    --base-font-size: 10px;
}
@media print {
    :root {
        --base-font-size: 10pt;
    }
}
html {
    font: var(--base-font-size) / 1.5 sans-serif;
}
body {
    font-size: 1.6rem;
}
```

In this case, we're using media-appropriate units for print and screen. For both media, we'll use a base font size of 10 units—pixels for screen, points for print. We'll also use the value of `--base-font-size:` to set a starting size for our root element (`html`). We can then use `rem` units to size our typography relative to the base font size.

Rem Values

As defined in the CSS Values and Units Module Level 3 specification, a `rem` unit is always "equal to the computed value of font-size on the root element". If the root element's computed value of `font-size` is 10px, `1.6rem` will create a computed value of `16px`.

Custom properties can help us write simpler, more maintainable CSS.

Using Custom Properties with JavaScript

Remember: custom properties are CSS properties, and we can interact with them as such. For example, we can use the `CSS.supports()` API to test whether a browser supports custom properties:

```
const supportsCustomProps = CSS.supports('--primary-text:
#000');

// Logs true to the console in browsers that support custom
properties
console.log(supportsCustomProps);
```

You can learn more about the `CSS.supports()` API, as well as the `@supports` CSS rule, in Chapter 10, "Applying CSS Conditionally".

We can also use the `setProperty()` method to set a custom property value:

```
document.body.style.setProperty('--bg-home', 'whitesmoke');
```

Using `removeProperty()` works similarly. Just pass the custom property name as the argument:

```
document.body.style.removeProperty('--bg-home');
```

To use the custom property as a value with JavaScript, use the `var()` function with the property name as its argument:

```
document.body.style.backgroundColor = 'var(--bg-home)';
```

Alas, you can't set custom properties using square-bracket syntax or camelCased properties of the style object. In other words, neither `document.body.style.--bg-home` nor `document.body.style['--bg-home']` will work.

Custom Properties and Components

JavaScript frameworks like React, Angular and Vue let developers use JavaScript to create reusable, sharable blocks of HTML, often with CSS that's defined at the component level.

Here's an example of a React component, written in **JSX**, a syntax extension for JavaScript. It resembles XML, and gets compiled into HTML or XML. It's a common way of building React components:

```
import React from 'react';

/* Importing the associated CSS into this component */
import '../css/field-button.css';

class FieldButtonGroup extends React.Component {
    render() {
        return (
            <div className="field__button__group">
                <label htmlFor={this.props.id}>
{this.props.labelText}</label>
                <div>
                    <input type={this.props.type}
                      name={this.props.name}
                      id={this.props.id}
                      onChange={this.props.onChangeHandler}
/>
                    <button type="submit">
{this.props.buttonText}</button>
                 </div>
            </div>
        );
    }
}

export default FieldButtonGroup;
```

More on JavaScript Frameworks

SitePoint has extensive resources on React, Angular and Vue if you want to learn more about working with JavaScript frameworks. For React, check out *Your First Week With React* and extensive React articles. For Angular, there's *Learn Angular: Your First Week* and plenty of Angular articles and tutorials. For Vue, check out *Jump Start Vue.js* and more Vue articles.

Our React component imports CSS into a JavaScript file. When compiled, the contents of `field-button.css` are loaded inline. Here's one possible way to use this with custom properties:

```
.field__button__group label {
    display: block;
}
.field__button__group button {
    flex: 0 1 10rem;
    background-color: var(--button-bg-color, rgb(103, 58,
183)); /* include a default */
    color: #fff;
    border: none;
}
```

In this example, we've used a custom property—`--button-bg-color`—for the button's background color, along with a default color in case `--button-bg-color` never gets defined. From here, we can set a value of `--button-bg-color`, either in a global stylesheet or locally via the `style` attribute.

Let's set the value as a React "prop". React **props** (short for *properties*) mimic element attributes. They're a way to pass data into a React component. In this case, we'll add a prop named `buttonBgColor`:

```
import FieldButtonGroup from '../FieldButtonGroup';

class NewsletterSignup extends React.Component {
    render() {
        // For brevity, we've left out the onChangeHandler
prop.
        return (
            <FieldButtonGroup type="email" name="newsletter"
id="newsletter"
              labelText="E-mail address"
buttonText="Subscribe"
              buttonBgColor="rgb(75, 97, 108)" />
        );
    }
}

export default NewsletterSignup;
```

Now we need to update our `FieldButtonGroup` to support this change:

```
class FieldButtonGroup extends React.Component {
    render() {
        /*
        In React, the style attribute value must be set using
a JavaScript
```

```
        object in which the object keys are CSS properties.
Properties
        should either be camelCased (e.g. backgroundColor) or
enclosed in
        quotes.
        */

        const buttonStyle = {
            '--button-bg-color': this.props.buttonBgColor
        };

        return (
            <div className="field__button__group">
                <label htmlFor={this.props.id}>
{this.props.labelText}</label>
                <div>
                    <input type={this.props.type}
                      name={this.props.name} id=
{this.props.id}
                      onChange={this.props.onChangeHandler}
/>
                    <button type="submit" style=
{buttonStyle}>
                      {this.props.buttonText}
                    </button>
                </div>
            </div>
        );
    }
}
```

In the code above, we've added a `buttonStyle` object that holds the name of our custom property and sets its value to that of our `buttonBgColor` prop, and a `style` attribute to our button.

Using the `style` attribute probably runs counter to everything you've been taught about writing CSS. A selling point of CSS is that we can define one set of styles for use across multiple HTML and XML documents. The `style` attribute, on the other hand, limits the scope of that CSS to the element it's applied to. We can't reuse it. And we can't take advantage of the cascade.

But in a component-based, front-end architecture, one component may be used in multiple contexts, by multiple teams, or may even be shared across client projects. In those cases, you may want to combine the “global scope” of the cascade with the narrow “local scope” provided by the `style` attribute.

Setting the custom property value with the `style` attribute limits the effect to *this particular instance* of the `FieldButtonGroup` component. But because we’ve used a custom property instead of a standard CSS property, we still have the option of defining `--button-bg-color` in a linked stylesheet instead of as a component prop.

Conclusion

Custom properties take one of the best features of pre-processors—variables—and make them native to CSS. With custom properties, we can:

- create reusable, themed components
- easily adjust padding, margins, and typography for a range of viewport sizes and media
- improve the consistency of color values in our CSS

Variables have a range of applications, and are particularly useful in component-based design systems.

I hope you’re leaving this chapter with a better understanding of how to use custom properties. In the next chapter, we’ll dive into how to create layouts with CSS, including flexible boxes, grids, and shapes.

Chapter 5: Layouts

CSS layouts have come a long way in the last decade. In an earlier era of the Web, we wrestled and wrangled `<div>` tags or used heavy CSS frameworks that relied on floats and clearing. Or we threw a bunch of JavaScript at them. These days, it's much easier to create the kinds of complex layouts that used to require nested elements, extensive knowledge of browser quirks, or expensive DOM operations.

In this chapter, we'll look at several aspects of CSS layout. In the first half, we'll review some of the basics: normal flow, floated elements, and how to clear floats. We'll follow that up with refreshers on both the box model and stacking context. Understanding these concepts helps us diagnose and fix layout bugs.

In the second half of this chapter, we'll look at shapes, multicolumn layout, flexible box layout (better known as *Flexbox*), and Grid.

This chapter is long and dense. But by the end of it, you'll have a good sense of how to create layouts that are robust and adaptable.

Display Types and Normal Flow

One of the most important points to understand about CSS is that *everything is a box*.

During the parsing and layout process, browsers generate one or more boxes for each element, based on its *display type*.

Display types are a newer CSS concept, introduced in the CSS Display Module Level 3 specification. There are two of them: inner and outer. The **inner** display type affects how the descendants of an element—what's *inside* the box—are arranged within it. **Outer** display types affect how the element's box behaves in *flow layout* or *normal flow*. Display type is determined by the computed value of an element's `display` property.

In practical terms, this means that there are two display box types that participate in normal flow:

- **block-level** boxes that participate in a *block formatting context*
- **inline-level** boxes that participate in an *inline formatting context*

Formatting context is a fancy way of saying that an element *behaves according to the rules for boxes of this type*.

Both `block` and `inline` are outer display values. The `block` value triggers a block formatting context for an element's **principal box**, or its outermost, containing box. Using `inline` triggers an inline formatting context.

Inner display types include the `flex`/`inline-flex`, `grid`/`inline-grid`, and `table` values for the `display` property. These properties tell the browser how to lay out contents inside the principal box. They also provide a shorthand way to tell the browser to *treat the outer box as a block-level (or inline-level) box, but arrange the stuff inside it according to the rules of its formatting context*.

Block Formatting versus Inline Formatting

Block-level boxes are stacked in the order in which they appear in the source document. In a horizontal writing mode, they stack vertically from the top to the bottom of the screen.

Writing Modes

If you need a refresher on writing modes, refer to Chapter 6, "Working with Text".

In vertical modes, they sit horizontally—side by side and across the screen. With the exception of `display: table` and its related properties, block-level boxes also expand to fill the available width of their containing element.

Browsers generate a block-level box when the computed value of the `display` property is one of the following:

- `block`
- `list-item`
- `table` or any of the `table-*` values such as `table-cell`
- `flex`
- `grid`
- `flow-root`

Other property–value combinations can also trigger block-level box behavior and a block formatting context. Multicolumn containers, for example, trigger a block formatting context when the value of `column-count` or `column-width` is something other than `auto`. Using `column-span: all` also triggers a block formatting context. We'll discuss multicolumn layout later in this chapter.

Floating or positioning an element (with `position: absolute` or `position: fixed`) also triggers a block formatting context. So does the `contain` property when its value is `layout`, `content`, or `strict`.

Inline-level boxes, by contrast, don't form new blocks of content. Instead, these boxes make up the lines inside a block-level box. They're displayed horizontally and fill the width of the containing box, wrapping across lines if necessary, as shown in the image below, which shows an inline box with `margin: 1em` and `padding: 5px` applied.

Inline-level boxes have a `display` value of `inline`, `inline-block`, `inline-table`, or `ruby`.

Just about every browser ships with a user agent stylesheet that sets default rules for element selectors. These stylesheets typically add a `display: block` rule for elements such as `<section>`, `<div>`, `<p>`, and `<ul>`. Most *phrasing content* elements—such as `<a>`, `<span>`, and `<canvas>`—use the initial value of `display`, which is `inline`. When you view a document without any developer-authored CSS, you're really seeing the computed values from the browser's own stylesheet.

User agent stylesheets also set default styles for the root SVG element, particularly when SVG documents are combined with HTML. However, SVG documents rely on a coordinate system for layout instead of the box model. SVG elements *do* create a bounding box, but elements within the bounding box don't participate in the box model or normal flow, and don't affect the position of other elements in the document. As a result, most layout-related CSS properties don't work with SVG elements. We'll discuss that in greater depth in Chapter 12, "Using CSS with SVG".

Logical Properties

Logical properties are closely related to block formatting. Defined by the Logical Properties and Values Level 1 specification, they affect the dimensions and position of elements.

Properties such as `margin-left` and `width` use directional or physical features of the viewport. Logical properties, on the other hand, are *flow-relative*. They're affected by the value of the `direction` and `writing-mode` properties, and fall into two broad categories:

- properties that affect the block direction
- properties that affect the inline direction

For example, when the writing mode is horizontal, `inset-block-start` and `inset-block-end` are the top and bottom of the container respectively, as pictured below.

For vertical writing modes, however, `inset-block-start` and `inset-block-end` are the physical left and right of the container, as shown below.

The `block-size` property determines the *vertical* dimension of block-level elements when the writing mode is horizontal, and `inline-size` determines its *horizontal* dimension. They're the equivalent of `height` and `width` respectively. When the writing mode is vertical, the inverse is true: `block-`

`size` is the equivalent of `width`, and `inline-size` is the equivalent of `height`.

The Logical Properties specification also adds longhand properties for margins, padding, and borders. For example, the flow-relative alternative to `margin-top` is `margin-block-start`.

Be aware that browsers map the `margin`, `padding`, and `border-width` shorthand properties to top, right, bottom, and left. For a declaration such as `border-width: 1rem`, this is fine. But a declaration such as `border-width: 10rem 1rem 1rem 1rem` creates a physical, `10rem` top border instead of a flow-relative one. You *must* use the longhand `border-block-*`, `margin-block-*` and `padding-block-*` properties if you want flow-relative borders, margins, or padding.

You'll see flow-relative properties sprinkled throughout this chapter.

Box Dimensions and the Box Model

How does the browser calculate the dimensions of a block? Box dimensions are the sum of the box's content area, plus its padding size and border size, as defined by the CSS Level 2 specification. The margin size creates a *margin box* for the element.

Margin boxes affect the placement of other elements in the document, but the size of the margin has no effect on the dimensions of the box itself.

Adjacent margin boxes also **collapse**. If two paragraph elements have top and bottom margins of 20 pixels, the margin space between them will be 20 pixels —not 40 pixels.

When Margins Don't Collapse

In some formatting contexts, such as Grid, margins *do not* collapse. We'll discuss this in the "Creating Layouts with CSS Grid" section below.

For instance, a `<p>` element with `width: 300px`, `padding: 20px`, and `border: 10px`, has a calculated width of 360 pixels. That's the sum of its width, left and right padding, and left and right `border-width` properties. To create an element that's 300 pixels wide with 20 pixels of padding and a ten-pixel border, the `width` needs to be `240px`.

Let's take a brief detour to add some historical context. Although today's browsers calculate the width as I've just described, Internet Explorer 5.5 didn't. Instead, IE5.5 used the `width` property as the final arbiter of box dimensions, with padding and border drawn inside the box, as shown in the image below, which compares the CSS 2.1 box with the old Internet Explorer 5.5 "quirks mode" box model.

In IE5.5, both padding and border values were, in effect, subtracted from `width`, decreasing the size of the content area. Though this was the exact opposite of the behavior defined in early CSS specifications, many web developers thought it was the more sensible approach.

As a way to resolve these competing models, the CSS Working Group introduced the `box-sizing` property. It lets us *choose* how the browser should calculate box dimensions.

Managing Box Dimensions with `box-sizing`

The `box-sizing` property is defined in the [CSS Basic User Interface Module Level 3]() specification. It has two possible values: `content-box` and `border-box`.

Initially, the value of `box-sizing` is `content-box`. With this value, setting the `width` and `height` (or `inline-size` and `block-size`) properties of an element affects the size of its content area. This matches the behavior defined by the CSS 2.1 specification. It's also the default behavior in browsers (as illustrated in the image above).

Setting the value of `box-sizing` to `border-box` creates a little bit of magic. Now the values of `width` and `height` are applied to the outer border edge instead of the content area. Borders and padding are drawn inside the element box. Let's look at an example that mixes percentage widths and `px` units for padding and borders:

```
<div class="wrapper">
    <article>
        <h2>This is a headline</h2>
        <p>Lorem ipsum dolor sit amet, consectetur
adipisicing ... </p>
    </article>
    <aside>
        <h2>This is a secondary headline</h2>
        <p>Lorem ipsum dolor sit amet, consectetur
adipisicing ... </p>
    </aside>
</div>
```

Both our `<article>` and `<aside>` elements have the following CSS applied. Our first element has a width of 60%, while the second has a width of 40%:

```
article, aside {
    background: #FFEB3B;
    border: 10px solid #9C27B0;
    float: left;
    padding: 10px;
}
article {
    width: 60%;
}
aside {
    width: 40%;
}
```

The image below shows how this code renders in the browser.

By default, both `<aside>` and `<article>` have a `box-sizing` value of `content-box`. The `border-width` and `padding` values add 40 pixels to the width of each element, which throws off the 60%/40% split. Now let’s add `box-sizing: border-box` to the `<article>` and `<aside>` elements:

```
article, aside {
    box-sizing: border-box;
}
```

You can see the change below.

The elements have the same width, but the `box-sizing: border-box` means that the width includes the border and padding. Because the `width` property applies to the border edge instead of the content area, our elements now fit side by side.

I recommend using `box-sizing: border-box` in your projects. It makes life easier, as there's no need to calculate the `width` value to account for the values of `padding` and `border`. Boxes behave more predictably.

The best way to apply `box-sizing: border-box` is with reset rules. The following example is from Chris Coyier's CSS-Tricks post, "Inheriting box-sizing Probably Slightly Better Best-Practice":

```
html {
    box-sizing: border-box;
}
*, *:before, *:after {
    box-sizing: inherit;
}
```

This applies `border-box` sizing to every element by default, without affecting the box-sizing behavior of existing parts of your project. If you *know* that there'll be no third-party or legacy components that rely on `content-box` behavior, you can simplify these rules:

```
*, *:before, *:after {
    box-sizing: border-box;
}
```

In some cases, you may not want an element to generate a box at all, but still keep its contents visible to the user and retain its semantics. That's when you'll want to use `display: contents`.

Preventing Box Generation with `display: contents`

Using the `contents` value for the `display` property prevents the browser from generating an element box, without removing its semantics. Applying `display: contents` to an unordered list, for example, removes its default margin and padding.

In visual terms, it's as if the `<ul>` element isn't there. This is particularly useful when working with Grid and Flexbox. In grid and flexible box layout, the direct children of the grid or flex container participate in the formatting context. Adding `display: contents` to the child of a grid or flex container means that the container's "grandchild" elements participate in that formatting context instead. We'll come back to this point later in the chapter.

Use `display: contents` with Caution

`display: contents` is not *supposed* to affect the semantics of an element. Unfortunately, there's a severe accessibility bug in Safari's implementation (versions 15 and older as of this writing). In Safari, `display: contents` strips semantic meaning from elements, which prevents them from being exposed to the accessibility tree. Landmark elements such as `<h1>` and interactive elements such as `<button>` become imperceptible to screen readers. It makes document navigation impossible. Adding ARIA attributes doesn't fix it.

For this reason, *do not* use `display: contents` to reset margins and padding for every element until this bug is fixed in Safari. For a detailed overview of the accessibility issues `display: contents` can cause, see Adrian Roselli's "Display: Contents Is Not a CSS Reset".

Chrome and Edge fixed a similar bug in their implementations of `display: contents` as of version 89 (released March 2021) and later. Firefox resolved its version of this bug with Firefox 62 (released in 2018).

Floating Elements and Normal Flow

When we float an item by setting the value of the `float`, we remove it from the normal flow. Instead of stacking in the block direction, the box is shifted to either end of the current line until its edge aligns with the containing block or another floated element. `float` has four possible values:

- `left`, which shifts the box to the left of the current line
- `right`, which shifts the box to the right of the current line
- `inline-start`, which shifts the box to the start of the current line

- `inline-end`, which shifts the box to the end of the current line

Both `inline-start` and `inline-end` depend on the language direction of the document. For languages that are written and read horizontally, from left to right, `inline-start` aligns the box to the left, while `inline-end` shifts it to the right. For languages that are written and read from right to left, such as Arabic, `inline-start` shifts the box to the right and `inline-end` shifts it to the left. When the language is written vertically, `inline-start` is the top of the container, and `inline-end` is the bottom.

Content flows along the far edge of a floated box if there's enough horizontal space. If, for example, we left-float an image that's 300 pixels wide by 225 pixels high, the adjacent lines of text fill in along its right edge. If the computed height of the content exceeds 225 pixels, the text wraps around the bottom of the image.

If, however, the computed height of the remaining content is shorter than 200 pixels, the floated element overflows its container.

Text in sibling elements will also flow along the edge of a float if there’s enough room. The length of each line—its line box—will be shortened to accommodate the float.

Floating a series of elements works a little bit differently. Let’s apply `float: left` to a series of `<div>` elements that are 500 pixels wide inside a container that’s 1500 pixels wide. As you can see in the image below, these elements stack horizontally to fill the available space. Elements that don’t fit in the available horizontal space will be pushed down until the box fits or there are no more floated elements.

Floated elements don’t wrap neatly. Yet before Flexbox and Grid layout, developers used floats to create gridded layouts. This requires setting an explicit `height` value to elements within a floated grid to ensure that elements don’t “snag” on previously floated elements. The drawback, of course, is that you have to adjust the height of every element should the content require it, or edit your text and images to ensure that they don’t overflow the container, as is happening in the image below.

Removing elements from the normal flow can be tricky. Other content in the document will want to cozy up next to a floated element, but that may not be the layout we're trying to achieve.

Consider the UI pattern known as a *media object*. A **media object** consists of an image or video thumbnail that's aligned to the left or right of its container, accompanied by some related text. You've probably seen this style of component in comments sections, on news sites, or as part of YouTube's "Up Next" feature.

Here's what the markup for a media object might look like:

```
<div class="media__object">
    <img src="video_thumbnail.jpg">
    <div class="media__object__text">
        Lorem ipsum dolor sit amet, consectetur adipiscing
elit, sed do eiusmod tempor incididunt utlabore et dolore
magna aliqua.
    </div>
</div>
```

The media object is a simple pattern: it consists of an image placed to the left or right of its container and some related text. Floating an image is one way to create a media object layout:

```
.media__object {
    background: #ccc;
    padding: 1rem;
}
.media__object img {
    float: left;
    margin-right: 1rem;
}
```

The drawback of simply floating an image is that the height of the accompanying text may be shorter than the height of the floated image. When that happens, our container will collapse to match the height of the text. Content in adjacent elements will flow around the floated element, as illustrated in the image below.

To prevent this, we need to **clear** our float. That is, we need to force the container to wrap around its floated contents. Let’s look at some methods for doing so in the next section.

Clearing Floats

The simplest way to clear a float is to establish a new block formatting context for its container. A block formatting context always contains its children, even when those children are floated.

So how do we create a new block formatting context? We have a couple of options.

Using `display: flow-root`

One way is to use the `display: flow-root` property. Let’s add `display: flow-root` to our `.media__object` rule set:

```
.media__object {
    background: #ccc;
    padding: 1rem;
    display: flow-root;
}
.media__object img {
    float: left;
    margin-right: 1rem;
}
```

Now our floated image is completely contained by its parent, as shown below.

The `flow-root` value for the `display` property is a fairly recent addition to CSS. Its sole purpose is to trigger a block formatting context for a containing element. It's well supported in modern browsers, but it's not the only option for creating a new block formatting context.

Using the `contain` Property

The `contain` property is a performance-related property. Its purpose is to constrain reflow and repaint operations to an element and its children, instead of the entire document. However, it also creates a new block formatting context for that element. That means we can also use it to clear floats.

Let's update our CSS. We'll float the image to the right this time. The image below shows what our media objects look like before clearing the float.

Now let’s add `contain: content` to the `.media__object` rule set:

```
.media__object {
    background: #ccc;
    padding: 1rem;
    contain: content;
}
.media__object--right img {
  float: right;
    margin-left: 1rem;
}
```

Now our `.media__object` elements encompass their floated image children.

Unfortunately, Safari versions 15 and below don't support the `contain` property. Since `display: flow-root` has broader compatibility, use that instead.

For older browsers—versions of Firefox prior to 52, Chrome prior to 57, Safari prior to 13, Edge prior to 79, and every version of Internet Explorer—use the "clearfix" method.

Clearfix

Clearfix uses the `::after` pseudo-element and the `content` property to insert a box at the end of a containing element. Since pseudo-elements can be styled like actual elements, we can apply `display: table` (or `display: block`) and `clear: both` to clear our float:

```
.media__object::after {
    content: "";
    display: table;
    clear: both;
}
```

Floats are best used for aligning images and tables to the left or right, or for placing content asides within a page. For most other uses, Flexbox and Grid are better choices. Flexbox and Grid typically require less markup and less CSS, while offering more flexibility in the kinds of layouts we can create. Grid and Flexbox also make vertical centering remarkably easy, as we'll see later in this chapter.

But first, let's discuss another way to remove elements from the normal flow: the `position` and `z-index` properties.

Positioning and Stacking Elements

Every element in a document participates in a *stacking context*. The **stacking context** is a model or set of rules for how elements are painted to the screen. If you've ever used the `z-index` property, you've worked with stacking contexts.

The root `<html>` element creates a *root stacking context*. Some CSS properties and values can also trigger a stacking context for the elements they're applied to. Whether part of a root or local context, children within a stacking context are painted to the screen from back to front as follows:

- child stacking contexts with a negative stack level (for example, positioned and with `z-index: -1`)
- non-positioned elements whose computed `position` value is `static`
- child stacking contexts with a stack level of 0 (for example, positioned and with `z-index: auto`)
- child stacking contexts with positive stack levels (for example, positioned and with `z-index: 1`)

If two elements have the same stack level, they'll be layered according to their order in the source HTML.

Let's look at an example. Here's our HTML:

```
<div id="a">
    <p><b>div#a</b></p>
</div>
<div id="b">
    <p><b>div#b</b></p>
</div>
<div id="c">
    <p><b>div#c</b></p>
</div>
<div id="d">
    <p><b>div#d</b></p>
</div>
    <div id="e">
    <p><b>div#e</b></p>
</div>
```

And here’s our CSS:

```
#a {
    background: rgba( 233, 30, 99, 0.5 );
}
#b, #c, #d, #e {
    position: absolute;
}
#b {
    background: rgba( 103, 58, 183, 0.8 );
    bottom: 120px;
    width: 410px;
    z-index: 2;
}
#c {
    background: rgba( 255, 235, 59, 0.8 );
    top: 190px;
    z-index: 1;
}
#d {
    background: #03a9f4;
    height: 500px;
    top: 10px;
    z-index: -1;
}
#e {
    background: rgba( 255, 87, 34, 0.7 );
    top: 110px;
    z-index: 1;
}
```

This produces the stacking order shown in the image below.

The bottommost layer is `#d`, because its `z-index` value is -1. Since `#a` isn't positioned, it sits above `#d`, but below the positioned elements (`#b`, `#c`, and `#e`). The next layer is `#c`, followed by `#e`. Since both elements have the same `z-index` value, `#e` is stacked higher, because it's last in the source order. The topmost layer is `#b`, due to its `z-index` of `2`.

To make it a bit easier to visualize, the following image shows a three-dimensional projection of the above stacking context.

All of the elements in the previous example are part of the root stacking context. But let's see how stacking is affected by the `opacity` property, which forces a local context when its value is less than `1`. Consider the following HTML:

```
<div id="f">
    <p><b>div#f</b></p>
</div>
<div id="g">
    <p><b>div#g</b></p>
</div>
```

It's paired with this CSS:

```
#f, #g {
    position: absolute;
}
#f {
    background: rgba( 255, 193, 7, .9 );
}
#f p {
    background: rgb( 34, 34, 34 );
    color: whitesmoke;
    position: relative;
    z-index: 1;
}
#g {
    background: rgba( 3, 169, 244, .7 );
    top: 50px;
    left: 100px;
}
```

According to the rules of the stacking context, `#f p` occupies the topmost layer in the stack. That's what we see in the following image.

But if we change our CSS and add `opacity: .99` to the `#f` rule set, something interesting happens:

```
#f {
    background: rgba( 255, 193,7, .9 );
    opacity: .99;
}
```

The `opacity` property creates a new stacking context any time its value is less than `1`. As a result, the `z-index` for its child element becomes relative to its parent rather than the root stacking context.

Let’s add an absolutely positioned `<div>` element to `#f` and give it a `z-index` value of `2`. Now `<div>` is stacked on top of `#f p` (see below), but it’s still layered behind `#g` because `#f` has a local stacking context. Children of a local stacking context can only be reordered relative to that context. Elements that sit in other contexts can’t be layered within a local one.

A Workaround for Opacity Transition Issues

Because `opacity` triggers a new stacking context, you may run into undesired behavior when transitioning the `opacity` of layers that overlap. To work around this, use `rgba()` or `hsla()` values for `color` or `background-color` and transition those instead.

Let's look at an example of using the stacking context to manage layers and positioned elements. In this case, we'll create a menu that slides in from the top of the screen. But rather than slide in *over* the logo and menu button, we'll make it slide in beneath it. First, our HTML:

```
<header>
    <img src="dont-awesomenews.svg">
    <button type="button" id="menu">
        <img src="dont-menu.svg">
    </button>
    <nav>
        <ul id="menu-list">
            <li><a href="/sports">Sports</a></li>
            <li><a href="/politics">Politics</a></li>
            <li><a href="/arts">Arts &amp; Entertainment</a>
</li>
            <li><a href="/business">Business</a></li>
            <li><a href="/travel">Travel</a></li>
        </ul>
    </nav>
</header>
```

Clicking the `<button>` element causes the element to slide into view. Here's our our (simplified) CSS:

```
header {
    background: hsl( 206, 9%, 15% );
    color: whitesmoke;
    width: 100%;
}
nav {
    background: hsla( 206, 9%, 15%, .9 );
    position: absolute;
    width: 100%;
    left: 0;
    top: -33vw;
    transition: top 500ms;
}
.open {
    top: 6rem;
}
```

The CSS above creates a menu that slides down from the top when triggered. But as it slides in, it passes over the AwesomeNews logo, as pictured below.

Our menu (the `<nav>` element) slides over the logo and menu button because it has a higher stack level. Remember that when multiple elements have the same `z-index` value, the last one in the source will be the topmost layer.

Let's change this. What happens when we add `z-index: -1` to the `nav` rule set? Well, you get the mess you see pictured below.

The navigation slides in behind the logo and menu button, but it also slides in behind the content. It's hard to read and impossible to click.

Because its parent element (`<header>`) isn't positioned and has a computed `z-index` value of `auto`, the `<nav>` element is still part of the root stacking context. Adding `z-index: -1` shoves it to the bottom of the root element's stack, which means it sits behind other elements in the root stacking context.

So how do we fix this? By creating a new stacking context on our `<header>` element, which is the parent element of `<nav>`. We already know that the `opacity` property can create a new stacking context when its value is less than `1`. However, positioned elements can also create a new stacking context.

Positioning and `z-index`

If using `position: absolute` or `position: relative`, you'll also need to set `z-index` to a value other than `auto` in order to create a new stacking context.

Let's add `position: fixed` to our `<header>` element. Now our `<nav>` element participates in the stacking context of `header` instead of the document root:

```
header {
    background: #222629;
    color: whitesmoke;
    width: 100%;
    top: 0;
    position: fixed;
}
```

Since `<nav>` now participates in the stacking context of `<header>`, the menu sits above the rest of our content. But because `<nav>` has a negative stack level, it sits at the bottom of the `<header>` element's stacking context, as illustrated below.

For the rest of this chapter, we'll switch gears and talk about some newer modules for creating more complex layouts. We'll learn how to wrap text around complex shapes, flow content across multiple columns, create more flexible components, and build complex grid layouts—all of which was previously difficult if not impossible, and often required extra markup or JavaScript.

Outside-the-box Layouts with CSS Shapes

Everything is a box in CSS, but with CSS Shapes we can sometimes have circles and triangles. The CSS Shapes specification makes it possible to flow content around non-rectangular shapes.

Shaped elements *must* be floated. Remember: when you float an element, content will flow along its right or left edge. When an element is **shaped**, content instead flows along the *edges of the shape*. Shaped elements must also have a width and height. Width and height can either be explicitly set, derived from the element's `padding` value, or derived from the size of its contents.

Let's look at a simple example of flowing text around a circle. Here's our (abbreviated) HTML:

```
<p>
Etiam pharetra nibh tempus, viverra sem vel, eleifend eros.
Integer semper lorem odio, ac feugiat mi tincidunt et. Donec
luctus ante sit amet elementum facilisis. Suspendisse
potenti. In varius eros at mollis eleifend.…
</p>
<p>
Phasellus blandit elit vitae euismod volutpat. Maecenas
venenatis sed leo euismod aliquam.
</p>
```

We'll pair it with the following CSS:

```
p:first-of-type::before {
    content: ' ';
    float: left;
    width: 40rem;
    height: 40rem;
    shape-outside: circle( 50% );
}
```

In this example, our rule set floats the element and explicitly sets its width and height. We've defined our actual shape using the `shape-outside` property. `shape-outside` defines the type of shape, along with its size and position. The above CSS results in the layout shown below.

Using `shape-outside` generates a *reference box*. This reference box functions as a sort of coordinate system in which the shape is drawn. Even though content *flows around the shape*, the reference box still exists. You can see it by adding a background color to `p:first-of-type::before`.

Shapes can be defined in one of two ways:

- using the `ellipse()`, `circle()`, `inset()`, `path()`, or `polygon()` basic shape functions
- using an image with an alpha channel—that is, an image with full or partial transparency, including gradients

Let’s look at shape functions next.

Using Shape Functions

Shapes are a type of CSS value, like length, angle, or color units. Here, we’re going to discuss them in the context of `shape-outside`, but they also apply to the `clip-path` and `offset-path` properties. Some of the examples in this section will use `clip-path()` to illustrate the contours of the shape.

This section uses *value definition syntax* to explain the arguments for each of these functions. I’ve lifted each function definition directly from the CSS Shapes specification. Value definition syntax is a bit awkward to read, but it’s far clearer than saying “it accepts from one to six arguments, the first four of which use the same order and shorthand as `margin`, and the last two of which are really one argument that sets a border radius.” That is, by the way, an explanation of the `inset()` function.

Value definition syntax is easy to read once you get the hang of it. If you’re familiar with regular expressions, some of it will be familiar.

The image above illustrates a value definition statement using a generic example. Mozilla Developer Network has a far more thorough explanation if you want more details.

`ellipse()` and `circle()`

The `ellipse()` function creates an ellipse or oval around two focal points within the reference box. Its arguments use the following pattern:

```
ellipse( [ <shape-radius>{2} ]? [ at <position> ]? )
```

Both sets of arguments for `ellipse()` are optional, and the center point of the reference box (50%, 50%) is the default value for both. Yes, `shape-outside: ellipse()` is perfectly valid. It creates a circle.

Remember Geometry?

Ellipses are easy to recognize but hard to explain. If you're like me and didn't pay enough attention during your geometry lessons, Math Is Fun has an easy-to-understand primer. Also recall that circles are a special kind of ellipse.

The first set of arguments indicates the x-radius and y-radius positions around which the ellipse will be drawn. If you include one, you *must* include the other. Here's an example:

```
.ellipse {
    float: left;
    width: 40rem;
    height: 40rem;
    shape-outside: ellipse( 20% 40% );

    /* Added to illustrate the ellipse and how text flows
around it */
    background-color: #666;
    clip-path: ellipse( 20% 40% );
}
```

Adding x- and y-radius arguments to the `ellipse` function causes an oval-shaped text flow:

The second argument *must* begin with the `at` keyword, and it indicates the position of the center point for the ellipse. Let’s set a position for our ellipse:

```
.ellipse {
    float: left;
    width: 40rem;
    height: 40rem;
    shape-outside: ellipse( 20% 40% at 0% 50% );

    /* Added to illustrate the ellipse and how text flows
around it */
    background-color: #666;
    clip-path: ellipse( 20% 40% at 0% 50% );
}
```

Adding a position value to the ellipse function shifts where the center point of the ellipse sits within the shape’s reference box. Now the center point for our ellipse sits completely to the left of the reference box, and halfway down.

Position values can be lengths or percentages. Or you can use position keywords like `left`, `bottom`, `right`,`top`, and `center`.

The `circle()` function is a simpler version of `ellipse()`. Rather than requiring an x-radius and y-radius, `circle()` instead accepts a single shape radius argument. Using `circle(30%)` is the same as `ellipse(30% 30%)`.

`inset()`

We can create rectangular shapes with the `inset()` function. I suspect you're wondering why we need a rectangular shape when floats are already rectangular. Well, floats don't let you create neat inset effects, nor do they cause text to flow around a rectangle's rounded corners. With `inset()` we can do both. Let's update our example from earlier in the chapter. We'll change our shape from `circle()` to `inset()`:

```
p:first-of-type::before {
    content: ' ';
    float: left;
    width: 40rem;
    height: 40rem;
    shape-outside: inset( 10% 20% 20% 0 round 8rem );
    background:  hsl( 271, 76%, 83% );
}
```

Using `inset()` with a border radius causes text to flow around the borders' curves, as illustrated below.

Arguments for `inset()` use the following syntax:

```
inset( <length-percentage>{1,4} [ round <'border-radius'> ]?
)
```

The `inset()` function requires at least one length or percentage value, but accepts as many as four, representing the top, right, bottom, and left offsets from the reference box. These arguments mirror the syntax of the `margin` shorthand property. Using two values, for example, sets the top/bottom and right/left offsets. Using three values sets the top/bottom and right offsets.

The `inset()` function also accepts an *optional* `border-radius` argument, indicated by the `round` keyword, and followed by a valid border radius value.

polygon()

We can use the `polygon()` function for more complex shapes. A **polygon** is a closed shape made from straight lines—which includes triangles, rhomboids, stars, and octagons. The `polygon()` function takes arguments in the following form:

```
polygon( <'fill-rule'>? , [<length-percentage> <length-
percentage>]# )
```

Its `fill-rule` argument is optional. It should be `nonzero` or `evenodd`, but defaults to `nonzero` if omitted. The second argument should be a series of length or percentage values that describe the shape. Let’s create a triangle, this time floated to the right:

```
p:first-of-type::before {
    content: ' ';
    float: right;
    width: 60rem;
    height: 60rem;
    shape-outside: polygon( 100% 0, 0% 50%, 100% 100% );

    /* Helps visualize the shape */
    clip-path: polygon( 100% 0, 0% 50%, 100% 100% );
    background-color: hsl( 182, 25%, 50%, .25 );
}
```

The image below shows the result—a triangle-shaped float created using the `polygon()` function, and floated to the right.

Notice here that some languages and writing directions may cause text to flow less tightly around a floated shape. The language in this example (Latin) uses a left-to-right language direction, and spaces to mark the beginning and ending of words. As a result, our lines differ in length, and our shape is less clearly articulated. Use `word-break: break-all` or `text-align: justify` to mitigate this.

If you want to use complex, multi-sided polygons, or shapes that include curves and arcs, these basic shapes can be quite limiting. Luckily for us, we aren't limited to using shapes with `shape-outside`.

Using Images

We can also use images to define shapes for use with `shape-outside`. You can use any image with an alpha channel. This includes PNG images, SVG images with HSLA or RGBA `fill` values, and CSS gradients. It doesn't, however, include image formats such as GIF, which only support binary transparency. Such formats are incompatible with `shape-outside`.

The image below illustrates the use of `shape-outside` with a PNG image that contains an alpha channel.

Mug of coffee photo by Foodie Factor from Pexels.

Let’s look at the CSS used to create the layout for *A Good Cuppa Joe* (pictured above). In this example, we’ve added the `shape-margin` property, which adds space around the shape. Unlike `margin`, `shape-margin` accepts a single length or percentage value that’s applied around the contours of the shape:

```
[src='coffee-cup.png'] {
    float: right;
    width: 615px;
    height: 569px;
    shape-outside: url( 'coffee-cup.png' );
    shape-margin: 2rem;
}
```

The CSS above is paired with the markup shown below. It’s been abbreviated for the sake of space:

```
<p>
  <img src="coffee-cup.png" alt="a cup of coffee on a saucer
with a spoon">
  Etiam pharetra nibh tempus, viverra sem vel, eleifend eros.
Integer semper lorem odio, ac feugiat mi tincidunt et. Donec
luctus ante sit amet elementum facilisis. Suspendisse
potenti. In varius eros at mollis eleifend. …
</p>
<p>
  Phasellus blandit elit vitae euismod volutpat.…
</p>
```

The trick to getting text to flow along the curved edge of `coffee-cup.png` is to apply the `shape-outside` property to the image element, and set the image’s URL as its value. But it’s not the image itself that causes text to flow along the image curve. It’s the combination of `float` and `shape-outside`. Remove the `<img>` tag and change the selector to `p:first-of-type::before`, and there will be an empty, rounded space where the cup used to be. The image below shows how the combination of `float` and `shape-outside` causes text to flow around the curve of the image, whether the image is included in the markup or not.

When using external images with `shape-outside`, the browser makes a *potentially CORS-enabled fetch* for the resource. The browser *must* be able to resolve the document's path in such a way that it can generate an origin. In practical terms, this means two things:

- Linked and external images need to share the same origin as your document, or be served with the correct `Access-Control-*` response headers.
- You'll need to use a server or content delivery network when developing or viewing layouts that use `shape-outside` so that the browser can create an origin.

You'll need to use a web server even during local development. Loading files directly from your computer's file system won't work.

Cross-origin Resource Sharing

Refer to the "Fonts and Origins" section of Chapter 6, "Working with Text", for an explanation of cross-origin resource sharing.

CSS Gradients Are Images Too!

Image files and data URIs are not the only images we can use with `shape-outside`. CSS gradients also work. The image below shows an example of `shape-outside` combined with `background-image` with a CSS gradient to create a beach-like page design.

To create it, we’ve used the following CSS:

```
body {
    background-color: hsl( 183, 80%, 80% );
}
body::before {
    content: ' ';
    display: block;
    float: right;
    height: 100vh;
    width: 50%;
    background-size: cover;

    background-image:  linear-gradient( -45deg, hsl( 240,
86%, 25% ) 0%, hsla( 213, 100%, 50%, 0.8 ) 22%, hsla( 183,
100%, 50%, 0 ) 53% );

    shape-outside:     linear-gradient( -45deg, hsl( 240,
86%, 25% ) 0%, hsla( 213, 100%, 50%, 0.8 ) 22%, hsla( 183,
100%, 50%, 0 ) 53% );
}
```

In this case, the text flows along the edge of the gradient at the point where it becomes fully transparent (`hsla( 183, 100%, 50%, 0 )`).

We can adjust the point at which the text flows by adjusting the value of the `shape-image-threshold` property. `shape-image-threshold` represents an alpha value between 0 and 1, inclusive. Portions of the image with an alpha level that exceeds this threshold will be included in the shape. Adding `shape-image-threshold: 0.4` to the preceding CSS changes the size of the triangle created by the linear gradient.

Changing the alpha threshold means that text now flows around areas of the gradient that have a transparency value of 0.4 or higher. These are interpolated values between `hsla(213, 100%, 50%, 0.8)` (*alpha = 0.8*) and `hsla(183, 100%, 50%, 0)` (*alpha = 0*).

The Shape of the Future (or the Future of Shapes)

As you may have figured out from the `shape-outside` property name, CSS Shapes is concerned with flowing text around floated items. Content *inside* the shaped element does not, in fact, follow the contours of the inside of the shape. Instead, that content is contained by the dimensions of the reference box.

Level 2 of the CSS Shapes Module specification defines `shape-inside` and `shape-padding` properties that do affect the shape of content *inside* the box. The image below shows what a layout that uses `shape-inside` might look like.

Some day, we may be able to create layouts like the one pictured above, in which the text inside the shape follows its contours. It's not clear how soon these features will be implemented, however. That specification is still in its early stages.

Using CSS Multicolumn Layout

Multicolumn layout allows text and elements to flow from one column to another automatically. With it, we can create text layouts that mimic those found in newspapers, magazines and ebooks. We can also use it to create space-efficient user interfaces.

Defining Column Number and Width Using `columns`

To create multiple columns, set the `columns` property:

```
<div style="columns: 2">
    <p>Lorem ipsum dolor sit amet, consectetur adipisicing
... </p>
    <p>Duis aute irure dolor in reprehenderit in voluptate
... </p>
</div>
```

The `columns` property is a shorthand property for `column-width` and `column-count`. With `columns`, the first value that can be interpreted as a length becomes the value of `column-width`. The first value that can be interpreted as an integer becomes the value of `column-count`. Order doesn't matter. A declaration such as `columns: 10em 3` is the shorthand way of typing `column-width: 10em; column-count: 3`. It's also equivalent to typing `columns: 3 10em`.

If a value is unspecified, its initial value is `auto`. In other words, `columns: 4` has the same effect as typing `columns: 4 auto` or `column-width: auto; column-count: 4`.

Setting `column-width` determines the *optimal size* for each column. Its value should be in length units—such as `column-width: 200px` or `column-width: 10em`. Percentages won't work.

“Optimal”, of course, means that `column-width` sets the *ideal* width. The actual width of a column may be wider or narrower than the value of `column-width`. It depends on the available space and/or viewport size.

In the example pictured below, for example, the container is `800px` wide and the `column-width` value is `15em`. That gives us three columns.

But if we expand the width of the container to 1920 pixels, there’s now room for six columns:

Shrinking the width of the container to 450 pixels, on the other hand, reduces this layout to a single column.

Setting the `column-count` property defines the *optimal number* of columns to create. Its value must be an integer greater than `0`.

When `column-width` is something other than `auto`, the browser creates columns of that width up to the number of columns specified by `column-count`. If `column-width` is `auto`, the browser will create the number of columns specified by `column-count`. That may be more easily shown than explained, so let's illustrate it.

In the images that follow, our container has a `column-count` value of `3`, and a `column-width` value of `auto`. Whether our container is 800 pixels wide (as in the first image) or 450 pixels wide (as in the second image), we still have three columns.

Now compare these to the following two images. In both cases, our container has a `column-count` value of `3`, and a `column-width` value of `10em`. When the container is 800 pixels wide, the number of columns remains the same, as shown below.

But when our container is 400 pixels wide, we can only fit two columns, as shown below.

This goes out of the window entirely if we set the `height` of a column container. Setting a fixed height on a container forces the browser to create additional columns to accommodate the container's content. In this case, the `column-count` property is ignored.

Spacing Columns with `column-gap` and `column-rule`

The number of columns that can fit in a container also depends on the value of `column-gap`. Known as the **gutter** in print design, the **column gap** sets the distance between each column. The initial value of `column-gap` is `normal`. In most browsers, that's `1em` or `16px`.

Increasing or decreasing the width of the `column-gap` can affect both the width of each column and the space between. For example, if a container is `45em` wide with `column-width: 15em` and `column-gap: normal` applied, its contents will be divided into two columns rather than three, as shown below.

Changing `column-gap` to `0`, however, gives us our full three-column layout, as shown below.

Without a `column-gap`, there's now sufficient space for three columns.

As with `column-width`, the value of `column-gap` should be either `0` or a positive length value. Negative lengths such as `-2em` are invalid.

Although originally defined as part of the multicolumn specification, newer browsers (Firefox 63 and later, Chrome and Edge 84 and later, and Safari 14.2 and later) also support the use of `column-gap` with Grid and Flexbox. Similarly, you can use the `gap` shorthand property—which was first defined by the CSS Grid specification—with multicolumn and Flexbox layouts.

With `column-rule`, we can add lines to visually separate columns. It functions similarly to `border`, and accepts the same values. For example:

```
.multi-col {
    column-rule: 5px hsl( 190, 73%, 50% ) dashed;
}
```

Like `border`, `column-rule` is a shorthand for the `column-rule-width`, `column-rule-style`, and `column-rule-color` properties. Each `column-rule-*` property accepts the same values as its `border` counterpart. An example of using `column-rule` is shown below.

Column width isn’t affected by changes to `column-rule`. Instead, column rules sit at the midpoint of the column gap. If the width of the rule exceeds that of the gap, the column rule renders beneath the columns’ contents.

Images within Columns

In cases when an image is wider than its column, the text gets rendered on top of the image. This is the expected behavior that's defined by the specification.

The image below shows an image within a column sitting at the bottom of the stacking context in Firefox.

Adding a `width: 100%` declaration to the image or object constrains the image width to that of the column box, as shown below. It also constrains the height based on the aspect ratio of the image. You can add `height: auto` to ensure this behavior.

Floated elements within a multicolumn layout are floated within the column box. In the image below, the `<img>` element has a `float: inline-start` rule applied. Text still flows around the image, but within the constraints of the column.

Multicolumn layout doesn’t automatically create a new stacking context. Positioned elements in a multicolumn container are positioned relative to the root stacking context unless you create a local stacking context on the container.

Making Elements Span Columns

We can also make a particular element span columns with the `column-span` property. This property accepts two values: `none` and `all`. Using `none` means that the element will be part of the normal column flow, while using `all` will make the element span every column.

It’s not currently possible to make an element span *a particular number* of columns. We’re limited to specifying whether it should span all columns or none at all. Consider the layout shown below, which shows how an `<h1>` element fits into the multicolumn layout flow.

Corgi photo by Alvan Nee from Unsplash.

Here, the `<h1>` element (the article headline “Dog bites man …”) is part of the multicolumn layout flow. It sits within a column box, wrapping as appropriate. Now let’s add `column-span: all`:

```
article > h1 {
    column-span: all;
}
```

This gives us the layout shown below, with a headline that spans all of our columns.

Managing Column Breaks within Elements

In a multicolumn layout, a long block of text may start in one column and end in another, as illustrated below.

To prevent this, use `break-inside: avoid` or `break-inside: avoid-column`. The `break-inside` property applies to the children of a multicolumn container. For example, to prevent all children of `.multi-col` from breaking across column boxes, use the following:

```
.multi-col > * {
    break-inside: avoid-column;
}
```

Now the purple paragraph no longer breaks across columns, as can be seen below.

The `break-inside` property also affects paged media, which explains why there are both `avoid` and `avoid-column` values. The difference? The `avoid-column` property only prevents a box from breaking across columns, while `avoid` prevents a box from breaking across both columns *and pages*.

CSS Fragmentation Module Level 3

The CSS Fragmentation Module Level 3 specification is closely related to the multicolumn and paged media specifications. It unifies the `column-break-` *and* *`page-break-`* properties from earlier specifications, and further defines how block boxes should break across columns and pages.

It's also possible to force a break before or after an element using `break-before` and `break-after`. Let's force a column break before the third paragraph:

```
.multi-col p:nth-of-type( 3 )  {
    background-color: #c09a;
    break-before: column;
}
```

Here, we've used the `column` value to force a column break before the selected element, as illustrated below.

The `break-after` property works similarly, forcing a column break *after* the selected element. Using `break-before: always` also forces column breaks. However, the `always` value also applies to paged media, where `column` only applies to multicolumn layout.

Basic browser support for multicolumn layout is quite good. Support for `break-before`, `break-after`, and `break-inside`, however, is a different story.

Safari (versions 14.2 and later), Chrome, and Edge support the `column` and `avoid-column` values for `break-before` and `break-after`, but lack support for the `always` value. Instead, you can use the `-webkit-column-break-before` and `-webkit-column-break-after` properties with `always`. These prefixed properties are remnants of an earlier version of the multicolumn specification, but they function in much the same way as `break-before` and `break-after`. If you use the `-webkit-column-*` properties, future-proof your CSS by including the standardized properties.

Firefox, on the other hand, only supports the `always` value of `break-before` and `break-after`, and only for *printed* documents. Firefox also lacks support for the `avoid-column` value of `break-inside`. As a workaround, you can use the `avoid` value, since `break-inside: avoid` works for columns and pages in every browser.

Optimizing the User Interface

Arranging paragraphs of text isn't the only use case for multicolumn layouts. We can also use them with lists to optimize the use of horizontal space. Consider the layout shown below, representing a list split into three columns.

Your first thought might be to split this list into three separate lists and use floats, Flexbox or Grid to place them side by side. Here’s what that markup might look like:

```
<div class="grid-3-across">
    <ul>
        <li>Apples</li>
        <li>Oranges</li>
        <li>Bananas</li>
        <li>Dragon fruit</li>
    </ul>
    <ul>
        <li>Cherries</li>
        <li>Strawberries</li>
        <li>Blueberries</li>
        <li>Raspberries</li>
    </ul>
    <ul>
        <li>Durian</li>
        <li>Mangosteen</li>
        <li>Mangoes</li>
    </ul>
</div>
```

And the accompanying CSS:

```
.grid-3-across {
    display: grid;
    grid-template-columns: repeat(3, 1fr);
}
```

While this approach works, it requires slightly more markup than a single list element. We're using three `<ul>` elements instead of one. With a multicolumn layout, we can use a single element:

```
<ul style="columns: 3">
    <li>Apples</li>
    <li>Oranges</li>
    <li>Bananas</li>
    <li>Dragon fruit</li>
    <li>Cherries</li>
    <li>Strawberries</li>
    <li>Blueberries</li>
    <li>Raspberries</li>
    <li>Durian</li>
    <li>Mangosteen</li>
    <li>Mangoes</li>
</ul>
```

What's more, multicolumn layout automatically balances the number of items in each column. If our list grows to 16 or 18 items, we don't have to edit the markup to rebalance the list.

Missing Bullets

Blink- and WebKit-based browsers remove bullets and numbers from some or all list items in a multicolumn layout. To resolve this, add `margin-inline-start: 20px` or `margin-left: 20px` to `<li>` elements in a multicolumn container.

Another use case for multicolumn layouts is wrangling lists of checkbox inputs. Here, too, we can maximize the use of horizontal space to create more compact forms, as pictured below.

Use multicolumn layout when you have blocks of content to be automatically distributed and evenly spaced across several columns. It isn't well suited to creating page layouts—such as adding a navigation column and a main content column. For page layouts, CSS Grid is a better choice. We'll discuss Grid later in the chapter.

Creating Flexible Layouts with Flexbox

The CSS Flexible Box Layout Module, better known as Flexbox, was designed to distribute elements and space in one direction—a row (`flex-direction: row` or `row-reverse`) *or* a column (`flex-direction: column` or `column-reverse`).

A basic flexible box layout is simple to create: just add `display: flex` or `display: inline-flex` to the containing element. These values for `display` trigger a *flex formatting context* for that containing element's children. Both `flex` and `inline-flex` are inner display modes. We set these values on the container, which behaves like a block-level or inline-level box, respectively. The children of that container are then arranged according to the rules of flex layout.

By adding `display: flex` or `display: inline-flex` to a containing element, its immediate children become *flex items*. Flex items may be element children or non-empty text nodes, as shown in the following example:

```
<div style="display: flex; width: 1000px; margin: 5rem
auto;">
    <span>This text is contained by a SPAN element.</span>
    <b>This text is contained by a B element.</b>
    This un-wrapped text node still behaves like a flex item.
</div>
```

Flex items don't have to be elements. Non-empty text nodes can also be flex items. The code above produces the layout shown below.

Let’s look at an example using the markup below. Notice here that the contents of `.alpha`, `.beta`, `.gamma`, and `.delta` differ in size:

```
<div class="flex-container">
    <div class="alpha">
        <b>A</b>
    </div>
    <div class="beta">
        <b>B</b>
        <p>This is a short sentence made of short words.</p>
        <p>Antidisestablishmentarianism is a long word.</p>
    </div>
    <div class="gamma">
        <b>C</b>
        <img src="400x300.png" width="400" height="300"
alt="placeholder">
    </div>
    <div class="delta">
        <b>D</b>
    </div>
    <div class="epsilon">
        <b>E</b>
    </div>
</div>
```

The CSS we’ll use is simple. We’ll add a `display: flex` declaration to `.flex-container`, and set its inline size to 1500 pixels:

```
.flex-container {
    display: flex;
    inline-size: 1500px; /* Can also use width: 1500px */
}
```

The image below shows the result: a simple flexible box layout with `display: flex` applied, and no other Flexbox properties. The dashed line represents the boundaries of our flex container. Each flex item is about as wide as its contents, plus whatever padding exists inside the flex item.

This example uses the initial value of `flex`, which is `0 1 auto`. The `flex` property applies to the *children* of a flex container. Since the `flex` property sits at the heart of Flexbox, it's important to understand how it works.

Understanding the `flex` Property

The `flex` property is actually a shorthand for three other properties:

- `flex-grow` indicates the factor by which an element should *grow*, if necessary, and must be a positive integer. Its initial value is `0`.
- `flex-shrink` indicates the factor by which that an element should *shrink*, if necessary, and must be a positive integer. Its initial value is `1`.
- `flex-basis:` indicates the initial or minimum size of a flex item. This may be its width when the main axis is horizontal (for example, with `flex-direction: row`), or the flex item's height when the main axis is vertical. (See the note below for more detail on this.) The value of `flex-basis` must be a valid value for the `width` property.

Horizontal and Vertical Writing Modes

There's a little more nuance to the point above about horizontal and vertical axes. A flex item's main size is determined by a combination of its writing mode and whether the value of `flex-direction` is `row` or `column`. For **horizontal writing modes**:

- when `flex-direction: row`, the main axis is horizontal, and the main size is the flex item's *width*
- when `flex-direction: column`, the main axis is vertical, the main size is the flex item's *height*

For **vertical writing modes**:

- when `flex-direction: row`, the main axis is vertical, and the main size is the flex item's *height*
- when `flex-direction: column`, the main axis is horizontal, and the main size is the flex item's *width*

When `flex-basis` is `auto`, the maximum size of the flex item becomes its initial size, also known as the *flex base size*. In the example above, `.gamma` is 400 pixels wide because one of its children is 400 pixels wide. Similarly, `.alpha` (**A**), `.beta` (**B**), `.delta` (**D**), and `.epsilon` (**E**) are as wide as their longest line of text. For `.beta`, that's 389.633 pixels. For `.alpha`, `.delta`, and `.epsilon`, that ranges from 37.333 to 47.733 pixels wide, depending on the width of the letters A, D, and E.

Limiting Maximum Size

You can limit the maximum size of a flex item with the `max-width` property. In that case, the browser uses the `max-width` as the initial size.

The value of each flex item's `flex-basis` determines whether the browser will use `flex-grow` or `flex-shrink` when allocating free space within the flex container. When the sum of the flex base size for each flex item is greater than the inner size of the flex container, the browser allocates space using `flex-shrink`. If it's less than the inner size of the flex container, it uses `flex-grow`. This is the **used flex factor**.

Let's add the flex basis values of our flex items from the example above:

```
47.733 + 386.663 + 400 + 43.983 + 37.333 = 915.712
```

Remember that our container is 1500 pixels wide. Since 915.712 pixels is less than 1500 pixels, the browser uses the `flex-grow` value to allocate free space inside the container; `flex-shrink` is ignored.

In this case, the initial `flex: 0 1 auto` declaration means that each flex item grows from its initial size by a factor of zero. In other words, it won't grow at all.

Using the `flex` Property

Although it's possible to set `flex-grow`, `flex-shrink`, and `flex-basis` individually, the specification recommends using the `flex` shorthand property. It accepts one, two, or three values.

When using **one-value syntax**, the value must be a number *or* one of the `initial`, `auto`, or `none` keywords. When the value is a number—say, `flex: 4`—this value is interpreted as the value for `flex-grow`. The `flex-shrink` value is assumed to be `1`, which is its initial value. However, the value of `flex-basis` is assumed to be `0`, instead of its initial value of `auto`. In other words, `flex: 4` is the equivalent of `flex: 4 1 0` and *not* `flex: 4 1 auto`. That seems counterintuitive at first, but changing the value of `flex-basis` to `0` makes `flex-grow` and `flex-shrink` behave more predictably.

When using a **two-value syntax**, the first value must be a number. Here, too, it's interpreted as the value of `flex-grow`. The second value, however, can be either a number *or* a valid value of the `width` property.

If the second value is a *number*, it's interpreted as a value of `flex-shrink`, and the value of `flex-basis` is assumed to be `0`. If it's a width value, it's interpreted as the `flex-basis` value, and `flex-shrink` is assumed to be `1`.

Two-value Dangers

Be careful using the two-value syntax. If your intent is to set a width of `0` for `flex-basis`, include a unit so that it's not misinterpreted as the `flex-shrink` value.

To explain it another way, take this CSS:

```
.item {
    flex: 4 1;
}
```

The code above is the equivalent of this:

```
.item {
    flex-grow: 4;
    flex-shrink: 1;
    flex-basis: 0%;
}
```

On the other hand, take this CSS:

```
.item {
    flex: 2 auto;
}
```

It's equivalent to this:

```
.item {
    flex-grow: 2;
    flex-shrink: 1;
    flex-basis: auto;
}
```

When using the three-value syntax, values are interpreted as `flex-grow`, `flex-shrink`, and `flex-basis`, in that order. The first two values *must* be numbers. The last value *must* be a valid value for the `width` property.

Flex Factors and Space Distribution

Both `flex-grow` and `flex-shrink` represent proportions. They tell the browser how to allocate the free space inside of a flex container.

So how does the browser determine the free space? First, it adds the flex base size of every flex item and deducts that from the inner width of the flex container. Then it subtracts the size of *inflexible* items. An item is considered **inflexible** if:

- its flex factor is zero
- its flex base size is greater than its hypothetical main size, and the browser is using the flex shrink factor
- its flex base size is smaller than its hypothetical main size, and the browser is using the flex grow factor

The space that remains is the free space. Keep in mind that both `gap` and `margin` can further reduce the amount of free space in a container.

Let’s return to our previous example. We’ll add an explicit value for `flex` to all of our flex items:

```
.flex-container > div {
    flex: 1;
}
```

The image below shows the result.

Adding `flex: 1` to a flex item is the equivalent of adding `flex: 1 1 0`. However, `.gamma img` has an intrinsic width of 400 pixels, so `.gamma` will be at least 400 pixels wide. Since the browser already knows what size `.gamma` needs to be, the browser subtracts its width from that of the container: 1500 minus 400 equals *1100 pixels* of free space.

Once the browser has determined the free space available, it calculates the main size of each *flexible* item:

- If the flex used factor is `flex-grow`, the calculation is roughly: *free space ÷ sum of each flexible item's* `flex-grow` *value* × `flex-grow`*.*
- If the flex used factor is `flex-shrink`, the calculation is roughly: *flex base size - ((free space ÷ sum of each flexible item's* `flex-shrink` *value)* × `flex-shrink`*).*

Gaps and margins also affect size of flex items. Margins don't collapse in a flex formatting context. Instead, margins along the main axis are deducted from the main size of each flex item. Gaps set using the `row-gap`, `column-gap` or `gap` properties, work similarly. Each gap reduces the size of a flex item by one half of the gap length.

The specification details a far more complex process for this, but for the purposes of this book, the explanation above will do.

Let's apply this formula to our current example. We have four items with an *indefinite* or undetermined width, and a `flex-grow` factor of `1`: *1100 ÷ (1 + 1 + 1 + 1) = 275*. For each flex item, multiply 275 by the value of that item's `flex-grow` property. In this case, all of our items will be 275 pixels wide.

Here's another example. We'll set the `flex-grow` value of `.epsilon` to 5. It inherits the `flex-shrink` and `flex-basis` value of the `.flex-container > div` rule set:

```
/* Remember this is the same as flex: 1 1 0; */
.flex-container > div {
    flex: 1;
}
.flex-container > .epsilon {
    flex-grow: 5;
}
```

You can see the result in the image below. The `flex-grow` factor of 5 changes the ratio by which space gets distributed within the flex container.

Since “Antidisestablishmentarianism” is one (long) word, it forces `.beta` to be 249 pixels wide. In this case, the browser deducts the width of `.beta` and `.gamma` from the inner width of the flex container: *1500 - (400 + 249) ≈ 851*. Our flex container has 851 pixels of free space to distribute.

Let’s determine the size of our flexible items. We’ll add the flex factors for `.alpha`, `.delta`, and `.epsilon`, then divide our free space value by that number: *851 ÷ (1 + 1 + 5) = 121.571*. Now we can multiply this number by our flex grow factor to determine the size of each flex item:

- `.alpha`: *121.571 × 1 121.571*
- `.beta`: *121.571 × 1 = 121.571*
- `.epsilon`: *121.571 × 5 = 607.855*

Flex item `.epsilon` is roughly five times as wide as `.alpha` and `.beta`.

When Flex Items Shrink

When the chosen flex mode is `flex-shrink`, the browser determines how much space to shrink each flex item by. Remember, flex items shrink when their initial main size exceeds the inner main size of the container. Let’s change the `flex-basis` value of our flex items from `0` to `410px`. We’ll also give `.epsilon` a `flex-shrink` factor of 5:

```
.flex-container > div {
    flex: 1 1 410px;
}
.flex-container > .epsilon {
    flex-shrink: 5;
}
```

This makes the sum of the hypothetical main size of each flex item 2050 pixels —greater than the 1500 pixel width of our container. First, determine the amount of free space available: *1500 - 2050 = -550 pixels*.

Next, account for the width of `.gamma img`. It’s still 400 pixels wide, which means `.gamma` shrinks by ten pixels—the difference between its contents and `flex-basis`. Deduct that ten-pixel difference from our free space: *-550 + 10 = -540 pixels*.

We’ve accounted for the width of `.gamma`, so we can perform the next step. Divide the amount of free space by the sum of the flex factors of our flexible items, `.alpha`, `.beta`, `.delta`, and `.epsilon`: *-540 ÷ (1 + 1 + 1 + 5) ≈ -67.5*.

Now we can calculate the size of each flex item:

- `.alpha`: *410 + (-67.5 × 1) = 342.5 pixels*
- `.beta`: *410 + (-67.5 × 1) = 342.5 pixels*
- `.delta`: *410 + (-67.5 × 1) = 342.5 pixels*
- `.epsilon`: *410 + (-67.5 × 5) = 72.5 pixels*

The image below shows the result. When the chosen flex factor is `flex-shrink`, the browser shrinks flex items proportionately.

Creating Multi-line Flexible Layouts

So far, our examples have looked at flex box spacing in a single direction, along a single line. We can also make flex items wrap across multiple lines using the `flex-wrap` property. Its initial value is `nowrap`. Other values are `wrap` and `wrap-reverse`.

The spacing formulas for `flex-grow` and `flex-shrink` work *per line*. Flex items wrap and form a new line when the sum of their hypothetical main size exceeds the inner main size of the flex container. Take the following CSS:

```
.flex-container {
    display: flex;
    width: 1500px;
    gap: 20px;
    flex-wrap: wrap;
}
.flex-container > div {
    flex: 0 1 33.333%;
}
```

Since `flex-basis` is 33.33%, each flex item should fill one third of the available space. But this time, we've added a gap of 20 pixels. Although the sum of the hypothetical main size of these flex items is 1500 pixels—33.333% of 1500 is roughly 500 pixels, and *500 + 500 + 500 = 1500*—the additional gap changes how many items can fit on a line—as illustrated below.

Only two flex items fit on each line. Because our `flex-grow` value is zero, these flex items don’t expand to fill the width of our flex container. If we changed the declaration to `flex: 1 1 33.333%;`, they would.

The `wrap-reverse` value works the same way as `wrap`, but reverses the *visual ordering* of flex items. The following image shows how `flex-wrap: wrap-reverse` changes a flex layout, reversing the visual ordering of the container’s children.

The examples we’ve used thus far have looked at horizontal spacing. But in some circumstances, you may want to distribute space vertically. We’ll discuss the `flex-direction` property in the next section.

Distributing Space Vertically with `flex-direction`

The `flex-direction` property lets us change the main axis direction of our flex container. Its initial value is `row`; other values are `column` and `column-reverse`. The following image shows a flex container with a `flex-direction` value of `column`, which changes the axis along which space space gets distributed.

The title of this section is a little misleading. The `flex-column` also depends on the writing mode of the document. In languages that are written and read horizontally, columns are vertical, as shown by the image above. Vertical languages rotate the row and column axes by 90 degrees clockwise (as with `writing-mode: vertical-rl`) or counterclockwise (as with `writing-mode: vertical-lr`) . Consider the CSS below:

```
.flex-container {
    display: flex;
    flex-direction: column;
    writing-mode: vertical-rl;
}
```

This code creates the layout pictured below.

When using `flex-direction: column` with multilingual sites, the `block-size` property is a better choice than `height` or `max-height`. Remember that `block-size` and `inline-size` are relative to the writing mode, rather than to the vertical and horizontal dimensions as `height` and `width` are. Using logical properties removes the need to reset properties based on the document's language.

Creating Layouts with CSS Grid

CSS Grid allows us to create two-dimensional grid-based layouts that were previously impossible, or only possible with lots of JavaScript.

Keep in mind that the CSS Grid specification is dense, and it introduces several new concepts that are a bit complex. Consider this section an overview rather than a comprehensive look at Grid. Don't worry, though: we'll point you to lots of resources for learning more.

The Grid Formatting Context

Adding `display: grid` to an element triggers a *grid formatting context* for that element and its children. In a grid formatting context, three things happen:

- The element becomes a block-level element that participates in the normal flow.
- Its children—whether elements or text nodes—create block-like, grid-level boxes that can be arranged into rows and columns. Immediate children of a grid container are **grid items**.
- In a horizontal writing mode, each member in a grid row will have the same height as its tallest element (as determined by content), unless an explicit height value is set. When the document uses a vertical writing mode, it takes on the same length as its longest element (as determined by content).

The image below illustrates how using `display: grid` creates a block-level container, and block boxes for its children.

Using `display: inline-grid` works similarly. Children of inline-level grid containers create grid-level boxes, but the container itself participates in an inline formatting context.

By themselves, `display: grid` and `display: inline-grid` won't automatically arrange these boxes into rows and columns. We also need to tell the browser where and how to place things.

Before creating your grid, determine whether you want a fixed number of columns and/or rows, whether you'd like the browser to calculate the number of columns and rows automatically, or whether you'd like a mix of the two. Knowing what kind of grid you want to create determines the approach you'll take. Let's look at a few techniques.

Defining a Grid Layout

After defining a grid container, we'll need to tell the browser how many rows and columns our grid should contain. We can define the number of rows and columns using the `grid-template-rows` and `grid-template-columns` properties. They're applied to the grid container.

Both `grid-template-rows` and `grid-template-columns` accept what's known as a *track list*. The **track list** is a space-separated string that specifies grid line names and sizes of each position in the row or column.

Each value in a track list creates a new space—a **track**—within the row or column. You can use lengths, flexible length units (discussed later in this chapter), or percentages. You can also use sizing values such as `auto`, `min-content` and `max-conent`.

Let's define a grid with three columns, each `25rem` units wide and two rows, each `10rem` units tall:

```
.grid {
    display: grid;
    grid-template-columns: 35rem 35rem 35rem;
    grid-template-rows: 10rem 10rem;
}
```

Let's now apply that CSS to the following HTML. Yes, this is all the markup required:

```
<div class="grid">
    <div>Grid item A</div>
    <div>Grid item B</div>
    <div>Grid item C</div>
    <div>Grid item D</div>
    <div>Grid item E</div>
</div>
```

We've created an explicit grid, organized into the columns and rows, with `grid-template-columns` and `grid-template-rows`. The result is pictured below.

Here, we've created a grid of evenly sized rows and columns, but that isn't a requirement of Grid. Let's tweak our CSS slightly. We'll change the value of `grid-template-columns` to `40rem 35rem 25rem`:

```
.grid {
    display: grid;
    grid-template-columns: 40rem 35rem 25rem;
    grid-template-rows: 40rem 10rem;
}
```

Now the second column in our grid is narrower than the first and third, as pictured below.

Explicit Grid versus Implicit Grids

In the previous section, we explicitly stated that this grid should have six available grid cells formed by three columns and two rows. This is what’s known as an **explicit grid**. Here, our grid container only has five children. The remaining position is empty. What if we add more children to the container? When grid items exceed the number of explicitly defined cells, the remaining items are arranged in an implicit grid.

Now we have three rows. Notice, however, that our third row is only as tall as its contents and padding. It’s part of the grid because these items are the children of a grid container. Yet the row isn’t explicitly defined by `grid-`

`template-rows`. What we have instead is an **implicit grid**—an explicit grid with additional grid items that exceed the defined number of explicit grid cells.

Items within an implicit grid are sized `auto` by default. Grid items will expand to accommodate their contents, or fill the remaining vertical space in the container—whichever is taller. If, for example, we set the `height` property of our container to `100rem`, our implicit grid track will expand to be `60rem` tall, because implicit grid rows expand to fill the available height of the container.

If we add enough items to create a fourth row, the height of our implicit grid items will be distributed evenly across the remaining `60rem` of vertical space in the container. Their computed height will be `30rem` each.

In our original example, we've explicitly defined only two rows with a height of `10rem` each, so our third row defaults to `auto` sizing. Its height will adjust to the size of its contents and padding.

Specifying Track Size for an Implicit Grid

It's possible, however, to set a kind of default height or width for implicit grid items using the `grid-auto-rows` and `grid-auto-columns` properties. Let's update our CSS with `grid-auto-rows`:

```
.grid {
    display: grid;
    grid-template-columns: 25rem 15rem 25rem;
    grid-template-rows: 10rem 10rem;
    grid-auto-rows: 30rem;
}
```

Now items in our third row—and any subsequent rows—will be `30rem` in height.

There’s one drawback to using the `grid-auto-*` properties: when the contents of a grid item exceed its dimensions, they will overflow the container (as shown below), and may be clipped visually by elements in other rows. This can happen when using length or percentage units.

One way to avoid this is to use the `minmax()` function. Let's rewrite our CSS to use `minmax()`:

```
.grid {
    display: grid;
    grid-template-columns: 25rem 15rem 25rem;
    grid-template-rows: 10rem 10rem;
    grid-auto-rows: minmax( 30rem, auto );
}
```

As you may have guessed from its name, `minmax()` lets us define the minimum and maximum size of a track. It requires two arguments, the first of which is the minimum desired track size. The second argument is the maximum desired size.

In this case, our row will be at least `30rems` high. But since we've set our maximum size to `auto`, our track will expand to accommodate the content of that cell. Arguments for `minmax()` can be lengths or percentages, or one of the `auto`, `min-content`, and `max-content` keywords. Flexible length units, discussed in the next section, are also valid.

Lengths and percentages can be used to define track sizes. Using them may mean that the grid items don't fill the entire width or height of the container. For example, if our grid container is `70rem` wide, `grid-template-columns: 25rem 15rem 25rem;` will only fill about 90% of its horizontal space. On the other hand, if our grid container is only `50rem` wide, the total width of our columns will overflow the container's bounds. To prevent this, use *flexible length units*.

Creating Flexible Grids with Flex Units

Flexible length or **flex** units are expressed using the `fr` unit indicator. Flex units tell the browser what fraction or proportion of the leftover space in a grid container should be allocated to each grid item. They're a ratio, not a true length value in the way `px`, `em`, or `cm` are.

There's a formula for calculating the used width of an item when using flexible units: *(flex × leftover space) ÷ sum of all flex factors*. Leftover space is what remains after deducting the known size of items (the specification calls this “definite size”), the size of grid gaps, and grid item padding from the size of the grid container.

Consider the CSS below:

```
[id=grid] {
    display: grid;
    grid-template-columns: 3fr 2fr 1fr;
    width: 1500px;
}
[id=grid] > div {
    padding: 10px;
}
```

We’ll pair it with the following HTML:

```
<div id="grid">
    <div>Grid item A</div>
    <div>Grid item B</div>
    <div>
        Grid item C
        <img src="400x300.png">
    </div>
    <div>Grid item D</div>
    <div>Grid item E</div>
    <div>Grid item F</div>
</div>
```

You can see the result in the image below, which illustrates how flexible length units maintain grid proportions, rather than absolute lengths.

The image in our grid has intrinsic dimensions: a width of 400 pixels and height of 300 pixels. As a result, the third column must be at least 400 pixels wide. Each grid item also has ten pixels of padding along both the horizontal and vertical axes. That increases the size of our column by 20 pixels, for a width of `420px`.

Next, the browser subtracts that width from the width of our grid container: *1500 - 420 = 1080*. That leaves 1,080 pixels of leftover space to distribute across the other two columns of this grid.

Dividing *1080* by *5*—the sum of the flex factors of the first two columns—gives us a quotient of 216. Multiply the first two flex factors by that quotient and we end up with columns that are `648px` (*216 times 3*) and `432px` (*216 times 2*) wide, as shown above. It's very similar to the way browsers distribute remaining space for flexible box layout.

Because these units are ratios and not absolute lengths, `grid-template-columns: 2fr 2fr 2fr` is equivalent to `grid-template-columns: 1fr 1fr 1fr`.

Not True Length Units

`fr` units are not true length values. This makes them incompatible with other length units such as `px` and `rem`. It also means that you can't use `fr` units with the `calc()` function. For example, `calc(1fr - 1rem)` is an invalid length value.

Using the `grid-template` Shorthand Property

We can also indicate the number of rows and columns using the `grid-template` property. Its syntax is as follows:

```
grid-template: [row track list] / [column track list]
```

Remember this block of CSS from earlier in the chapter?

```
.grid {
    display: grid;
    grid-template-columns: 25rem 25rem 25rem;
    grid-template-rows: 10rem 10rem;
}
```

We can combine the second and third lines using `grid-template`:

```
.grid {
    display: grid;
    grid-template: 10rem 10rem / 25rem 25rem 25rem;
}
```

For clarity, however, you may still prefer to use the longhand properties.

Repeating Rows and Columns

In many cases, you'll want grid columns or rows that repeat automatically; think of a list of products for sale, or recipe search results. Grid offers a syntax for that—the `repeat()` function:

```
.grid {
    display: grid;
    grid-template-columns: repeat( 3, 1fr );
}
```

The `repeat()` function accepts two arguments:

- the number of times to repeat the track list
- a track list to repeat

Arguments must be separated by a comma. The first argument may be a positive integer, or the `auto-fit` or `auto-fill` keywords. The above CSS produces the following grid. Our `1fr` track list is repeated three times.

We could also use a two-column pattern that repeats twice. For example, `grid-template-columns: repeat( 2, 1fr 3fr );` produces a four-column grid. As the next image shows, the first and third columns are one third the width of the second and fourth. In both cases, the value of `grid-template-rows` is `auto`.

Repeating Columns with `auto-fit` or `auto-fill`

Both of the preceding examples tell the browser: “Here’s a track list pattern; please repeat it X number of times.” What you may want to tell the browser instead, is: “Please fit as many columns or rows as you can within this grid container.” For that, we can use `auto-fit` or `auto-fill` as the first argument for `repeat()`, in combination with `minmax()`.

What’s the difference between `auto-fit` and `auto-fill`?

- `auto-fit` fits as many grid items as it can within a track line, and collapses empty tracks.
- `auto-fill` fits as many grid items as it can within a track line, but *doesn’t* collapse empty tracks.

This difference becomes apparent when the grid container’s width exceeds the maximum total width of its grid items. Let’s compare some CSS:

```
.grid {
    display: grid;
    width: 1500px;
}
.autofill {
    grid-template-columns: repeat( auto-fill, minmax( 100px,
1fr ) );
}
.autofit {
    grid-template-columns: repeat( auto-fit, minmax( 100px,
1fr ) );
}
```

And let’s apply this CSS to the HTML below:

```
<div class="grid autofill">
    <div>Grid item A</div>
    <div>Grid item B</div>
    <div>Grid item C</div>
    <div>Grid item D </div>
    <div>Grid item E</div>
</div>
<div class="grid autofit">
    <div>Grid item A</div>
    <div>Grid item B</div>
    <div>Grid item C</div>
    <div>Grid item D </div>
    <div>Grid item E</div>
</div>
```

The only difference between these two grid layouts is that one uses `auto-fill` and the other uses `auto-fit`. But compare the two grids in the image below.

In both grids, the total maximum width of the grid items is less than that of the grid container. However, in the top grid—our `auto-fill` grid—that excess space is *filled in* by additional, empty grid items. The following image provides a visualization of the difference between `auto-fill` and `auto-fit` provided by the Firefox grid inspector.

The empty, `auto-fill` cells are highlighted with dotted lines. Compare that to the bottom grid, in which each grid item is stretched to fit the available space.

More on Auto-sizing Columns

If this still doesn't make any sense, read Sara Soueidan's "Auto-sizing Columns in CSS Grid: `auto-fill` vs `auto-fit`". It contains some video examples that illustrate the difference better than static images can.

Line-based Grid Placement

So far, we've discussed simple grids that are neatly aligned rows and columns of boxes. But Grid layout is far more robust and flexible than that. We can also use it to create complex layouts, like the one pictured below.

The layout pictured above uses *line-based grid placement*, a core feature of CSS Grid. We'll look at how to create this layout later in this section. But first, let's discuss grid lines.

Understanding Grid Lines

Grid lines are horizontal and vertical lines that separate rows and columns, as shown below. These lines exist on each side of a row or column, but don't affect its dimensions.

The space between each grid line is known as a **grid track**. A grid track can be a row *or* a column; the phrase itself is a generic term for both. Grid columns and grid rows intersect to form **grid cells**.

Most desktop browsers have grid layout inspectors as part of their developer tools. In Firefox, look for the crosshatch icon between `display` and `grid`. Clicking that crosshatch icon displays (or hides) the grid overlay.

In Chrome, Safari and Edge, look instead for the **grid** label next to a grid container in the Elements panel. Clicking the label activates the grid inspector in those browsers.

The image below shows the Firefox grid overlay in action.

Notice that each edge of each column in this grid is bounded by a grid line, and each of these lines has a numeric index. The same is true for each row.

Grid line numbering begins with 1, and the count begins at the start of the grid container. When the text direction is left to right, the starting point is the left edge. When the direction is right to left, the starting point is the right edge.

Each grid line may also have a negative index. Negative index counts begin with -1, and decrement from the ending edge of the explicit grid. So an index of -1 specifies the ending edge of the container, while an index of -2 specifies one grid line in from that one, and so on. (We'll see an example of negative line numbers in use shortly.)

Line numbers can be used to place items within the grid using the `grid-column-start`/`grid-column-end` and `grid-row-start`/`grid-row-end` properties. Here's an example:

```
.grid-10cols {
    display: grid;
}
#a {
    grid-column-start: 1;
    grid-column-end: 11;
}
#b {
    grid-column-start: 1;
    grid-column-end: 6;
}
#c {
    grid-column-start: 6;
    grid-column-end: 11;
}
```

We’ll pair that CSS with the HTML below:

```
<div class="grid-10cols">
    <div id="a">Grid item A</div>
    <div id="b">Grid item B</div>
    <div id="c">Grid item C</div>
</div>
```

The image below illustrates the result.

As shown above, `#a` fills the space between line 1 and line 11, or the entire width of the grid. `#b` begins at the first line and ends at the sixth. `#c` begins at the sixth line and extends to line 11. With `grid-*-start` and `grid-*-end`, we’re telling the browser to align the starting and ending edges of our grid items with specific grid lines.

We haven’t used either of the `grid-template-*` properties in this example. By defining a start line and an end line, we’ve created ten *implicit grid tracks*. Note that because this grid hasn’t been explicitly defined—with something like `grid-template-columns: repeat(10, 1fr)`—we've lost the ability to use negative grid line indexes for placement.

Spanning Rows or Columns

In the example above, we've used line numbers to indicate where our grid items should begin and end. Another way to do this is with the `span` keyword. The `span` keyword indicates how many tracks—that is, how many rows or columns—a grid item should occupy. We could, in other words, rewrite our CSS like so:

```
.grid-10cols {
    display: grid;
}
#a {
    grid-column-start: span 10;
}
#b {
    grid-column-start: span 5;
}
#c {
    grid-column-start: span 5;
}
```

Again, `span` indicates how many columns or rows a grid item should occupy. Line indexes indicate where to align the edges of a grid item.

Complex Layouts with Line-based Placement

Let's return to some code we used earlier. Once again, our markup is simple—a containing `<div>` with eight children:

```
<div class="grid-10cols-complex">
    <div id="a">Grid item A</div>
    <div id="b">Grid item B</div>
    <div id="c">Grid item C</div>
    <div id="d">Grid item D </div>
    <div id="e">Grid item E</div>
    <div id="f">Grid item F</div>
    <div id="g">Grid item G</div>
    <div id="h">Grid item H</div>
</div>
```

For this layout, we'll explicitly define a five-row, ten-column grid:

```
.grid-10cols-complex {
    display: grid;
    /* Syntax: grid-template: [rows] / [columns] */
    grid-template: repeat(5, 9.5rem) / repeat(10, 10%);
}
```

Explicitly defining a grid isn't strictly necessary for line-based placement. In this case, however, it ensures that each box in our layout has the right proportions. The next step is to place our grid items:

```
#a, #h {
    grid-column-start: span 10;  /* Span the entire grid */
}
#b {
    grid-row-start: span 3;
    grid-column-start: span 2;
}
#c, #d {
    grid-column-start: span 4;
}
#e, #f {
    grid-column-start: span 3;
}
#f, #g {
    grid-column-end: -1;  /* Begin from the container's
ending edge */
}
#g {
    grid-column-start: span 5;
}
```

Here, both `#a` and `#h` span all ten columns of our grid, while the other elements span between two and five columns, as illustrated below. Element `#b` also spans three rows.

Notice that for `#f` and `#g`, we've used a negative line index. Remember that negative line indexes begin at the ending edge of the container. With `grid-column-end: -1`, we've told the browser to *align the ending edge of `#f` and `#g` with the ending edge of our grid container*. These elements still span three and five columns, respectively, but their ending edges align with the ending edge of the container.

Using Named Grid Areas

One of the more clever aspects of CSS Grid is *template areas*. **Template areas** use the `grid-template-areas` property, and let us define our grid in terms of named slots. We can use template areas, in combination with grid placement properties, to define complex grid layouts that are still readable.

Take the layout shown in the image below. It's a fairly conventional, two-column layout with a header, footer, and main content area, along with a right-hand navigation menu.

Here’s the markup we’ll use to create this page. It’s simplified to emphasize the document’s structure:

```
<!DOCTYPE html>
<html lang="en-US">
    <head>
        <title>GoodRecipes! Tuna with zucchini
noodles</title>
    </head>

    <body>
        <header>…</header>
        <article>…</article>
        <nav>…</nav>
        <footer>…</footer>
    </body>
</html>
```

Named template areas can be hard to understand at first. We still need to define rows and columns. Then we can define named areas that span some or all of those rows and columns using the `grid-template-areas` property. Here’s an example:

```
body {
    display: grid;
    /*
    Using the longhand properties for
    the sake of clarity. We could also use
    grid-template: repeat(2, auto) / 4fr 1fr
    instead.
    */
    grid-template-rows: repeat(2, auto);
    grid-template-columns: 4fr 1fr;
    grid-template-areas: "pagehead pagehead"
                         "mains navigation"
                         "pagefoot pagefoot";
}
```

Yes, the syntax of `grid-template-areas` is a little weird. Template areas are strings and must be enclosed in single or double quotes. Each template area corresponds to a row in the grid. Columns within each row are delineated by a space.

Line Breaks Not Required

You’re not required to use line breaks when setting the value of `grid-template-areas`. We could put our definition on a single line: `grid-template-areas: "pagehead pagehead" "mains navigation" "pagefoot pagefoot";`. Line breaks do, however, make it easier to visualize and understand the layout.

Now, in order for `grid-template-areas` to work, we have to account for every position in the grid. That’s why we’re repeating `pagehead` and `pagefoot`. Repeating a name within a template string indicates that the area should span multiple columns.

Once we’ve defined our template areas, the last step is to assign our elements to each area using the `grid-area` property:

```
header {
    grid-area: pagehead;
}
article {
    grid-area: mains;
}
nav {
    grid-area: navigation;
}
footer {
    grid-area: pagefoot;
}
```

This tells the browser to place the `<header>` element in the `pagehead` area, the `<article>` element in the `mains` area, and so forth.

Spacing Grid Items

In all of the grids we've created thus far, the edges of our grid items abut each other. As with multicolumn layout, however, we can add gutters to our grid. We know that the `column-gap` property adds spacing between columns. With grid layout, we can also use the `row-gap` property to add spacing between grid rows. Both properties apply to the grid container:

```
.grid {
    display: grid;
    grid-template: 20rem 20rem / 35rem 35rem 35rem 35rem;
    column-gap: 1rem;
    row-gap: 1rem;
}
```

The image below shows the effect of adding `1rem` row and column gaps.

In a grid formatting context, `column-gap: normal` and `row-gap: normal` resolve to a used value of `0px`. That behavior differs from multicolumn layout, where `column-gap: normal` resolves to a used value of `1em`.

Only length and percentage values are valid for `column-gap` and `row-gap` (and the `gap` shorthand property). If you'd rather have the browser automatically distribute boxes along each grid axis, use `justify-content` or `align-items` instead. We'll discuss both properties in the "Box Alignment and Distribution" section below.

The `gap` Shorthand Property

We can also specify both `column-gap` and `row-gap` at once using the `gap` shorthand property. The first value of `gap` becomes the size of the row gap; the second value is the column gap. Providing only one value sets the same gap size for both properties. In other words, we can rewrite `column-gap: 1rem; row-gap: 1rem;` as `gap: 1rem;`.

Older versions of the Grid specification defined `grid-column-gap` and `grid-row-gap` properties. These have been replaced by `column-gap`, which can also be used with multicolumn layout and Flexbox in most browsers. (Safari, as of version 15, doesn't support the use of `gap` with multicolumn layout.) For compatibility with older browsers, include the legacy `grid-row-gap` and `grid-column-gap` properties in addition to `column-gap` and `row-gap`. Or, if you use the shorthand `gap` property, include `grid-gap` as well.

Using `column-gap` and `row-gap` aren't the only ways to space grid content. We can also use the `justify-*` and `align*` properties to distribute grid items within the available space. Since most of these properties are common to Grid *and* Flexbox, we'll discuss them together in the section "Box Alignment and Distribution" later in this chapter.

Grid Items and Margins

Grid items can have margins of their own. However, margins work a bit differently in a grid formatting context than they do in a block formatting context.

Grid cells, and the grid lines that bound them, form containing blocks for grid items. As a result, adjacent margins of grid items *do not* collapse. That's the opposite of what happens in a block formatting context.

For grid items, top and bottom margins of `1rem` result in `2rem` of space between the content boxes, as shown above. And because grid item margins fall within the containing block, they may affect the dimensions of `auto`-sized grid tracks.

Images within Grids

Images within grid cells work similarly to the way they behave in multicolumn layouts. When the track size uses length or percentage units, images may overflow the grid cell if their dimensions exceed those of the cell.

However, when the track sizing function is `auto`, or uses flex units (`fr`), the track containing that grid cell expands to accommodate the image.

As in multicolumn layout, we can constrain the image dimensions to those of its grid cell by setting its width to 100%.

Floating an image or other elements *within a grid cell* works as you'd expect. But you can't float a grid item. Floated siblings of grid containers also don't intrude on the grid container.

Progressively Enhanced Layouts with Grid and `display: contents`

Earlier in this chapter, we talked about the `contents` value of the `display` property. Remember that `display: contents` prevents the browser from generating an element box. In a grid formatting context, this turns grandchild elements into grid items.

Say you have extra `<div>` elements in your markup to create a grid-like layout in browsers that don't support CSS Grid, as we have in the following block of code:

```
<div class="grid">
    <div class="grid-row">
        <div>Item 1</div>
        <div>Item 2</div>
        <div>Item 3</div>
        <div>Item 4</div>
    </div>
    <div class="grid-row">
        <div>Item 5</div>
        <div>Item 6</div>
        <div>Item 7</div>
        <div>Item 8</div>
    </div>
</div>
```

In browsers that don't support Grid, you might use Flexbox to create a grid-like layout like the one shown below:

```
.grid {
    width: 80%;
    margin: auto;
}
.grid-row {
    display: flex;
}
.grid-row > * {
    flex: 0 0 calc(25% - 2rem);
    background-color: #121212;
    color: var(--color-a);
    margin: 0 2rem 2rem 0;
    padding: 1rem;
}
```

Then to progressively enhance this layout, you might add `display: grid` to `div.grid`. Doing so, however, turns those `.grid-row` elements into grid items. As a result, the children of `.grid-row` don’t participate in a grid formatting context, as illustrated below.

If we add `display: contents` to `.grid-row`, though, the grandchild elements participate in the grid formatting context created by `div.grid`. Here’s that CSS:

```
@supports (display: grid) {
    .grid {
        display: grid;
        grid-template-columns: repeat(4, 1fr);
    }
    .grid-row {
        display: contents;
    }
    .grid-row > * {
        width: unset;
    }
}
```

The image below shows the resulting layout.

Notice that we didn’t undo or override the `flex` declaration. Since we’ve switched the `display` value to `grid`, flex-related properties no longer apply and they’re ignored.

Although `display: contents` can cause severe accessibility issues, in this particular instance it doesn’t. Unlike lists or headings, `<div>` elements don’t describe document structure, nor do they have defined functionality in the way that elements such as `<button>` and `<input>` do. Because there’s no behavior to break, using `display: contents` for our `<div>` element doesn’t break anything.

Grid Conclusion

CSS Grid is a dense topic. We’ve really just scratched the surface here. Luckily, there’s a wealth of resources that can help you learn more.

I believe in reading specifications where possible. In my opinion, the CSS Grid specification is quite readable, and it’s a good place to begin your own explorations of grid layout. But specifications do tend to contain a lot of jargon. They’re written not only for web developers, but also for those tasked with implementing them in browsers.

Rachel Andrew’s Grid by Example was created for a web developer audience. The site includes grid layout tutorials and a collection of common user interface patterns. Be sure to visit the site’s Resources section too. It’s a cornucopia of links that demonstrate what you can do with CSS Grid.

Jen Simmons’ Experimental Layout Lab is also chock-full of examples that illustrate Grid’s possibilities. If video is more your style, Simmons’ Layout Land YouTube channel includes video walk-throughs of grid and other layout topics.

When you need more of a cheatsheet-style reference, try “A Complete Guide to Grid”, by CSS-Tricks.

Box Alignment and Distribution

Before closing this chapter, let’s take a look at some properties that we can use for distributing and spacing boxes. These properties can be used with either Flexbox or Grid. Most examples in this section use Flexbox.

First, some terminology: *justify* versus *align*. As the CSS Box Alignment specification explains, **justify** refers to alignment in the main or inline axis, while **align** refers to alignment in the cross or block axis. Each of the three `justify-*` properties is concerned with the main, inline dimension. Their `align-*` counterparts are concerned with the block dimension.

For layouts that use Grid, the main axis depends on the document’s writing mode. When the writing mode is horizontal, the main or inline axis is horizontal—either left to right or right to left—and the cross axis is vertical. For vertical writing modes, the main axis is vertical, and the cross axis is horizontal.

For layouts that use Flexbox, the value of the `flex-direction` property determines the main axis, and the initial value is `row`. We discuss writing modes in Chapter 6, “Working with Text”. The document’s writing mode determines whether that main axis is horizontal or vertical.

Distributing Items in the Main Axis with `justify-content`

The `justify-content` property indicates to the browser how the contents of a container should be aligned along the main or inline axis. It only has an effect when there’s leftover space available—for example, when no flex items have `flex-grow: 0`, or when the length of an explicit grid is less than that of its container.

The `justify-content` property accepts more than a dozen different values. The table below illustrates each value and its impact on box alignment and distribution, when using a left-to-right writing direction. The dotted line represents the outline of the flex or grid container.

Value	Effect

Value	Effect
`center`	
`left`	
`right`	
`start`	
`end`	
`flex-start`	
`flex-end`	
`space-between`	
`space-around`	

Value	Effect
`space-evenly`	
`stretch` (shown here with Grid)	

The values above can be split into two broad groups: *positional alignment* values and *distributed alignment* values.

Positional alignment values indicate where items should be stacked within a container, and include:

- `center`
- `left`
- `right`
- `start`
- `end`
- `flex-start`
- `flex-end`

Both `justify-content: flex-start` and `justify-content: flex-end` only apply to flex containers. For a similar effect in Grid and multicolumn containers, use `start` or `end`.

Despite appearances, `left`, `start`, and `flex-start` are not the same. Neither are `right`, `end` and `flex-end`. Writing mode and language direction affect box alignment and distribution for `start`/`flex-start` and `end`/`flex-end`. For example, when the `direction` is `rtl` (right to left), `justify-content: flex-start` packs boxes against the right edge of the flex container, as shown in the image below.

When `flex-start` and `flex-end` are used on non-flex containers, they behave like `start` and `end`.

On the other hand, `justify-content: left` and `justify-content: right` always pack boxes to the left or right of the container, respectively. The following image illustrates the effect of `justify-content: left` on a container with a horizontal writing mode and right-to-left language direction.

Distributed alignment values indicate how to divvy up the remaining space in a container. Using `justify-content: stretch`, for example, causes the size of each element to increase evenly to fill the available space, within `max-height`/`max-width` constraints. However, it has no effect on flex items.

`space-around` versus `space-evenly`

Where `space-between` places the first and last items flush against the edges of their container and evenly distributes the space, the difference between `space-around` and `space-evenly` is more subtle:

- `space-around` distributes items evenly within the alignment container, but the size of the space between the first/last item and the edge of the container is half that of the space between each item.
- `space-evenly` distributes space evenly between each item. The size of the space between the first/last item and the edge of the container is the same as the size of the space between each item.

Aligning Items in the Cross Dimension with `align-content`

Where `justify-content` affects items in the main dimension, `align-content` affects them in the cross or block dimension. The `align-content` property accepts most of the same values as `justify-content`, except for `left` and `right`.

Remember that `align-content`, like `justify-content`, affects the distribution of *leftover* space. You'll only notice its impact when the used height of the container is something besides `auto`. The table below illustrates the impact of `align-content` and its values when using a horizontal writing mode and a left-to-right text direction.

Value	Effect

Value	Effect
`center`	
`start/flex-start`	
`end/flex-end`	

Value	Effect
`space-between`	
`space-around`	
`space-evenly`	

Value	Effect
`stretch`	

Because `align-content` distributes space in the cross direction, you only notice its impact when there are multiple rows of content.

We can also combine values for `justify-content` and `align-content` by using the `place-content` shorthand property. It accepts up to two, space-separated values. The first value assigns the `align-content` value; the second is the `justify-content` value. Take, for example, the following CSS:

```
.centerstart {
    place-content: center start;
}
```

This is the equivalent of:

```
.centerstart {
    align-content: center;
    justify-content: start;
}
```

Firefox handles `place-content` a bit differently from Chrome, Edge, and Safari. Beginning with version 60, Firefox supports a single value for `place-content` only when it's a valid value for both `align-content` and `justify-content`:

```
.place-start {
    place-content: start;  /* Valid value for both
properties. Firefox supports it */
}
.place-left {
    place-content: left; /* Not supported. left is an invalid
value for align-content */
}
.place-start-left {
    place-content: start left; /* Supported. Values are valid
and ordered correctly */
}
```

In the preceding code example, the `.place-start` rule works in Firefox, but `.place-left` doesn’t.

Aligning Items with `align-items` and `align-self`

Where `align-content` affects the distribution of rows in the block dimension, `align-items` and `align-self` are concerned with the cross/block alignment of each item within a grid or flex container. The table below shows how `align-items` and its values work with a horizontal writing mode.

Value	Effect
`center`	

Value	Effect
`start/flex-start`	
`end/flex-end`	
`baseline`	

Value	Effect
`first baseline`	
`last baseline`	
`normal`	

Baseline alignment is probably the trickiest concept to understand. When items are aligned along a baseline, they're vertically aligned with the bottom of each letter, without regard for descender height. **Descenders** are the stem parts of lowercase letters such as *q* and *p* that dangle below a line of text. Both `baseline` and `first baseline` align items along the first baseline of a row, while `last baseline` aligns them along the last line of a row.

The `align-items` property applies to the container element and sets the alignment for all of its children. `align-self`, on the other hand, applies to the child elements, and overrides the value of `align-items`. It accepts the same values as `align-items`. Here’s an example:

```
.flex-baseline {
    display: flex;
    align-items: flex-start;
}
.flex-baseline *:nth-child(2) {
    align-self: flex-end;
}
```

Now our second flex item is aligned with the end of the flex container, as shown below.

Choosing `flex` or `grid`

As you develop page or component layouts, you may find yourself wondering when it’s better to use Flexbox and when to use Grid.

Use Grid when you want items to line up along a vertical, a horizontal axis, or both.

Use Flexbox when you want to distribute items and space vertically or horizontally.

Of course, these aren't absolute rules. There's a lot of overlap in terms of what you can do with these modules. Jen Simmons' video "Flexbox vs. CSS Grid — Which is Better?" walks you through some things to consider when choosing between Grid and Flexbox. Rachel Andrew's "Should I use Grid or Flexbox?" is another great resource for understanding both.

In practice, your projects will mix both of these techniques, as well as floats. For instance, you may use Grid to define the overall page layout, while using Flexbox for your navigation menu or search box, and floats to place tables or images.

Conclusion

We've covered a lot of ground in this chapter! Now that you've made it through, you should understand:

- what the CSS box model is, and how it affects page rendering and layout
- how the `float` property affects normal flow, and how to clear floated elements
- what stacking contexts are, and how to use them to create layered effects in CSS
- when and how to use multicolumn, Grid, and flexible box layout

In the next chapter, we'll take a look at the fun topic of creating animations with CSS.

Chapter 6: Working with Text

In this chapter, we'll look at two features of CSS that relate to text: `@font-face`, and writing modes. These features both play a role in **internationalization**—the process of making websites that work with the range of humanity's languages and writing forms.

This chapter won't be a comprehensive look at *every* text-related CSS property. There are far too many properties for that. Instead, we'll focus on some features that are related to internationalization and text display.

Fonts are an integral part of web design and how we display text on the Web, but they can also add bloat. In the first half of this chapter, we'll look at how to go beyond system fonts like Arial, or generic families such as `sans-serif`, with `@font-face`. We'll also discuss strategies for font optimization.

We'll end the chapter with a look at writing modes. **Writing modes** and the `writing-mode` property affect the display of text, particularly for non-Latin scripts such as Arabic and Japanese. Newer CSS layout modules such as Flexbox, Grid, and Box Alignment are writing mode agnostic. Understanding the basics of how writing modes work lays a good foundation for what we'll cover in the next chapter.

Better-looking Text with `@font-face`

In the early days of CSS, font choice was limited to whatever fonts users had installed on their system, and generic font values such as `sans-serif` and `monospace`. Towards the end of the "aughts", however, CSS introduced web fonts and the `@font-face` CSS rule. Web design and typography changed forever.

With `@font-face`, we can use just about any font for our web pages, as long as the font is available in a browser-compatible format.

Check Your Licenses

Not all fonts are licensed for web use, even if it's possible to convert them to a web-friendly format. Do the right thing, and don't risk being on the losing end of a lawsuit. Ensure that you're adhering to the licensing terms of any font you use on your site.

Setting an `@font-face` Rule

Here's what a basic `@font-face` rule set looks like. This is the bare minimum you'll need in order to use a web font:

```
@font-face {
    font-family: 'MyAwesomeFont';
    src:
url('https://example.com/fonts/myawesomefont.woff2');
}
```

The `@font-face` at-keyword tells the browser that we want to use an external font file. The `font-family` line sets a **descriptor**, or nickname, for this font. Don't confuse this with the `font-family` *property*. When used within an `@font-face` rule set, `font-family` sets the value that will be used for CSS font-name matching. The last line defines a font source with the `src` descriptor, which is the location of a font file.

To apply this font to your text, include the descriptor value in the `font` or `font-family` declaration:

```
body {
    font: 16px / 1.5 'MyAwesomeFont', sans-serif;
}
```

The browser will match instances of `MyAwesomeFont` to the source we've specified in our `@font-face` rule set. If `MyAwesomeFont` isn't available, or the browser doesn't support web fonts, it will fall back to the `sans-serif` generic.

Just because we've defined a font for use doesn't mean the browser will load it. Our font also needs to be in a format the browser can parse. For current browsers, that means WOFF2. However, a surprising number of web users don't or can't update their devices. We can accommodate these users by defining multiple font sources.

Using Multiple Font Formats

While the `@font-face` example above takes care of the latest and greatest browsers, older browser versions lack support for the WOFF2 format. They do, however, support its predecessor, WOFF. Let's update our `@font-face` rule to provide a WOFF alternative:

```
@font-face {
    font-family: 'MyAwesomeFont';
    src: url('http://example.com/fonts/myawesomefont.woff2')
format('woff2'),
         url('http://example.com/fonts/myawesomefont.woff')
format('woff');
}
```

The `src` descriptor takes the format `<url> format()`, where `<url>` is the location of a font resource, and `format()` is a format hint. We can provide multiple `src` options by separating them with a comma. Using `format()` helps the browser select a suitable format from the ones provided. Its argument should be one of `woff`, `woff2`, `truetype`, `opentype`, or `embedded-opentype`. In this example, browsers that don't support WOFF2 will download the WOFF-formatted font file instead.

More on Font Formats

The CSS Fonts Module Level 4 specification includes a more complete list of formats and their corresponding font hint values.

You may see examples of `@font-face` rules that include EOT, SVG, TrueType, or OpenType font formats. You can safely exclude these formats. EOT font support is limited to ancient versions of Internet Explorer 9 and below. Most browsers have removed support for SVG fonts, or never implemented it to begin with. TrueType and OpenType enjoy wide browser support, but WOFF2 file sizes are much smaller. The only reason to use either format is if the font in question isn't available as a WOFF2-formatted or WOFF-formatted file.

Fonts and Origins

Web fonts are subject to the **same-origin policy**. Under this policy, a browser loads a resource only if it shares the same “origin” as the requesting document. An **origin** is the combination of a document’s scheme or protocol, host name, and port number.

In other words, if your web page is served from `https://example.com` and your fonts are served from `https://cdn.example.com`, they won't load. To get around this restriction, you'll need to enable "cross-origin resource sharing".

Cross-origin resource sharing, or CORS, is a system of headers that tell the browser whether or not a document from a requesting origin has permission to use a requested asset from another. A full discussion of CORS is well beyond the scope of this book, but I'll try my best to explain it.

More about CORS

If you'd like to learn more about CORS, MDN Web Docs has what may be the Web's most thorough explanation of cross-origin resource sharing and its collection of response headers.

When an HTML or CSS document links to external assets, the browser first checks whether those assets share the same origin as the requesting script or file. If so, it loads the asset.

If the requesting document *doesn't* share the same origin as the requested resource, the browser makes a "preflight request" for the resource. A **preflight request** asks the external server: "Does https://example.com have permission to load GreatGroteskWebFont.woff2?" If the server response includes the `Access-Control-Allow-Origin` response header and https://example.com as its value, the browser follows up with a `GET` request for the font file and loads it. If the response doesn't include that, the browser won't make the `GET` request and the font won't be loaded.

To enable CORS, you'll need to add an `Access-Control-Allow-Origin` response header to your font URLs. This header grants permission to the requesting document's origin. Origins must not have a trailing slash. Here's an example:

```
Access-Control-Allow-Origin: https://example.com
```

Adding headers requires access to your server or content delivery network configuration. If you don't have such access, or don't feel comfortable managing headers, you have two options:

- serve your font files from the same origin as your document
- use a hosted web font service such as Google Fonts (free), Adobe Fonts or Fontspring

Hosted services implement their own cross-origin headers so that you don't have to worry about it.

Using Multiple Font Weights and Styles

A **font** is actually a collection of typefaces or faces. A **face** is a single weight, width, and style of a font. EB Garamond is a font. EB Garamond Regular and EB Garamond Bold Italic are faces. Most people use the terms interchangeably, but differentiating between the two is helpful here.

When incorporating a web font into your site's design, you may also want to incorporate its stylistic variants for **bolded** or *italicized* text. We can do this using the `font-weight` and `font-style` descriptors. These descriptors tell the browser which face (and corresponding file) to match with a particular weight or style:

```
@font-face {
    font-family: 'EB Garamond Regular';
    src: url('EB-Garamond-Regular.woff2') format('woff2'),
         url('EB-Garamond-Regular.woff') format('woff');
    /*
     The next line is optional, since this is the initial
value.
     It's the equivalent of font-weight: 400
    */
    font-weight: normal;
}
@font-face {
    font-family: 'EB Garamond Italic';
    src: url('EB-Garamond-Italic.woff2') format('woff2'),
         url('EB-Garamond-Italic.woff') format('woff');
    font-style: italic;
}
@font-face {
    font-family: 'EB Garamond Bold';
    src: url('EB-Garamond-Bold.woff2') format('woff2'),
         url('EB-Garamond-Bold.woff') format('woff');
    font-weight: bold; /* The equivalent of font-weight: 700
*/
}
@font-face {
    font-family: 'EB Garamond Bold Italic';
    src: url('EB-Garamond-Bold-Italic.woff2')
format('woff2'),
         url('EB-Garamond-Bold-Italic.woff') format('woff');
    font-weight: bold;
    font-style: italic;
}
```

In the example above, we've matched faces from the EB Garamond font family with an appropriate style and weight. Here, too, `font-weight` and `font-style` are descriptors that tell the browser to download an additional font file to display weight and style variants, should the page use bold and/or italic text.

Browsers synthesize bold or italic text from the primary font when an appropriate weight or style isn't available. However, this may lead to less readable or less attractive text. Compare the synthetic italic text (using EB Garamond, top) to the italic version (EB Garamond Italic) of this font in the image below.

That said, pretty isn't always fast. Using multiple faces increases the amount of data that must be sent to the browser. As with most aspects of web development, you'll need to make trade-offs between style and performance.

Variable Fonts

Variable fonts—more accurately called “OpenType font variations”—are an extension of the OpenType specification. Variable fonts are single font files with support for additional features that can be managed using CSS. You can, for example, control the width of each glyph or the degree of tilt used for oblique text. If the font file supports it, you can even adjust the width of serifs, as with the Foreday font by DSType Foundry.

With variable fonts, a single font file behaves like multiple font faces. Variable fonts make the previous section of this chapter moot.

Below is pictured the letter A, from the open-source variable font Jost, in varying weights.

OpenType

OpenType is a file format that enables cross-platform font support by combining support for TrueType and PostScript font data in a single file.

To use variable fonts in your project, you’ll first need a font file that supports variable font features. Axis-Praxis (pictured below) and v-fonts are two sources for discovering and experimenting with variable fonts.

Both sites include specimens and controls to play with the variable features that each font supports. Most hosted font services have small but growing selections.

Although most major browsers have implemented support for variable fonts, the number of such fonts is still fairly small in comparison to the number of traditional fonts.

Incorporating Variable Fonts

To incorporate a variable font, we’ll need to add another source and format hint to our CSS:

```
@font-face {
    font-family: 'FontFamilyName';
    src: url('FontFamilyName-Variable.woff2') format('woff2-
variations'),
         url('FontFamilyName.woff2') format('woff2'),
         url('FontFamilyName.woff') format('woff');
}
```

If the browser supports variable fonts, it will download `FontFamilyName-Variable.woff2`. If it doesn’t, it will download a file that it’s capable of parsing. This syntax above works today in every browser that supports variable fonts.

In April 2018, the CSS Working Group decided to change the syntax of format hints. As Richard Rutter explains in his article “Upcoming changes to the CSS you need for variable fonts”:

> the list of potential format strings is growing fast and could in future contain other kinds of font features, such as colour fonts. With an eye on the future, the CSS Working Group recently resolved to change the syntax of the `format()` hint [to] separate out the font features from the file type.

Format hints will soon use a `format('format_name', supports feature_name)` syntax, which is shown below:

```
@font-face {
    font-family: 'FontFamilyName';
    /* New CSS syntax. Not yet widely implemented. */
    src: url('FontFamilyName-Variable.woff2') format('woff2'
supports variations);
}
```

A future-proof `@font-face` declaration with support for variable fonts might look like this:

```
@font-face {
    font-family: 'FontFamilyName';
    src: url('FontFamilyName-Variable.woff2') format('woff2-
variations'),
         url('FontFamilyName.woff2') format('woff2'),
         url('FontFamilyName.woff') format('woff');
    /* New CSS syntax. Not yet widely implemented. */
    src: url('FontFamilyName-Variable.woff2') format('woff2'
supports variations);
}
```

Why two `src` declarations? Remember: browsers ignore CSS rules they can't understand, and the last rule wins. Adding the `format('woff2' supports variations)` hint to our existing `src` declaration would cause browsers to ignore the *entire* rule. By using two `src` declarations, we guarantee that the browser will use one of them. The first `src` declaration will be used by browsers that don't support the newer format hint syntax. Browsers that do support it will override the first declaration with the second.

Specifying Font Weight When Using Variable Fonts

As mentioned in the previous section, the `font-weight` descriptor lets us tell the browser which font-face file should be matched to a particular weight. Variable fonts, however, can support a range of font weights within a single file.

Instead of using a `src` declaration for each font-face weight, CSS4 has modified the behavior of the `font-weight` descriptor to accept a value range:

```
@font-face {
    font-family: 'FontFamilyName';
    src: url('FontFamilyName-Variable.woff2') format('woff2-
variations'),
    src: url('FontFamilyName-Variable.woff2') format('woff2'
supports variations);
    font-weight: 1 1000; /* Use this file for values within
this font range. */
}
```

Adding a `font-weight` range instructs the browser to use the same file for every `font-weight` value that falls within the range. This includes `font-weight: bold`, which is the equivalent of `font-weight: 700`, and `font-weight: normal`, which is the equivalent of `font-weight: 400`.

Historically, `font-weight` accepted numeric weight values ranging from 100–900, in increments of 100. As of CSS4—and with the advent of variable fonts—we no longer have those restrictions. For example, `font-weight: 227` is now a valid, supported `font-weight` value. Any number greater than or equal to 1 and less than or equal to 1000 is a valid `font-weight` value. Fractional weights, such as `font-weight: 200.5` are also valid.

Lower-level Font Control with `font-variation-settings`

CSS4 has also introduced a `font-variation-settings` property for finer-grained control of font features. It lets us manipulate fonts along one of five axes, using one of the registered axis tags defined in the OpenType specification.

Axis tag	Name	Notes
`ital`	italic	Typically a float value between 0 and 1, although some fonts may exceed those bounds
`opsz`	optical size	Adjusts the shape of glyphs according to the target point size. For example, `"opsz" 72` adjusts the shape of each glyph to match that of 72pt type, regardless of the value of `font-size`. Requires the font to support optical sizing
`slnt`	slant	The degree of slant for oblique text
`wdth`	width	Works similarly to the `font-stretch` property
`wght`	weight	Works similarly to the `font-weight` property

We could, for example, use `wght` and `ital` to set the weight and amount of italicization of an `h1` selector:

```
h1 {
    font-variation-settings: "wght" 900, "ital" .9;
}
```

Keep in mind that not all variable fonts support all of these axis tags. Some fonts, such as Amstelvar, support additional settings such as `YTSE`, which controls serif height. On the left of the image below is the Latin letter A with the default serifs. On the right is the same letter with `"YTSE" 48` as part of its font-variation-settings declaration.

Which values we can modify, and the boundaries of those values, depends on the font file itself. You'll need to consult the documentation for each font, if available. Because of this hurdle, your best bet is to use the `font-weight`, `font-style`, `font-optical-sizing` and `font-stretch` properties.

Shaping Loading Behavior with `font-display`

Early browser implementations of `@font-face` differed pretty significantly in how they handled text while waiting for the web font to download. Firefox, for example, drew text using the fallback font face, then swapped the font face once downloaded—resulting in the dreaded "flash of unstyled text", or FOUT.

Safari's implementation, on the other hand, drew *invisible* text, then swapped it for the visible font face once it finished loading. Users would see large blocks of empty space until the font face was available to the browser. Developers dubbed this the “flash of invisible text”, or FOIT.

The `font-display` descriptor came about largely as a way to let developers choose which behavior they prefer. It has five possible values. Excluding `auto`, each of these values changes the duration of the *block* and *swap* periods of the *font display timeline*.

- `auto`: *initial value*. The “Browser’s Choice” option. It uses whatever policy the browser prefers.
- `block`: draws invisible text until the font loads, then swaps the font face. (Approximately a three-second block period, and an infinite swap period.)
- `swap`: draws text using the fallback font until the font face loads, then immediately swaps the font face. (Roughly a 100ms block period, and an infinite swap period.)
- `fallback`: draws text using the fallback font, while waiting for the font face to load. If the font face takes too much time to load, it continues to use the fallback. Otherwise, it swaps the font face once loading completes. (Approximately a 100ms block period, and about a three-second swap period.)
- `optional`: uses the font only if it can be downloaded immediately. Otherwise, it will use the fallback text. The browser can decide whether or not to download the font and swap it, or continue to use the fallback text. Optional won’t cause layout shifts. The browser may never swap the font face.

When a browser begins downloading a font face, it enters the **block period**. During the block period, the browser draws text using an invisible fallback font. If the font face finishes downloading during the block period, the browser will use it.

Next comes the **swap period**. During the swap period, if the font face hasn’t loaded, the browser draws the text using the fallback font. It then swaps the font face when loading completes.

If the font hasn’t loaded by the end of the swap period, the browser enters the **failure period**. The browser uses the fallback font.

Understanding `auto`

When `font-display` is `auto`, you’re relying on the browser’s default font-face handling. For most recent browsers, that behavior resembles `block`. There’s a short period of about three seconds during which the user won’t see any text. The image below shows what this looks like in Firefox using a simulated 2G connection.

In most browsers, `font-display: auto` means the text will be invisible until the font face loads. Text that doesn't use the web font is immediately drawn.

If the font face hasn't loaded by the end of this timeout period, the browser draws text using the fallback font. When the font fully downloads, as shown below, the browser swaps the text.

On very slow connections, font faces can take several seconds to download. Until then, your site visitors won't be able to read any text that uses a font face.

Instead, use `fallback` or `optional`, particularly for larger blocks of text. They're both particularly well suited to serving users with slow internet connections. For smaller blocks of text, such as headlines, `swap` also works well. All three values keep your text visible while the font face loads. *Something* is happening. As a result, the page loading time seems faster to users than seeing no text at all, as happens with `block` or `auto`.

Optimizing Fonts with Subsetting and `unicode-range`

Languages are written using **scripts**, or groups of symbols or *characters* used to express a language. English, Spanish, and Norwegian use Latin script. Farsi uses a variant of Arabic script. Hindi and Rajasthani use Devanagari.

Scripts are comprised of **characters**. In computing, each character in a script is represented by a hexadecimal numeric value, also known a character code. Mapping codes to characters is called **character encoding**.

There are multiple systems of character encoding available in computing. On the Web, however, you should use Unicode. **Unicode** is a system that maps characters from multiple scripts to unique hexadecimal numeric values. The Latin letter A, for example, is represented by the number `0041`, while the Armenian character Ֆ is represented by the number `0556`. Depending on the context, these numbers may be prefixed by `U+` or a `\u` when used with CSS.

More on Unicode

I've left out a lot of background about the history of character encodings and how Unicode came to be. This is, after all, a book about CSS, not character encoding. If you'd like to learn more about the whys and what-fors of Unicode, visit the Unicode Consortium's website—unicode.org.

Stick with me here—I promise there's a point to all of this background information. Fonts map character codes to "glyphs". A **glyph** is the actual shape that represents a character. A lowercase letter "a", for example, can be represented by glyphs from several different fonts, as shown below. From left to right are glyphs representing the letter "a" from the Bodoni 72 Bold, Juju Outline, Junction Bold, and Futura Bold fonts.

Now, font files contain the *entire* character set or glyph set available for that font. That includes obscure punctuation, characters from other scripts, and symbols such as © and ™. There's a very good chance you won't use all of those characters on your site. But if your web font contains them, you're still sending those bytes to your users.

The good news is that we can manage this using the `unicode-range` descriptor and a process known as "subsetting". **Subsetting** is the process of breaking a font into multiple files, each containing a smaller collection—a subset—of glyphs.

Browsers that fully support `unicode-range`—and this includes most versions released since 2016—only download a font face when characters in the document fall within its corresponding unicode range.

Most web font services automatically manage subsetting and unicode ranges. For self-hosted fonts, there's FontTools.

Subsetting Self-hosted Fonts with FontTools

Consider a multi-script font such as Gaegu (available with an SIL Open Font License), which includes characters from Latin and Hangul scripts. We might split this font into two files: `gaegu-latin.woff2` and `gaegu-hangul.woff2`. We can then use the `unicode-range` descriptor to assign each file to a different Unicode range:

```
@font-face {
    font-family: 'Gaegu';
    src: url('https://example.com/fonts/gaegu-latin.woff2')
format('woff2');
    unicode-range: U+000-5FF; /* Latin glyph range */
}
@font-face {
    font-family: 'Gaegu';
    src: url('https://example.com/fonts/gaegu-hangul.woff2')
format('woff2');
    unicode-range: U+1100-11FF; /* Hangul glyph range
(partial) */
}
```

Licensing Requirements

The SIL Open Font License (OFL) requires that variations of a font file be completely renamed. This may include file format conversions, such as TrueType to WOFF. It probably includes subsetting. For the sake of clarity, I've retained the Gaegu font name for both files. In a public-facing project, you may wish to use a different name.

For self-hosted fonts, we'll need to create the subset version of the font ourselves using FontTools. FontTools is a Python library for manipulating fonts. While this does require us to have Python installed, we don't need to know how to program with Python.

To install FontTools, we'll need to use `pip`, the Python package manager. In a terminal window or at the Windows command-line prompt, type the following:

```
pip install fonttools[woff]
```

This installs `fonttools` and two additional libraries that we'll need for creating WOFF and WOFF2 files: brotli and zopfli.

Mac Users

Although macOS comes with Python installed, it may not include pip. Your best bet for installing pip while creating the fewest headaches is to install the latest Python version using Homebrew: `brew install python`. Homebrew will install pip as part of the Python installation process. Use `pip3` instead of `pip` to run commands.

This command installs a few different subpackages, including ones for font format conversion (`ttx`) and merging fonts (`pyftmerge`). We're interested in `pyftsubset`, which can create subsets from OpenType, TrueType, and WOFF font files.

Let's use `pyftsubset` to create a Latin-only version of the Gaegu font:

```
pyftsubset ~/Library/fonts/Gaegu-Regular.ttf --
unicodes=U+000-5FF
```

At a minimum, `pyftsubset` needs an input file and one or more glyph identifiers or a Unicode range as arguments. In the example above, we've used the `--unicodes` flag to specify the range of characters to include. Again, both of these arguments are required.

To create a WOFF2 web font, we need to pass an additional `--flavor` flag:

```
pyftsubset Gaegu-Regular.ttf --unicodes=U+000-5FF --
flavor="woff2"
```

For OFL-licensed fonts, we should also rename our font file and remove name information from the font tables. To do that, we need to pass two more flags: `--output-file` flag, and `--name-IDs`:

```
pyftsubset ~/Library/fonts/Gaegu-Regular.ttf --
unicodes=U+000-5FF --flavor="woff2"  --output-
file='myproject/subsetfont-latin.woff2' --name-IDs=''
```

Passing an empty string as the argument for `--name-IDs` strips all existing name information from the font file. Now we can use our subset OFL-licensed font in our project.

`pyftsubset` is more feature-rich than we've discussed here. We can, for example, exclude ligatures and vertical typesetting data. To see a full list of commands and how they work, use `pyftsubset --help`.

Writing Modes

Writing modes are one of the more esoteric areas of CSS. However, they're important to understand for developers who work with languages that are written from right to left (such as Hebrew and Arabic), languages that can be written vertically (such as Mongolian), or languages that can be written using both (such as Japanese, Chinese, or Korean). In this section, we'll discuss:

- what writing modes are, and how browsers determine the writing mode of a document
- CSS properties that affect the writing mode

Although the primary purpose of writing modes is internationalization, you can also use them in creative ways. For instance, you might use vertical or sideways headings to mark sections of text, as shown below.

Let's dig in!

What Is a Writing Mode?

A document's **writing mode** is the combination of its *inline base direction* and its *block flow direction*. The **inline base direction**, or inline direction, is the primary direction in which lines of text are ordered. **Block flow** refers to the direction in which block-level boxes stack.

Languages such as English, French and Hindi are typically written and read from left to right. Lines of text start at the left edge of the container and continue horizontally, ending at the right edge of the container. Blocks of text —such as headings and paragraphs—stack vertically from the top of the screen to the bottom. These languages use a *horizontal writing mode*.

Languages such as Chinese and Korean, on the other hand, can also use a *vertical writing mode*. In a vertical writing mode, lines of text begin at the top of the container and continue to the bottom. Blocks of text stack horizontally.

Technically, what we're discussing here are **scripts**, or the groups of symbols used to express a language. Scripts can be used to write multiple languages. For example, Spanish, English, and Norwegian all use Latin script. The inverse is also true: some languages can be written using more than one script. As the World Wide Web Consortium explains, Azeri can be written using Latin, Cyrillic, or Arabic scripts. Scripts have a writing direction. Languages use the direction of the script in which they're written. In other words, when written using Latin or Cyrillic scripts, Azeri is read and written from left to right. When written using Arabic, it's read from right to left. For the sake of precision, we'll use "script" instead of "language" for the rest of this chapter.

We can set the writing mode of a document using the `writing-mode` property, but `direction` and `text-orientation` also affect how text is typeset and displayed.

Setting the Direction of Text with the `direction` Property

With the `direction` property, we can specify the direction of text—either `rtl` (right to left) or `ltr` (left to right). Its initial value is `ltr`. When the value of `direction` is `ltr`, text lines start at the left edge of the container and end at the right edge, as illustrated below.

When the value is `rtl`—as appropriate for Arabic and Hebrew scripts—text lines start at the right edge and end at the left, as shown below.

Using the HTML `dir` Attribute Is Best

Because browsers can strip CSS from HTML documents—say, when using Reader mode—the Writing Modes specification advises web developers to avoid using the `direction` property with HTML. Instead, use the HTML `dir` attribute to set text direction, and the `<bdo>` or `<bdi>` elements to override the direction for smaller bits of inline content:

```
<!DOCTYPE html>
<html lang="ar" dir="rtl">
    <head>
        <title>باستخدام السمة dir</title>
    </head>
    <body>
        <p>قفز الثعلب البني السريع على الكلب الكسول.<bdo
dir="ltr" lang="en">SitePoint.com</bdo>
        </p>
    </body>
</html>
```

Using markup ensures that user agents will properly display the document, even if its CSS has been stripped away. For markup languages that lack these features (such as SVG), the `direction` CSS property is appropriate.

Setting Block Flow Direction with the `writing-mode` Property

The `writing-mode` property determines how block-level boxes and table rows are ordered on the screen or page. It also determines whether lines of text within those boxes are arranged horizontally or vertically. Its initial value is `horizontal-tb`, which is a shorthand for “horizontal, top to bottom”.

If no other CSS is applied to a document, block boxes will flow from top to bottom. Lines of text within those boxes will be arranged horizontally, as was shown in the two images in the previous section. For languages that use Latin, Arabic, Hebrew, or Devanagari script, this is always appropriate.

Humanity, of course, is varied and complicated. A top-to-bottom block flow doesn’t work for every language. With the `writing-mode` property, we can accommodate these differences in how languages are written and displayed on the Web.

Writing Modes

For a more comprehensive look at writing modes and scripts, consult the "Language enablement" documentation of the World Wide Web Consortium's Internationalization Activity group.

In addition to `horizontal-tb`, the `writing-mode` property accepts four other values:

- `vertical-rl`
- `vertical-lr`
- `sideways-rl`
- `sideways-lr`

When the value of `writing-mode` is `vertical-rl`, text is arranged vertically, and the block boxes are ordered from right to left.

When the value of `writing-mode` is `vertical-lr`, text is arranged vertically, and blocks progress from left to right.

The following image features an example of Japanese text with a `vertical-rl` writing mode.

The text begins from the right edge of the image. Our Japanese glyphs are translated and rendered vertically. However, non-Japanese glyphs such as numerals are rotated 90 degrees.

Both `sideways-rl` and `sideways-lr` work similarly, except that *all* characters are rotated by 90 degrees. With `writing-mode: sideways-rl`, text is displayed vertically, from top to bottom, and all glyphs are rotated clockwise by 90 degrees, as illustrated below.

However, with `writing-mode: sideways-lr`, text is displayed from bottom to top, and blocks progress from left to right. Glyphs are instead rotated 90 degrees *counter*-clockwise.

Support for `sideways-rl` and `sideways-lr` is currently limited to Firefox 43 and above. Consider these values to be experimental for the time being. Their behavior may change, or support may be dropped from browsers entirely.

Note that the orientation of `<img>` and `<video>` elements isn't affected by `writing-mode`, as pictured below.

Managing Typesetting with `text-orientation`

Writing systems, and the fonts that use them, have one or more *native orientations*. Latin-, Arabic- and Devangari-based scripts are always written horizontally, and therefore have a horizontal native orientation. Mongolian script is always written vertically and has a vertical native orientation. Chinese, Japanese, and Korean can be written vertically *or* horizontally, which is known as *bidirectional orientation*. Native orientation helps determine how glyphs are displayed within a document.

Most contemporary fonts assign a horizontal orientation for every glyph that's used when glyphs are presented horizontally. But as we've mentioned, some scripts can be written vertically. Glyphs within those scripts are *transformed* when text is presented vertically.

Transformed glyphs may be *translated*, or shifted, so that they’re arranged vertically, as pictured above on the left. Or they may be rotated, so they’re typeset sideways, as illustrated above on the right. Some scripts have a native bidirectional orientation. Their font files usually contain vertical typesetting information that's used when glyphs are presented vertically.

It's not uncommon, however, to use characters from horizontally oriented scripts in a vertically oriented document. Think numerals such as 0, 2, or 4 within a paragraph of Japanese text. We can shape how these glyphs are typeset using the `text-orientation` property.

The `text-orientation` property accepts one of three values, each of which is described as follows:

- `mixed`: glyphs from horizontally oriented scripts are rendered sideways, or rotated by 90 degrees, but vertically oriented glyphs will be rendered vertically (as pictured below, left).
- `upright`: glyphs from horizontally oriented scripts are rendered in horizontal orientation. Glyphs from vertically oriented scripts are rendered in their intrinsic, vertical orientation (as pictured below, center).
- `sideways`: all text is rendered sideways, as if in a horizontal writing mode, and rotated 90 degrees (as pictured below, right).

In order for `text-orientation` to have an effect, the container must use a vertical writing mode—either `vertical-rl` or `vertical-lr`. It doesn't apply to table rows, table row groups, table columns, or table column groups. You can, however, use it with tables, table cells, and table headers.

Writing Mode and Alignment

Text alignment and box alignment are also affected by writing mode. Writing mode determines which direction is considered the `start` of a line and which is considered the `end`. In the image below, for example, our table has a `direction` value of `rtl` (right to left). As a result, `text-align: start` aligns the text of each cell along its right edge.

However, in the image below, the `direction` is `ltr` (left to right). In this case, `text-align: start` causes the text of each cell to be aligned with its left edge.

Similarly, `justify-content: flex-start` aligns items with the left edge of their container when the value of `writing-mode` is `horizontal-tb`, and the `direction: ltr`, as seen below.

However, when the value of `direction` is `rtl` (or the `dir` attribute value is `rtl`), `justify-content: flex-start` aligns items with the right edge, as shown below.

Conclusion

In this chapter, we've discussed how text can be manipulated and enhanced with CSS. You should now have a sense of how to:

- implement web fonts and optimize them for a better user experience
- support sites that use non-Latin scripts and multiple languages

You should also have a sense of how writing modes work.

In the next chapter, we'll look at how to use CSS to add a sense of fun and energy to our web pages with transitions and animations.

Chapter 7: Transitions and Animations

Now let's look at how to add some whimsy, delight, and polish to our documents and applications. In this chapter, we'll cover CSS transitions and animations. Transitions and animations can clarify the effect of an action. A menu that slides into view, for example, is less abrupt and jarring than one that appears suddenly after a button is clicked. Transitions and animations can also draw attention to a page change or problem. You might, for instance, transition the border color of a form field to highlight that its value is invalid.

Background

"Animation for Attention and Comprehension", from the Nielsen Norman Group, is a nice backgrounder on how animation and transitions can enhance usability.

This is probably a good time to explain how animations and transitions differ. With a **transition**, you define start and end states, and the browser fills in the states in between. With an **animation**, on the other hand, you can define those in-between states to control how the animation progresses.

CSS Transitions

CSS transitions are a CSS-based way—as opposed to a JavaScript way—to update the value of a CSS property over a specified duration. Given a start value and an end value, the browser will interpolate in-between values over the course of the transition. They're great for simple effects where you don't mind giving up control over how the animation progresses.

In my own work, I sometimes use transitions for `:hover` states. I also use them when revealing or concealing content, such as showing an off-screen menu. You *could* create animations for such effects, but animations are generally more verbose, as you'll see later in the chapter.

We can't transition every property. We can only use transitions with properties that accept *interpolatable* values. **Interpolation** is a method of calculating values that fall within a range. These values are typically numeric unit values such as lengths, percentages, or colors. That means we can't transition between `visibility: visible` and `visibility: hidden`, or `display: block` and `display: none`. Nor can we transition to or from `auto` values.

Animatable CSS Properties

The CSS Transitions specification includes a list of animatable CSS properties and values.

Creating Your First Transition

In this example, we'll make our link color transition from blue to pink when users move their mouse over it, and back to blue when users moves their mouse off it.

Here's our bare-bones HTML:

```
<!DOCTYPE html>
    <html lang="en-US">
    <head>
        <link rel="stylesheet" href="style.css">
    </head>
    <body>
        <p>Mouse over <a href="https://sitepoint.com/">this
link</a>to see the transition effect.</p>
    </body>
</html>
```

This gives us the page shown below.

Now let’s add the following CSS to our `style.css`:

```
a {
    transition: 1s;
}
a:link {
    color: #309;
}
a:hover {
    color: #f0c;
}
```

This is the bare minimum CSS required for a transition to work: a start value (`color: #309`), an end value (`color: #f0c`), and a transition duration (`transition: 1s;`). When you mouse over the link, you'll see a gradual transition from blue to hot pink, as illustrated below.

Transitions need to be triggered by some kind of event. Often, this is a user interaction. We might transition between colors when entering and leaving a `:hover` state, as we've done here. But we can also trigger a transition by adding or removing a class name using JavaScript. In the following example, we modify an element's `classList` attribute to do just that:

```
const btn = document.querySelector( 'button' );

const clickHandler = () => {
    document.body.classList.toggle( 'change' );
}

btn.addEventListener( 'click', clickHandler );
```

In the code, we've first defined a variable named `btn`. If you're unfamiliar with programming, a variable is simply a bucket of sorts that holds a value. We can then use the variable anywhere we need that value.

The value of `btn` is our button element, as returned by `document.querySelector('button')`. The `document.querySelector()` method is defined by the Selectors API specification. It accepts any CSS selector as its argument, and returns the *first* item that matches. It's a way to select elements using JavaScript.

Next, we've defined a `clickHandler` function. This will be the event listener for our `click` event. Finally, we've added the event listener to `btn` using `addEventListener`. The `addEventListener` method is part of the Document Object Model. It allows us to define a function that's invoked when a particular event occurs.

The magic happens within the `clickHandler` function. Here we've used the `Element.classList.toggle()` method to add or remove the `change` class from the `<body>` element (`document.body`). This is what triggers our transition. The `classList` property is part of the Document Object Model API. It provides a handful of methods for manipulating the class names of an element.

Understanding JavaScript

If any of that went over your head, don't worry. Pick up Darren Jones' *JavaScript: Novice to Ninja, 2nd Edition* if you want to get up to speed with JavaScript.

Now let's look at our CSS. It's only a few lines long:

```
body {
    background: #fcf;
    transition: 5s;
}
.change {
    background: #0cf;
}
```

Here, we've defined a starting background color for our `<body>` element, and a transition. We've also defined a `.change` class, which has a different value for `background`. When our event handler runs, it adds the `change` class to our `<body>` element. This triggers a transition from the original background color to the one defined in the `.change` declaration block, as shown below.

If you want a transition to work in both directions—for example, when the class is both added and removed—you should add it to whichever declaration block is your start state. We've done that here by including the `transition` property in the `body` declaration block. If we moved the transition to the `change` class, our transition would only work when `change` was added to our `<body>` element, but not when it was removed.

So far, we've used the `transition` shorthand property. It's a condensed way of specifying four "longhand" properties, which are listed in the table below.

Property	Description	Initial value
`transition-duration`	How long the transition should last	`0s` (no transition)

Property	Description	Initial value
`transition-property`	Which property to transition	`all` (all animatable properties)
`transition-timing-function`	How to calculate the values between the start and end values	`ease`
`transition-delay`	How long the browser should wait between changing the property and starting the transition	`0s` (no delay)

Each longhand property has an *initial* value. The browser uses the initial value for the property, unless you explicitly set its value. For example, the initial value of `transition-property` is `all` (all properties), and the initial value of `transition-timing-function` is `ease`. When we set a transition duration—such as `transition: 1s`—the values for `transition-property` and `transition-timing-function` are implied. This is why we can get away with setting the `transition` property and nothing else.

Using the `transition` Property

As we've already seen in the previous examples, time units are one acceptable value for the `transition` property. The [CSS Values and Units Module Level 3](#) specification defines two kinds of time units for use with transitions and animations: `s` for seconds, and `ms` for milliseconds. We can also collapse values for `transition-timing-function`, `transition-delay`, and `transition-property` into this shorthand `transition` property:

```
body {
    background: red;
    transition: background 500ms linear 1s;
}
```

Here, we've told the browser to transition the `background` property. The duration will last 500 milliseconds (which we could also write as `.5s`). It will use the `linear` timing function (discussed later in this chapter), and the start of the transition will be delayed by one second. It's a compact version of the following CSS:

```
body {
    background: red;
    transition-property: background;
    transition-duration: 500ms;
    transition-timing-function: linear;
    transition-delay: 1s;
}
```

Order matters somewhat when using the `transition` shorthand property. The first value that can be interpreted as a time will become the transition duration no matter where it sits in the value string. The second time value will determine the transition delay. In other words, we could reorder the values in our transition property like so:

```
body {
    background: red;
    transition: 500ms 1s background linear;
}
```

Here, our transition duration will be `500ms` with a one-second delay.

Using the `transition` property is the most concise way to define a transition. However, there may be cases in which you want to define a global transition effect (for example, `transition: 500ms ease`) in one part of your CSS, and limit it to specific CSS properties (for example, `transition-property: color`) in another. This is where the longhand properties are useful.

Transition Durations and Delays

The `transition-duration` property sets the duration of the transition, or how long it takes to complete. The `transition-delay` property determines how much time should elapse before the transition begins. Both properties accept time units as a value. These can be seconds or milliseconds: `1s`, `2.5s`, and `200ms` are all valid values.

Both `transition-duration` and `transition-delay` have an initial value of `0s`, or zero seconds. For `transition-duration`, this means there will be no gradual transition between the start and end states. For `transition-delay`, this means the transition will occur immediately.

With `transition-duration`, you must use values greater than zero, such as `.5s` or `2500ms`. Negative values will be treated like a value of `0s`, and the transition will fail to execute, as illustrated below.

However, negative values are valid for `transition-delay`. Positive `transition-delay` values shift the start of the animation by the specified amount of time. Negative values, however, offset the beginning of the transition, as seen above. Using `transition-duration: 2s; transition-delay: -1s` will cause the transition to jump one second into the play cycle before continuing. Using a negative `transition-delay` value can create a snappier transition experience by shortening its perceived duration.

Timing Functions

We can also shape transition effects using the `transition-timing-function` property. Timing functions are formulas of sorts that determine how the in-between values of a transition are calculated. Which timing function you use will depend on what kind of transition effect you'd like to achieve: a stepped transition or a smooth, gradual one.

Stepped Transitions

With **stepped transitions**, the play cycle is divided into intervals of equal value and duration. We can set how many intervals a transition should have using the `steps` timing function.

Let's revisit our background color example from earlier in this chapter. Instead of using the default `ease` timing function, we'll instead use the `steps` function to create a five-step transition. Our revised CSS looks like this:

```
body {
    background: #f0f;
    transition: 5s steps(5);
}
.change {
    background: #0cf;
}
```

Rather than a smooth, gradual shift between colors, this transition cycles through five distinct color states.

There are also two keywords we can use to create stepped animations: `step-start` and `step-end`. These are equivalent to `steps(1, start)` and `steps(1, end)`. With these keywords (or their `step` function equivalents), you'll see one transition step between the starting and ending values.

Smooth Transitions

Smooth transitions use the `cubic-bezier` function to interpolate values. Understanding *how* this function works involves a bit of math, along with some handwaving and magic. Read Pomax's "A Primer on Bézier Curves" if you're interested in the intimate details. What follows is a simplified explanation.

The cubic Bézier function is based on the cubic Bézier curve. A **Bézier curve** consists of a start point and an end point, and one or more control points that affect the shape of the curve. A *cubic* Bézier curve always has two of these control points, which can be seen below. Curves are drawn from the start point to the end point, towards the control points.

The arguments passed to the `cubic-bezier` function represent the coordinates of those control points: *x1*, *y1*, *x2*, *y2*. But there's a constraint on these points: X values (the first and third parameters) must fall between `0` and `1`. Y values (the second and fourth parameters) can exceed this range in either direction. In other words, `cubic-bezier(0, 1.02, 1, 0)` and `cubic-bezier(0, 1.08, .98, -0.58)` are valid values, but `cubic-bezier(2, 1.02, -1, 0)` is not.

Experimenting with Bézier Curves

Lea Verou's cubic-bezier.com is a great tool for experimenting with the `cubic-bezier` function. Easing Function Cheat Sheet also offers several ready-made `cubic-bezier` snippets for easing functions that are not defined by the specification.

Graphs are the best way to illustrate how `cubic-bezier` works. The X-axis is a function of the transition's duration, as can be seen in the image below, which shows a graph of `cubic-bezier(0.42, 0, 1, 1)`. The Y-axis is a function of the value of the property that's being transitioned. The outputs for these function determine the values of the property at a particular point in the transition. Changes in the graph match the changes in speed over the course of a transition. The image below shows a graph of `cubic-bezier(0.42, 0, 1, 1)`.

In most cases, it's easier to use a timing function keyword. We mentioned `step-start` and `step-end` in the previous section, but there are five more keywords, each of which is an alias for `cubic-bezier` values. They're listed in the following table:

Keyword	Equivalent function	Effect
`ease`	`cubic-bezier(0.25, 0.1, 0.25, 1)`	Begins slowly, accelerates quickly, then slows towards the end of the transition
`ease-in`	`cubic-bezier(0.42, 0, 1, 1)`	Begins quickly, then accelerates slowly but steadily until the end of the transition
`ease-out`	`cubic-bezier(0, 0, 0.58, 1)`	Accelerates quickly but slows towards the end of the transition
`ease-in-out`	`cubic-bezier(0.42, 0, 0.58, 1)`	Begins slowly, accelerates quickly, then decelerates towards the end of the transition
`linear`	`cubic-bezier(0, 0, 1, 1)`	Speed remains consistent over the course of the animation

Transitioning Multiple Properties

It's possible to transition multiple properties of a single element using a transition list. Let's look at an example:

```
div {
    background: #E91E63;
    height: 200px;
    width: 200px;
    margin: 10px 0;
    position: relative;
    left: 0;
    top: 3em;
    transition: left 4s cubic-bezier(0.175, 0.885, 0.32,
1.275),
                background 2s 500ms;
}
.transthem {
    left: 30%;
    background: #00BCD4;
}
```

Here, we’ve defined transitions for the `left` and `background` properties. The difference is that each item is separated by a comma. The `left` transition will last four seconds and use a `cubic-bezier` timing function. The `background` transition will only last two seconds, but it begins after a half-second (`500ms`) delay.

Occasionally, you may need to detect when a transition ends in order to take another action. For example, if you transition `opacity: 1` to `opacity: 0`, it’s a good idea to add a `hidden` attribute to the element for improved assistive technology support. This is where the `transitionend` event comes in handy.

When a transition completes, the browser fires a `transitionend` event on the affected element—one for each property. We can listen for these events using `addEventListener`:

```
const transitionEndHandler = function() {
    // Do something.
}

const element = document.getElementById('el');
element.addEventListener('transitionend',
transitionEndHandler);
```

HTML also supports an `ontransitionend` attribute. The code above could also be written as follows:

```
const transitionEndHandler = function() {
    // Do something.
}
const element = document.getElementById('el');
element.ontransitionend = transitionEndHandler;
```

Shorthand Properties

In cases where the property is a shorthand property, the browser will fire one event for each longhand property. In other words, a transition of the `padding` property will result in `transitionend` events for `padding-top`, `padding-right`, `padding-bottom`, and `padding-left`.

Let’s put this knowledge to use. In this example, we’ll hide unselected form options when the user picks one. Our (simplified) HTML follows:

```
<h1>Please select your favorite color of the ones shown
below.</h1>
<form>
    <ul>
        <li>
            <input type="radio" name="favecolor" id="red">
<label for="red">Red</label>
        </li>
        <li>
            <input type="radio" name="favecolor" id="yellow">
<label for="yellow">Yellow</label>
        </li>
        <li>
            <input type="radio" name="favecolor" id="blue">
<label for="blue">Blue</label>
        </li>
    </ul>
    <div id="thanks" hidden>Thank you for selecting your
favorite color.</div>
    <button type="reset">Reset</button>
</form>
```

And here’s our (also simplified) CSS:

```
li {
    transition: 500ms;
}
.fade {
    opacity: 0;
}
```

Add some styles for color and font size, and we end up with the example below.

Now let's tie it together with JavaScript. First, let's define an action that adds the `fade` class—in this case, a `change` event handler:

```
const changeHandler = function() {
    // Select unchecked radio buttons. Returns a NodeList.
    const notfave = document.querySelectorAll( 'input:not(
:checked )' );

    // Create a new array from the NodeList
    notfave.forEach( function( item ) {
        // Find the parent node, and add a 'fade' class
        item.parentNode.classList.add( 'fade' );
    });
};

const form = document.querySelector( 'form' );
form.addEventListener( 'change', changeHandler );
```

When the user selects a color, our form element will receive a `change` event. That in turn triggers the `changeHandler` method, which adds a `fade` class to the parent element of each radio button. This is what triggers our transition.

The `forEach` DOM Function

The `forEach` method used above is a DOM function for iterating through a **NodeList**, or collection of elements. It's supported in most major browsers, with the exception of Internet Explorer 11. It's not the `forEach` method of JavaScript. The Mozilla Developer Network covers forEach in depth.

Now let's take a look at our `transitionend` handler. It's slightly different from the other examples in this chapter:

```
const transitionendHandler = function( domEvent ) {
    domEvent.target.setAttribute( 'hidden', '' );
    document.getElementById( 'thanks' ).removeAttribute(
'hidden' );
};

document.addEventListener( 'transitionend',
transitionendHandler );
```

Our `transitionendHandler` accepts a single event object argument. Here, we've named it `domEvent`, but you could name it `evt`, `foo`—just about anything. This event object is passed automatically, according to behavior defined by the Document Object Model Level 2 specification. In order to reference this event object within our handler, we need to define it as a parameter for our function.

Every event object includes a `target` property. This is a reference to the element that received the event. In this case, it's a list item, and we're adding a `hidden` attribute to each (`eventObject.target.setAttribute('hidden', '')`). The last line of our event handler removes the `hidden` attribute from our "Thank you" message, as seen below.

Multiple Transitions and `transitionend` Events

Transitions of multiple properties trigger multiple `transitionend` events. A declaration such as `transition: left 4s linear, background 2s`

`500ms ease;` triggers a `transitionend` event for the `left` property and another for `background`. To determine which transition triggered the event, you can check the `propertyName` property of the event object:

```
const transitionendHandler = function ( eventObject ) {
    if ( eventObject.propertyName === 'opacity' ) {
        // Do something based on this value.
    }
};
```

Occasionally, a transition will fail to complete. This can typically happen when the property is overridden while it's in progress—such as when a user action removes the class name. In those situations, the `transitionend` event won't fire.

Because of this risk, avoid using the `transitionend` event to trigger anything "mission critical", such as a form submission.

CSS Animation

Think of CSS animation as the more sophisticated sister to CSS transitions. Animations differ from transitions in a few key ways:

- Animations don't degrade gracefully. If there's no support from the browser, the user is out of luck. The alternative is to use JavaScript.
- Animations can repeat, and repeat infinitely. Transitions are always finite.
- Animations use keyframes, which offer the ability to create more complex and nuanced effects.
- Animations can be paused in the middle of the play cycle.

The latest versions of all major browsers support CSS animations. Firefox versions 15 and earlier require a `-moz-` prefix; later version don't. Internet Explorer versions 10 and 11 also support animations without a prefix, as do all versions of Microsoft Edge.

We can check for CSS animations support in a few ways. The first is by testing for the presence of `CSSKeyframeRule` as a method of the `window` object:

```
const hasAnimations = 'CSSKeyframeRule' in window;
```

If the browser supports the `@supports` rule and the `CSS.supports()` API (discussed in Chapter 10, "Applying CSS Conditionally"), we can use that instead:

```
const hasAnimations = CSS.supports( 'animation-duration: 2s'
);
```

As with transitions, we can only animate interpolatable values such as color values, lengths, and percentages.

Creating Your First Animation

We first have to define an animation using an `@keyframes` rule. The `@keyframes` rule has two purposes:

- setting the name of our animation
- grouping our keyframe rules

Let's create an animation named `pulse`:

```
@keyframes pulse {

}
```

Our keyframes will be defined within this block. In animation, a **keyframe** is a point at which the action changes. With CSS animations specifically, keyframe rules are used to set property values at particular points in the animation cycle. Values that fall between the values in a keyframe rule are interpolated.

At the minimum, an animation requires two keyframes: a `from` keyframe, which is the starting state for our animation, and a `to` frame, which is its end state. Within each individual keyframe block, we can define which properties to animate:

```
@keyframes pulse {
    from {
        transform: scale(0.5);
        opacity: .8;
    }
    to {
        transform: scale(1);
        opacity: 1;
    }
}
```

This code will scale our object from half its size to its full size, and change the opacity from 80% to 100%.

The `keyframes` rule only *defines* an animation, though. By itself, it doesn't make elements move. We need to apply it. Let's also define a `pulse` class that we can use to add this animation to any element:

```
.pulse {
    animation: pulse 500ms;
}
```

Here, we've used the `animation` shorthand property to set the animation name and duration. In order for an animation to play, we need the name of an `@keyframes` rule (in this case, `pulse`) and a duration. Other properties are optional.

The order of properties for `animation` is similar to that of `transition`. The first value that can be parsed becomes the value of `animation-duration`. The second value becomes the value for `animation-delay`. Words that aren't CSS-wide keywords or animation property keyword values are assumed to be `@keyframe` rule set names.

As with `transition`, `animation` also accepts an animation list. The animation list is a comma-separated list of values. We could, for example, split our pulse animation into two rules—`pulse` and `fade`:

```
@keyframes pulse {
    from {
        transform: scale(0.5);
    }
    to {
        transform: scale(1);
    }
}
@keyframes fade {
    from {
        opacity: .5;
    }
    to {
        opacity: 1;
    }
}
```

We can combine them as part of a single animation list:

```
.pulse-and-fade {
    animation: pulse 500ms infinite, fade 500ms 8;
}
```

Or, as an alternative, we can combine them using longhand properties:

```
.pulse-and-fade {
    animation-name: pulse, fade;
    animation-duration: 500ms; /* used for both animations */
    animation-iteration-count: infinite, 8;
}
```

Animation Properties

Though using the `animation` property is shorter, sometimes longhand properties are clearer. Longhand animation properties are listed in the following table:

Property	Description	Initial value
`animation-delay`	How long to wait before executing the animation	`0s` (executes immediately)
`animation-duration`	How long the cycle of an animation should last	`0s` (no animation occurs)
`animation-name`	The name of an `@keyframes` rule	none
`animation-timing-function`	How to calculate the values between the start and end states	`ease`
`animation-iteration-count`	How many times to repeat the animation	`1`

Property	Description	Initial value
`animation-direction`	Whether or not the animation should ever play in reverse	`normal` (no reverse)
`animation-play-state`	Whether the animation is running or paused	`running`
`animation-fill-mode`	Specifies what property values are applied when the animation isn't running	`none`

The `animation-delay` and `animation-duration` properties function like `transition-delay` and `transition-duration`. Both accept time units as a value, either in seconds (`s`) or milliseconds (`ms`). Negative time values are valid for `animation-delay`, but not `animation-duration`.

Let's rewrite our `.pulse` rule set using longhand properties. Doing so gives us the following:

```
.pulse {
    animation-name: pulse;
    animation-duration: 500ms;
}
```

The `animation-name` property is fairly straightforward. Its value can be either `none` or the name of the `@keyframes` rule. Animation names have few restrictions. CSS keywords such as `initial`, `inherit`, `default`, and `none` are forbidden. Most punctuation characters won't work, while letters, underscores, digits, and emojis (and other Unicode characters) usually will. For clarity and maintainability, it's a good idea to give your animations descriptive names, and avoid using CSS properties or emojis as names.

To Loop or Not to Loop: The `animation-iteration-count` Property

If you're following along with your own code, you'll notice that this animation only happens once. We want our animation to repeat. For that, we'll need the `animation-iteration-count` property.

The `animation-iteration-count` property accepts most numeric values. Whole numbers and decimal numbers are valid values. With decimal numbers, however, the animation will stop partway through the last animation cycle, ending in the `to` state. Negative `animation-iteration-count` values are treated the same as `1`.

To make an animation run indefinitely, use the `infinite` keyword. The animation will play an infinite number of times. Of course, `infinite` really means until the document is unloaded, the browser window closes, the animation styles are removed, or the device shuts down. Let’s make our animation infinite:

```
.pulse {
    animation-name: pulse;
    animation-duration: 500ms;
    animation-iteration-count: infinite;
}
```

Or, using the `animation` shorthand property:

```
.pulse {
    animation: pulse 500ms infinite;
}
```

Playing Animations: The `animation-direction` Property

There’s still a problem with our animation, however. It doesn’t so much *pulse* as repeat our scaling-up animation. What we want is for this element to scale up and down. Enter the `animation-direction` property.

The `animation-direction` property accepts one of four values:

- `normal`: the initial value, playing the animation as specified
- `reverse`: flips the `from` and `to` states and plays the animation in reverse
- `alternate`: plays even-numbered animation cycles in reverse
- `alternate-reverse`: plays odd-numbered animation cycles in reverse

To continue with our current example, `reverse` would scale down our object by a factor of 0.5. Using `alternate` would scale our object up for the odd-numbered cycles and down for the even-numbered ones. Conversely, using `alternate-reverse` would scale our object down for the odd-numbered cycles and up for the even ones. Since this is the effect we want, we'll set our `animation-direction` property to `alternate-reverse`:

```
.pulse {
    animation-name: pulse;
    animation-duration: 500ms;
    animation-iteration-count: infinite;
    animation-direction: alternate-reverse;
}
```

Or, using the shorthand property:

```
.pulse {
    animation: pulse 500ms infinite alternate-reverse;
}
```

Using Percentage Keyframes

Our previous example was a simple pulse animation. We can create more complex animation sequences using percentage keyframes. Rather than using `from` and `to`, **percentage keyframes** indicate specific points of change over the course of the animation. Below is an example using an animation named `wiggle`:

```
@keyframes wiggle {
    25% {
        transform: scale(.5) skewX(-5deg) rotate(-5deg);
    }
    50% {
         transform: skewY(5deg) rotate(5deg);
    }
    75% {
         transform: skewX(-5deg) rotate(-5deg) scale(1.5);
     }
    100% {
        transform: scale(1.5);
    }
}
```

We've used increments of 25% here, but these keyframes could be 5%, 10%, or 33.2%. As the animation plays, the browser will interpolate the values between each state. As with our previous example, we can assign it to a selector:

```
/* Our animation will play once */
.wiggle {
    animation-name: wiggle;
    animation-duration: 500ms;
}
```

Or using the `animation` shorthand property:

```
.wiggle {
    animation: wiggle 500ms;
}
```

There's just one problem here. When our animation ends, it goes back to the original, pre-animated state. To prevent this, use the `animation-fill-mode` property.

The `animation-fill-mode` Property

Animations have no effect on properties before they begin or after they stop playing. But as you've seen with the `wiggle` example, once an animation ends, it reverts to its pre-animation state. With `animation-fill-mode`, we can fill in those states before the animation starts and ends.

The `animation-fill-mode` property accepts one of four values:

- `none`: the animation has no effect when it's not executing
- `forwards`: when the animation ends, the property values of the end state will still apply
- `backwards`: property values for the first keyframe will be applied during the animation delay period
- `both`: effects for both `forwards` and `backwards` apply

Since we want our animated element to remain in its final, scaled-up state, we're going to use `animation-fill-mode: forwards` (noting that `animation-fill-mode: both` would also work).

The effect of `animation-fill-mode: backwards` is most apparent when the `animation-delay` property is set to `500ms` or higher. When `animation-fill-mode` is set to `backwards`, the property values of the first keyframe are applied, but the animation isn't executed until the delay elapses.

Pausing Animations

As has been mentioned, animations can be paused. Transitions can be reversed midway, or stopped altogether by toggling a class name. Animations, on the other hand, can be paused partway through the play cycle using `animation-play-state`. It has two defined values—`running` and `paused`—and its initial value is `running`.

Let's look at a simple example of using `animation-play-state` to play or pause an animation. First, our CSS:

```
.wobble {
    animation: wobble 3s ease-in infinite forwards alternate;
    animation-play-state: paused;
}
.running {
    animation-play-state: running;
}
```

Here, we have two declaration blocks: `wobble`, which defines a wobbling animation, and `running`, which sets a play state. As part of our `animation` declaration, we've set an `animation-play-state` value of `paused`. To run our animation, we'll add the `running` class to our element. Let's assume that our markup includes a **Run** animation button with an `id` of `trigger`:

```
const trigger = document.querySelector( '#trigger' );
const moveIt = document.querySelector( '.wobble' );

trigger.addEventListener( 'click', function() {
    moveIt.classList.toggle( 'running' );
});
```

Adding `.running` to our element overrides the `animation-play-state` value set in `.wobble`, and causes the animation to play.

Detecting When Animations Start, End, or Repeat

Like transitions, animations fire an event when they end: `animationend`. Unlike transitions, animations also fire `animationstart` and `animationiteration` events when they begin to repeat. As with transitions, you might use these events to trigger another action on the page. For example, you might use `animationstart` to contextually reveal a **Stop Animation** button, or `animationend` to reveal a **Replay** button.

We can listen for these events with JavaScript. Below, we’re listening for the `animationend` event:

```
const animate = document.getElementById( 'animate' );

animate.addEventListener( 'animationend', function( domEvent
) {
    // Do something
});
```

Here, too, the event handler function receives an event object as its sole argument. In order to determine which animation ended, we can query the `animationName` property of the event object.

Animation and Accessibility

Transitions and animations can enhance the user experience by making interactions smooth rather than jumpy, and otherwise bring delight to the interface. But they have accessibility risks. Large spinning animations, for example, can cause dizziness or nausea for people with vestibular disorders, such as vertigo. Consider adding controls for larger, longer, or infinite animations so users can turn them off.

Vestibular Disorders and Accessible Animation

Rachel Nabors’ “Infinite Canvas 6: Vestibular Disorders and Accessible Animation” is a great introduction to the subject of vestibular disorders and animation.

You can also use media queries and the `prefers-reduced-motion` feature to reduce or disable animation. Users can indicate that they prefer less motion, typically by adjusting the accessibility settings for their operating system.

If you’d like your website to respect those preferences, you *must* include the `prefers-reduced-motion` media query. Browsers won’t do it on their own.

For example:

```
.wobble {
    animation: wobble 3s ease-in infinite forwards alternate;
    animation-play-state: paused;
}
.running {
    animation-play-state: running;
}
@media screen and ( prefers-reduced-motion ) {
    .running {
        animation-play-state: paused;
    }
}
```

In this example, if the user has indicated that they prefer reduced motion, the `animation-play-state` will be `paused`. If there are controls associated with this animation (such as a **Play** button), you might use JavaScript to add a `hidden` attribute to them.

You don’t have to completely disable your animations. For example, if your animation scales and also skews, as with our `.wobble` animation, you can instead disable a portion of it. Here we’ll change the scale value:

```
.wobble {
    --wobble-min-scale: .5;
    --wobble-max-scale: 1.5;
}
@media screen and ( prefers-reduced-motion ) {
    .wobble {
        --wobble-min-scale: 1;
        --wobble-max-scale: 1;
    }
}
@keyframes wobble {
    25% {
        transform: scale( var(--wobble-min-scale) )
skewX(-5deg) rotate(-5deg);
    }
    50% {
         transform: skewY(5deg) rotate(5deg);
    }
    75% {
         transform: skewX(-5deg) rotate(-5deg) scale( var(--
wobble-max-scale) );
```

```
     }
    100% {
        transform: scale( var(--wobble-max-scale) );
    }
}
```

Notice that we've used custom properties (see Chapter 4) to manage the scale factor, and applied them to the `.wobble` selector.

We'll cover the ins and outs of media queries in Chapter 10, "Applying CSS Conditionally".

Flashing animations can trigger seizures in some people with photosensitive epilepsy; WCAG 2.1 includes advice for for avoiding flashes and animations that are known to trigger seizures. Avoid flashing content more than three times per second, particularly across large areas of the screen.

A Note about Performance

Some properties create better-performing transitions and animations than others. If an animation updates a property that triggers a reflow or repaint, it may perform poorly on low-powered devices such a phones and tablets.

Properties that trigger a reflow are ones that affect layout. These include the following animatable properties:

- `block-size`
- `border-width` (and `border-*-width` properties)
- `border` (and `border-*` properties)
- `bottom`
- `font-size`
- `font-weight`
- `height`
- `inset-block` (and `inset-block-*`) longhand properties
- `inset-inline` (and `inset-inline-*`) longhand properties
- `inline-size`
- `left`
- `line-height`
- `margin` (and `margin-*` properties)

- `min-height`
- `min-width`
- `max-height`
- `max-width`
- `padding` (and `padding-*` properties)
- `right`
- `top`
- `vertical-align`
- `width`

When these properties are animated, the browser must recalculate the size and position of the affected—and often neighboring—elements. Use transforms where you can. Transitioning or animating translation transforms can replace `top`, `left`, `right`, and `bottom` or `inset-block-*` and `inset-inline-*` properties. Take, for example, the animation below that reveals a menu:

```
[id=menu] {
    left: -300px;
    transition: left 500ms ease-in;
}
[id=menu].open {
    left: 0;
}
```

We could rewrite this using a translation transform:

```
[id=menu] {
    transform: translateX( -300px );
    transition: transform 500ms ease-in;
}
[id=menu].open {
    transform: translateX( 0 );
}
```

Browsers calculate and apply transforms *after* they calculate the document's layout. As a result, transforms tend to be smoother, and less resource-intensive. We'll cover transforms in Chapter 8.

CSS Triggers

The CSS Triggers reference is a good starting point for learning how browsers treat various CSS-related properties. Keep in mind that it's a little dated. Newer properties such as `block-size` aren't included, and Microsoft Edge has moved away from the EdgeHTML engine. Still, it's one of the more comprehensive guides available.

Animations that take up a lot of screen real estate or that contain a lot of child elements may also perform poorly. In such cases, try adding the `will-change` property to an element:

```
header {
    perspective: 400px;
    perspective-origin: 50% 50%;
}
[id=megamenu] {
    width: 100vw;
    height: 100vh;
    transform: rotateX( -90deg );
    transform-origin: 50% 0;
    transition: transform 1s;

    will-change: transform;
}
[id=megamenu].open {
    transform: rotateX( 0deg );
}
```

The `will-change` property indicates to the browser that an element will change soon. Set its value to the value of the property you plan to animate. Be careful with `will-change`, however. It's best used for a single element, and only after you've determined that a particular animation or transition doesn't perform well. Consider it a property of last resort.

Using `will-change`

Use `will-change` sparingly. Even the Will Change specification says that the optimizations that `will-change` triggers may use more of the machine's resources if used too widely. Sara Soueidan's "Everything You Need to Know About the CSS `will-change` Property" has more detail about when to use—and not to use—`will-change`.

Properties that trigger a repaint are typically those that cause a color change. These include:

- `background`
- `background-image`
- `background-position`
- `background-repeat`
- `background-size`
- `border-radius`
- `border-style`
- `box-shadow`
- `color`
- `outline`
- `outline-color`
- `outline-style`
- `outline-width`

Changes to these properties are less expensive to calculate than those that affect layout, but they do still have a cost. Changes to `box-shadow` and `border-radius` are especially expensive to calculate, especially for low-powered devices. Use caution when animating these properties.

Conclusion

In this chapter, we've looked at how to add motion to web pages using CSS transitions and animations, and why you might like to do so. We've also touched on performance and accessibility concerns, and explained the finer points of the `cubic-bezier` function.

As you use transitions and animations, consider *how* you're using them. They're best used to focus the user's attention or clarify an action. But they can also be used to add whimsy and delight.

Chapter 8: Transforms

Transforms allow us to create effects and interactions that are otherwise impossible. When combined with transitions and animations, we can create elements and interfaces that rotate, dance and zoom. Three-dimensional transforms, in particular, make it possible to mimic physical objects.

Take, for example, the humble postcard received from a friend. Its front face displays a photo of the location your friend sent the card from. When you flip it over, you see expanded information about the photo and your friend's journey. (By the way, they wish you were there.)

A postcard isn’t a web interface, obviously, but it’s a metaphor for the kind of interfaces we can create. Perhaps you want to build a weather widget that functions similarly to a postcard. The front of our widget contains a current weather summary, as shown below.

Flipping it over—triggered by a tap or swipe—might show an expanded weather forecast, or reveal a **Settings** panel.

Card-style interfaces are a great example of what we can build with transforms. In this chapter, we’ll do a deep dive into the details of how they work.

Checking Out the Spec

Transforms are defined by the CSS Transforms Module Level 2 specification. At one point, two-dimensional and three-dimensional transforms were defined in separate specifications. As you move through the chapter, you'll notice some redundancy in function names.

How Transforms Affect Layout

Before we go too much further, there are some things you should know about how the `transform` property affects layout. When you apply the `transform` property to an element and its value is other than `none`, three things happen:

- the element becomes a containing block for child elements
- it establishes a new stacking context for the element and its children
- it imposes a local coordinate system within the element's bounding box

Let's look at these concepts individually.

`transform` Creates a Containing Block

When an element is **positioned**—that is, when the value of the `position` property is something other `static`—it's drawn relative to a containing block. A **containing block** is the closest positioned ancestor or, failing that, the root element (such as `<html>` or `<svg>`) of a document.

Consider the example pictured below.

In this image, the child rectangle has a `position` value of `absolute`. Its `right` and `bottom` properties are both set to `0`. Its parent element has a `position` value of `relative`. Because the parent in this case is positioned, it becomes a containing block for the child. If the parent rectangle were not positioned, this child element would instead be drawn at the bottom right of the browser window.

Transforms work similarly. Setting the value of `transform` to something other than `none` turns the transformed element into a containing block. Positioned children of a transformed element are positioned relative to that element, as seen below.

In the image above, the parent element *isn't* positioned. The `transform` property is what's creating this containing block. A child element with `position: absolute` is nested within an element with `transform: skewX(-15deg)`.

`transform` Creates a New Stacking Context

A transform also creates a new stacking context for the element it's applied to. As you may recall from Chapter 5, "Layouts", elements within a stacking context are painted from back to front, as follows:

1. child-stacking contexts with a negative stack level (for example, positioned `z-index: -1`)
2. nonpositioned elements

3. child-stacking contexts with a stack level of `0` (for example, positioned and `z-index: 0;` or `z-index: auto;`)
4. child-stacking contexts with positive stack levels (for example, `z-index: 1`), which sit at the top of the stack

Setting the value of `transform` to something other than `none` makes the element's stack level `0`. Transformed elements are stacked in front of non-positioned elements. The `z-index` values of each child element are relative to the parent. Let's update our example from Chapter 5 to see how this works:

```
<div style="position:relative;">
    <div id="a">
        <p><b>div#a</b></p>
    </div>

    <div id="b" style="transform: scale(2) translate(25%,
15%);">
        <p><b>div#b</b></p>
    </div>

    <div id="c" style="position:relative; z-index: 1">
        <p><b>div#c</b></p>
    </div>
    <div id="d" style="position:absolute; z-index: -1">
        <p><b>div#d</b></p>
    </div>
</div>
```

In this case, `div#d` sits at the bottom of the stack, and `div#a` sits above it (as pictured below). But `div#b` comes next because the `transform` property forces its `z-index` value to be `0` instead of `auto`. With `z-index: 1`, `div#c` sits at the top of the stack.

Three-dimensional transforms add additional complexity. An element shifted along the Z-axis may render on a different plane from its container. Elements may also intersect with other elements across layers. Still, the basic rules of the stacking order apply.

Transformed elements may also overlap other elements on the page and prevent them from receiving mouse, touch, or pointer events. Applying `pointer-events: none` to the transformed element solves this issue.

We’re Talking About the CSS Property Here

The `pointer-events` CSS property is distinct from the PointerEvents DOM event object.

Browsers apply transforms after elements have been sized and positioned. Unlike floated elements, transformed elements aren’t removed from the normal flow.

Document Flow

Document flow is described by the Visual formatting model section of the CSS2.1 specification. Updates to this model are partly described by the CSS Display Module Level 3.

Because transforms are applied after the layout has been calculated, they don’t affect document layout. Transformed child elements may overflow the parent element’s bounding box, but they don’t affect the position of other elements on the page. They also don’t affect the `HTMLElement.offsetLeft` or `HTMLElement.offsetTop` DOM properties of an element. Using these properties to detect the rendered position of an element will give you inaccurate results.

Transforms do, however, affect client rectangle values and visual rendering of elements. To determine the rendered left and top positions of an element, use the `HTMLElement.getClientRects()` or `HTMLElement.getBoundingClientRect()` DOM methods (for example, `document.getElementById('#targetEl').getClientRects()`). Because they don’t force the browser to recalculate page layout, transforms typically perform better than properties such as `left` and `height` when animated.

`transform` Creates a Local Coordinate System

You may recall from geometry class that the Cartesian **coordinate system** is a way of specifying points in a plane. You may also recall that a **plane** is a flat, two-dimensional surface that extends infinitely along the horizontal and vertical axes. These axes are also known as the X-axis and Y-axis.

Point values along the X-axis increase as you move from left to right, and decrease from right to left. Y-axis point values decrease as you move up from the origin, and decrease as you move down. The X- and Y-axes are perpendicular to each other. Where they cross is known as the **origin**, and the coordinates of its location are always (0,0), as illustrated below.

A three-dimensional coordinate system also has a Z-axis. This axis is perpendicular to both the X- and Y-axes, as well as the screen (see the image below). The point at which the Z-axis crosses the X- and Y-axes is also known as the origin. Its coordinates are (0,0,0).

A rendered HTML document is also a coordinate system. The top-left corner is the origin, with coordinates of (0,0) or (0,0,0). Values increase along the X-axis as you move right. However, unlike the Cartesian system mentioned above, values along the Y-axis *increase* as you move down the screen or page. Z-axis values increase as elements move towards the viewer and decrease as they move away from the viewer.

Setting the value of `transform` to a value besides `none` creates a **local coordinate system** for the selected elements. The origin—point (0,0) or (0,0,0)—in this local coordinate system sits at the center of the element's bounding box. We can change the position of the origin, however, by using the `transform-origin` property. Points within the element's bounding box are transformed relative to this local origin.

The `transform-origin` Property

The `transform-origin` property accepts up to three values, one for each of the X, Y, and Z positions—for example, `transform-origin: 300px 300px` for a 2D transformation, or `transform-origin: 0 0 200px` for a 3D transformation.

If one value is specified, the second value is assumed to be `center`, and the third value is assumed to be `0px`.

Both the X- and Y-coordinates may be percentages, lengths, or positioning keywords. Positioning keywords are `left`, `center`, `right`, `top`, and `bottom`. The Z position, however, must be a length. In other words, `transform-origin: left bottom 200px` works, but `transform-origin: left bottom 20%` doesn't.

Setting `transform-origin` moves the (0,0) point of the local coordinate system to a new location within the element's bounding box. This, of course, modifies the transformation, sometimes radically. The image below shows a `transform-origin` point of `50% 50%` and one at `0px 0px`.

Now that you know a little more about how transforms affect document layout, let's dig into the transform functions. This is how we make the magic. Transforms let us rotate, flip, skew, and scale elements. When combined with animations and transitions, we can create slick motion graphic effects.

Transforms can be grouped into two categories: 2D and 3D. Each group contains functions for rotating, skewing, scaling, and translating. 2D functions are concerned with transformations of points along the X- and Y-axes. 3D functions add the third dimension of depth and affect points along the Z-axis.

2D Transform Functions

There are four primary two-dimensional transform functions: `rotate()`, `scale()`, `skew()`, and `translate()`. Six other functions let us transform an element in a single dimension: `scaleX()` and `scaleY()`; `skewX()` and `skewY()`; and `translateX()` and `translateY()`.

`rotate()`

A rotation transform spins an element around its origin by the angle specified around the `transform-origin` point. Using `rotate()` tilts an element clockwise (positive angle values) or counterclockwise (negative angle values). Its effect is much like a windmill or pinwheel, as pictured below, where the purple box has been rotated 55 degrees from its start position, shown by the dotted line.

The `rotate()` function accepts values in angle units. Angle units are defined by the CSS Values and Units Module Level 3 specification. These may be `deg` (degrees), `rad` (radians), `grad` (gradians), or turn (`turn`) units. One complete rotation is equal to `360deg`, `6.28rad`, `400grad`, or `1turn`.

Rotation values that exceed one rotation (say, `540deg` or `1.5turn`) are rendered according to their remaindered value, unless animated or transitioned. In other words, `540deg` is rendered the same as `180deg` (540 degrees minus 360 degrees) and `1.5turn` is rendered the same as `.5turn` (1.5 minus 1). But a transition or animation from `0deg` to `540deg` or `1turn` to `1.5turn` rotates the element one-and-a-half times.

2D Scaling Functions: `scale()`, `scaleX()`, and `scaleY()`

With scaling functions, we can increase or decrease the rendered size of an element in the X-dimension (`scaleX()`), Y-dimension (`scaleY()`), or both (`scale()`). Scaling is illustrated below, where the border illustrates the original boundaries of the box, and the + marks its center point. The red box (left) is scaled by a factor or two (right).

Each scale function accepts a multiplier or factor as its argument. This multiplier can be just about any positive or negative number. Percentage values aren't supported. Positive multipliers greater than `1` increase the size of an element. For example, `scale(1.5)` increases the size of the element in the X and Y directions 1.5 times, as illustrated below. The northern cardinal drawing on the left is not transformed. The cardinal drawing on the right has been scaled to 1.5 times the size of the original illustration.

Cardinal drawing by [kattekrab] from Openclipart.

Positive multipliers between `0` and `1` reduce the size of an element. Negative multipliers scale the element, but they also flip or reflect it along one or both axes. In the image below, the northern cardinal drawing on the left is not transformed, while the one on the right has a negative scaling transformation (`transform: scale(-1.5)`) applied.

Watch Your Scale

Using `scale(0)` causes the element to disappear, because multiplying a number by zero results in a product of zero.

Using `scale(1)` creates an **identity transformation**, which means it's drawn to the screen as if no scaling transformation was applied. Using `scale(-1)` won't change the drawn size of an element, but the negative value causes the element to be reflected. Even though the element doesn't appear transformed, it still triggers a new stacking context and containing block.

You can also scale the X- and Y-dimensions separately by passing two arguments to the `scale()` function, such as `scale(1.5, 2)`. The first argument scales the X-dimension; the second scales the Y-dimension. We could, for example, reflect an object along the X-axis alone using `scale(-1, 1)`. Passing a single argument scales both dimensions by the same factor.

2D Translation Functions: `translateX()`, `translateY()`, and `translate()`

Translating an element offsets its painted position from its layout position by the specified distance. As with other transforms, translating an element doesn't change its `offsetLeft` or `offsetTop` positions. It does, however, affect where it's visually positioned on screen.

The 2D translation functions—`translateX()`, `translateY()`, and `translate()`—accept lengths or percentages for arguments. Length units include pixels (`px`), `em`, `rem`, and viewport units (`vw` and `vh`).

The `translateX()` function changes the horizontal rendering position of an element. If an element is positioned zero pixels from the left, `transform: transitionX(50px)` shifts its rendered position 50 pixels to the right of its start position. Similarly, `translateY` changes the vertical rendering position of an element. A transform of `transform: transitionY(50px)` offsets the element vertically by 50 pixels.

With `translate()`, we can shift an element vertically and horizontally using a single function. It accepts up to two arguments: the X translation value, and the Y translation value. The image below shows the effect of an element with a `transform` value of `translate(120%, -50px)`, where the left green square is in the original position, and the right green square is translated 120% horizontally and -50 pixels vertically from its containing element (the dashed border).

Passing a single argument to `translate` is the equivalent of using `translateX`; the Y translation value will be set to zero. Using `translate()` is the more concise option. Applying `translate(100px, 200px)` is the equivalent of `translateX(100px) translateY(200px)`.

Positive translation values move an element to the right (for `translateX`) or downward (for `translateY`). Negative values move an element to the left (`translateX`) or upward (`translateY`).

Translations are particularly great for moving items left, right, up, or down. Updating the value of the `left`, `right`, `top`, and `bottom` properties forces the browser to recalculate layout information for the entire document. But transforms are calculated *after* the layout has been calculated. They affect where the elements *appear* on screen, but not their actual dimensions. Yes, it's weird to think about document layout and rendering as separate concepts, but in terms of browsers, they are.

Speed Matters

Google's "Why does speed matter?" discusses some of the differences between layout or rendering, and painting or drawing.

`skew`, `skewX`, and `skewY`

Skew transformations shift the angles and distances between points while keeping them in the same plane. Skew transformations are also known as *shear transformations*, and they distort the shapes of elements, as seen below, where a rectangle is skewed 45 degrees along its X-dimension—the dashed line representing the original bounding box of the element.

The skew functions—`skew()`, `skewX()`, and `skewY()`—accept most angle units as arguments. Degrees, gradians, and radians are valid angle units for the skew functions, while turn units, perhaps obviously, are not.

The `skewX()` function shears an element in the X direction (or horizontally). In the image below, the left object isn't transformed, while the right object reveals the effect of `transform: skewX(30deg)`.

`skewX` accepts a single parameter, which again must be an angle unit. Positive values shift the element to the left, and negative values shift it towards the right.

Similarly, `skewY` shears an element in the Y direction (vertically). The image below shows the effect of `transform: skewY(30deg).`

With `skewY`, points to the right of the origin are shifted downward with positive values. Negative values shift these points upward.

This brings us to the `skew()` function. The `skew()` function requires one argument, but accepts up to two. The first argument skews an element in the X direction, and the second skews it in the Y direction. If only one argument is provided, the second value is assumed to be zero, making it the equivalent of skewing in the X direction alone. In other words, `skew(45deg)` renders the same as `skewX(45deg)`.

Current Transform Matrix

So far, we’ve discussed transform functions separately, but they can also be combined. Want to scale and rotate an object? No problem: use a **transform list**. For example:

```
.rotatescale {
    transform: rotate(45deg) scale(2);
}
```

This produces the results you see below.

Order matters when using transform functions. This is a point that’s better shown than talked about, so let’s look at an example to illustrate. The following CSS skews and rotates an element:

```
.transformEl {
    transform: skew(10deg, 15deg) rotate(45deg);
}
```

It gives us the result you see below.

What happens if you rotate an element first and then skew it?

```
.transformEl {
    transform:  rotate(45deg) skew(10deg, 15deg);
}
```

The effect is quite different, as seen below.

Each of these transforms has a different *current transform matrix* created by the order of its transform functions. To fully understand why this is, we'll need to learn a little bit of *matrix multiplication*. This will also help us understand the `matrix()` and `matrix3d()` functions.

Matrix Multiplication and the Matrix Functions

A **matrix** is an array of numbers or expressions arranged in a rectangle of rows and columns. All transforms can be expressed using a 4×4 matrix.

This matrix corresponds to the `matrix3d()` function, which accepts 16 arguments, one for each value of the 4×4 matrix. Two-dimensional transforms can also be expressed using a 3×3 matrix, seen in the following image.

This 3×3 matrix corresponds to the `matrix()` transform function. The `matrix()` function accepts six parameters, one each for values *a* through *f*.

Each transform function can be described using a matrix and the `matrix()` or `matrix3d()` functions. The image below shows the 4×4 matrix for the `scale3d()` function, where *sx*, *sy*, and *sz* are the scaling factors of the X-, Y-, and Z-dimensions respectively.

When we combine transforms—such as `transform: scale(2) translate(30px, 50px)`—the browser multiplies the matrices for each function to create a new matrix. This new matrix gets applied to the element.

But here's the thing about matrix multiplication: it isn't commutative. With simple values, the product of 3×2 is the same as 2×3. With matrices, however, the product of $A \times B$ is not necessarily the same as the product of $B \times A$. Let's look at the image below as an example, where we calculate the matrix product of `transform: scale(2) translate(30px, 50px).`

Our product results in a matrix that scales our element by a factor of two along the X- and Y-axes, and offsets each pixel in the element horizontally by 60 pixels and vertically by 100 pixels. We can also express this product using the `matrix()` function: `transform: matrix(2, 0, 0, 2, 60, 100)`.

Now let's switch the order of these transforms—that is, `transform: translate(30px, 50px) scale(2)`. The result is shown below.

Notice that our object is still scaled by a factor of two, but now it's offset by 30 pixels horizontally and 50 pixels vertically. Expressed using the `matrix()` function, this is `transform: matrix(2, 0, 0, 2, 30, 50)`.

It's also worth noting that inherited transforms function similarly to transform lists. Each child transform is multiplied by any transform applied to its parent. For example, take the following code:

```
<div style="transform: skewX(25deg)">
    <p style="transform: rotate(-15deg)"></p>
</div>
```

This is rendered the same as the following:

```
<div>
    <p style="transform: skewX(25deg) rotate(-15deg)"></p>
</div>
```

The current transform matrix of the `<p>` element will be the same in both cases. Though we've focused on 2D transforms so far, the above also applies to 3D transforms. The third dimension adds the illusion of depth. It also brings some additional complexity in the form of new functions and properties.

3D Transform Functions

There are nine functions for creating 3D transforms. Each of these functions modifies the Z-coordinates of an element and/or its children, in addition to the X- and Y-coordinates. Remember, Z-coordinates are points along the plane that sit perpendicular to the viewer. With the exception of `rotateZ()`, these functions create and change the illusion of depth on screen.

`rotateX()` and `rotateY()`

The `rotateX()` and `rotateY()` functions rotate an element around the X- and Y-axes respectively. Using `rotateX()` creates a somersault effect, causing an object to flip top-over-tail around a horizontal axis. With `rotateY()`, the effect is more like that of a spinning top, rotating around a vertical axis.

Like `rotate()`, both `rotateX()` and `rotateY()` accept an angle measurement as an argument. This angle can be expressed in degrees (`deg`), radians (`rad`), gradians (`grad`), or turn (`turn`) units. As mentioned earlier in the chapter, `rotateZ()` works the same way as `rotate()`. It's a relic from when 2D and 3D transforms were defined by separate specifications.

Positive angle values for `rotateX()` cause an element to tilt backwards, as shown in the image below, where `transform: rotate(45deg)` is applied.

Negative angle values for `rotateX()` do the opposite, causing the element to tilt forward, as shown below, where `transform: rotate(-45deg)` is applied.

Negative angles for `rotateY()` cause the element to tilt counterclockwise. In the image below, the element is rotated -55 degrees around the Y-axis (`transform: rotateY(-55deg)`).

Positive values tilt it clockwise, as shown below (`transform: rotateY(55deg)`).

As an aside, the three images above have a `perspective` value of `200px`. We'll discuss the `perspective` property later in this chapter. For now, it's enough to know that this property adds a sense of depth and exaggerates the effect of the three-dimensional rotation. Compare the image above to the image below. Both have been rotated along the Y-axis by 55 degrees (`transform: rotateY(55deg)`), but in the image below, the parent container has a `perspective` value of `none`. Our object looks more squished than rotated. Use `perspective` on a container element when creating a 3D transform.

Disappearing Elements

There’s another issue to be aware of when working with 3D rotations. Rotating an element by plus or minus 90 degrees, or plus or minus 270 degrees, can sometimes cause it to disappear from the screen. Each element on a page has an infinitesimal thickness. By rotating it a quarter or three-quarters of a turn, we’re looking at its infinitesimally thin side. It’s kind of like looking at the edge of a sheet of paper that’s perpendicular to your face. Adjusting the `perspective` and `perspective-origin` values of a parent element can prevent this behavior in some cases, but not all of them.

Rotating around Multiple Axes with `rotate3d()`

Sometimes we want to rotate an object around more than one axis. Perhaps you want to rotate an element counterclockwise and tilt it by 45 degrees, as in the image below, where our object is rotated around both the X- and Y-axes by 45 degrees.

This is what `rotate3d()` does. It's a function that accepts four arguments. The first three make up an X, Y and Z direction vector, and each of these should be a number. The fourth argument for `rotate3d()` should be an angle. The transformed object will be rotated by the angle around the direction vector defined by the first three arguments.

What those first three numbers *are* matters less than the ratio between them. For example, `transform: rotate3d(100,5,0,15deg);` and `transform: rotate3d(20,1,0,15deg);` have equivalent 3D matrices and produce the same effect.

That said, due to way the [`rotate3d` matrix gets calculated](), a declaration such as `transform: rotate3d(1, 500, 0, 15deg);` won't produce an effect significantly different from `transform: rotate3d(1, 1, 0, 15deg);`.

Just about any non-zero value for any of the first three parameters creates a tilt along that axis. Zero values prevent a tilt. As you may have guessed, `rotateX(45deg)` is the equivalent of `rotate3d(1, 0, 0, 45deg)`, and `rotateY(25deg)` could also be written as `rotate3d(0, 1, 0, 25deg)`.

If the first three arguments are `0` (such as `transform: rotate3d(0, 0, 0, 45deg)`), the element won't be transformed. Using negative numbers for the X, Y, or Z vector arguments is valid; it will just negate the value of the angle. In other words, `rotate3d(-1, 0, 0, 45deg)` is equivalent to `rotate3d(1, 0, 0, -45deg)`.

Using `rotate3d()` rotates an element by the given angle along multiple axes at once. If you want to rotate an element by different angles around multiple axes, you should use `rotateX()`, `rotateY()`, and `rotate()` or `rotateZ()` separately.

The `perspective()` Function

The `perspective()` function controls the foreshortening of an object when one end is tilted towards the viewer. **Foreshortening** is a specific way of drawing perspective—that is, simulating three dimensions when you only have two dimensions. With foreshortening, the ends of objects that are tilted towards the viewer appear larger, and the ends furthest from the viewer appear smaller. Foreshortening mimics the distortion that occurs when you view an object up close versus viewing it at a distance.

The more technical definition, pulled from the CSS Transforms Module Level 2 specification, says that `perspective()` "specifies a perspective projection matrix." The definition continues:

> This matrix scales points in X and Y based on their Z value, scaling points with positive Z values away from the origin, and those with negative Z values towards the origin. Points on the Z=0 plane are unchanged.

In practice, this means that `perspective()` will have a visible effect only when some of an object's points have a non-zero Z-coordinate. Use it with another 3D function in a transform list (for example, `transform: perspective(400px) rotateX(45deg)`), or apply it to the child of a transformed parent.

The `perspective()` function accepts a single argument. This argument must be a length greater than zero. Negative values are invalid, and the transform won't be applied. Lower values create a more exaggerated foreshortening effect, as you can see below. In this image, the value of our `transform` is `perspective(10px) rotate3d(1,1,1,-45deg)`.

Higher values create a moderate amount of foreshortening. The next image illustrates the impact of a higher perspective value. Its `transform` property value is `perspective(500px) rotate3d(1,1,1,-45deg)`.

Order *really* matters when working with the `perspective()` function. A good rule of thumb is to list it first, as we've done in the examples here. You *can* list it elsewhere in the transform list (for example, `rotate3d(1,0,1,-45deg) perspective(100px)`), but the resulting current transform matrix doesn't create much of an effect.

There's also a point of diminishing returns with the `perspective()` function (and with the `perspective` property, as well). Increasing the argument's value beyond a certain threshold will create little difference in how the element and its children are painted to the screen.

`perspective()` versus `perspective`

A word of caution: the transforms specification defines both a `perspective()` function and a `perspective` property. Though both are used to calculate the perspective matrix, they're used differently. The `perspective` property affects—and must be applied to—the containing element. It sets an imaginary distance between the viewer and the stage. The `perspective()` function, on the other hand, can be applied to elements as part of a `transform` list.

Translating Depth with `translateZ()` and `translate3d()`

Earlier in this chapter, we discussed how to translate an element horizontally or vertically using `translateX()` and `translateY()`. However, we can also translate along the Z-axis. There are two functions that allow us to do this: `translateZ()` and `translate3d()`. We can combine them with transitions to create zoom effects, or mimic the feeling of moving through a chute.

The `translateZ()` function accepts a single length parameter as its argument. Length units are the only valid units for this function. Remember that we're projecting three-dimensional coordinates into a two-dimensional space, so percentages don't make much sense. The `translateZ()` function shifts the object towards or away from the user by the specified length. Negative values shift the element or group away from the user—in effect shrinking it—as can be seen below with `transform: translateZ(-150px)`.

Positive values shift the element towards the viewer, making it appear larger. Sometimes the effect is to fill the entire viewport, thereby engulfing the viewer, as seen below with `transform: translateZ(150px).`

If the value of `translateZ()` is large enough, the element disappears from view. That's because it's moved behind the viewer in this imagined 3D space. Similarly, if the value of `translateZ()` is small enough—say `translateZ(-40000px)`—the element will disappear from view because it's now "too far" from the viewer and too small to draw on screen.

`translate3d()` is a more concise way of translating in two or three directions at once. It accepts three arguments: one each for the X, Y, and Z directions. Translation values for the X and Y direction arguments may be lengths or percentages, but the Z direction argument (the third argument) must be a length value. Keep in mind that `translateX(50%) translateY(10%) translateZ(100px)` is the equivalent of `translate3d(50%, 10%, 100px)`. Use `translate3d()` when you want to translate more than one dimension and you also want more concise code.

Scaling the Z-dimension: `scaleZ()` and `scale3d()`

We can also scale an object's Z-dimension using the `scaleZ()` and `scale3d()` functions. The `scaleZ()` function transforms points along the Z-axis alone, while `scale3d()` lets us scale all three dimensions at once. Scaling the Z-dimension changes the depth of an object, and in some combinations can be used to create zoom effects. Experiment with them and see.

The `scaleZ()` function accepts a number as its argument. As with `scaleX()` and `scaleY()`, positive values greater than `1` increase the size of the element's Z-dimension. Values between 0 and 1 decrease its size. Negative values between 0 and -1 decrease the element's size along the Z-dimension, while values less than -1 increase it. Since these values are negative, however, the element and its children are inverted. In the image below, the left die shows an element group with `transform: scaleZ(0.5)` applied. The box on the right has a transformation of `scaleZ(-0.5)` applied. Notice that the positions of the six face and one face have been swapped in the example with a negative scale.

The `scale3d()` function accepts three arguments—all of which are required in order for this function to work. The first argument scales the X-dimension. The second argument scales its Y-dimension, and the third argument scales the Z-dimension. As with `translate3d()`, the `scale3d()` function is just a more concise way to write transforms that scale in multiple dimensions. Rather than using `scaleX(1.2) scaleY(5) scaleZ(2)`, for example, you could use `scale3d(1.2, 5, 2)`.

Transform functions are only part of what you need to create 3D transforms. You’ll also need CSS properties that manage how objects are drawn in a simulated three-dimensional space. These properties affect the perception of depth and distance.

Creating Depth with the `perspective` Property

To make a 3D-transformed object look like it's sitting in a three-dimensional space, we need the `perspective` property. The `perspective` property adjusts the distance between the drawing plane and the viewer. We're still projecting three-dimensional coordinates into a two-dimensional space. But adding `perspective` to a containing element causes its children to have the appearance of being in a 3D space.

As with `transform`, `perspective` creates both a new containing block and a new stacking context when the value is something other than `none`. Along with the `perspective-origin` property, `perspective` is used to calculate the perspective matrix. We'll cover `perspective-origin` in the next section.

Safari and UC Browser

Use a `-webkit-` prefix for `perspective` and `perspective-origin` to support users of UC Browser and Safari 8 and under (`-webkit-perspective` and `-webkit-perspective-origin`).

In addition to the `none` keyword, `perspective` also accepts a length as its value. Values must be positive (such as `200px` or `10em`). Percentages don't work. Neither do negative values such as `-20px`.

Smaller values for `perspective` increase the visual size of the element, as seen below, which has a `perspective` value of `500px`. Items that are closer to the viewer on the Z-axis appear larger than those further away.

Larger values, on the other hand, make elements appear smaller. The container element pictured below has a `perspective` value of `2000px`. This is similar to how your eye perceives objects of varying distances.

Modifying the Point of View with `perspective-origin`

If you’ve ever studied how to draw in perspective, the `perspective-origin` property will feel like old hat. To draw in perspective, you first make a point on your page or canvas. This point is known as the **vanishing point**. It’s the point in your drawing at which items will theoretically disappear from view.

Next, draw a shape of your choosing. We’ll keep this example simple by using a rectangle.

Step three is to draw a series of lines towards the vanishing point, as shown in the image below. These lines, also known as **convergence lines**, serve as guides for drawing shapes that are sized appropriately given their perceived distance from the viewer.

As you can see in the following image, the rectangles that appear closer to the viewer are larger. Those that appear further away are smaller.

This is essentially how the `perspective-origin` property works. It sets the coordinates of the vanishing point for the stage. Negative Y values give the impression that the viewer is looking down at the stage, while positive ones imply looking up from below it. Negative X values mimic the effect of looking from the right of the stage. Positive X values mimic looking from its left. The following image shows a containing element with a `perspective-origin` of `-50% -50%`.

As with `transform-origin`, the initial value of `perspective-origin` is `50% 50%`—the center point of the containing element. Values for `perspective-origin` may be lengths or percentages.

Positioning keywords—`left`, `right`, `top`, `bottom`, and `center`—are also valid. The `center` keyword is the same as `50% 50%`. Both `bottom` and `right` compute to positions of `100%` along the vertical and horizontal positions respectively. The `top` and `left` keywords compute to vertical and horizontal positions of `0%`. In all cases, `perspective-origin` is an offset from the top-left corner of the container.

Preserving Three Dimensions with `transform-style`

As you work with 3D transforms, you may stumble across a scenario in which your transforms fail to work—or they work, but only for one element. This is caused by grouping property values. Some combinations of CSS properties and values require the browser to flatten the representation of child elements before the property is applied. These include `opacity` when the value is less than `1` and `overflow` when the value is something other than `visible`.

Here's the counterintuitive part: `transform` and `perspective` also trigger this flattening when their value is something other than `none`. In effect, this means that child elements stack according to their source order if they have the same z-index value, regardless of the transform applied. Consider the following source:

```
<div class="wrapper">
    <figure>a</figure>
    <figure>f</figure>
</div>
```

And the following CSS:

```
.wrapper {
    perspective: 2000px;
    perspective-origin: 50% -200px;
}
.wrapper figure {
    position: absolute;
    top: 0;
    width: 200px;
    height: 200px;
}
.wrapper figure:first-child {
    transform: rotateY(60deg) translateZ(191px);
    background: #3f51b5;
}
.wrapper figure:nth-child(2) {
    transform: rotateY(120deg) translateZ(191px);
    background: #8bc34a;
}
```

In this example, since we've applied `perspective: 1000px` to `.wrapper`, our `<figure>` elements are flattened. Since both elements also have the same calculated `z-index`, `.wrapper figure:nth-child(2)` will be the topmost element in the stack, as seen in the following image.

Note that `.wrapper figure:first-child` is still visible. It's just not the topmost element. Here the computed value of `transform-style` is `flat`.

To work around this, we set the value of `transform-style` to `preserve-3d`. Let's update our CSS:

```
.wrapper {
    perspective: 2000px;
    perspective-origin: 50% -200px;
    transform-style: preserve-3d;
}
.wrapper figure {
    position: absolute;
    top: 0;
    width: 200px;
    height: 200px;
}
.wrapper figure:first-child {
    transform: rotateY(60deg) translateZ(191px);
    background: #3f51b5;
}
.wrapper figure:nth-child(2) {
    transform: rotateY(120deg) translateZ(191px);
    background: #8bc34a;
}
```

Now `.wrapper figure:first-child` becomes the topmost element, as our `rotateY()` functions suggest it should be in the image below.

In the vast majority of cases, you should use `transform-style: preserve-3d`. Use `transform-style: flat` only when you want to collapse child elements into the same layer as their parent.

Transform Style

The WebKit team's Transform Style demo shows the effect of `transform-style: flat`. It's an old demo that was created for WebKit browsers, so you'll need to use a WebKit- or Blink-based browser such as Safari, Chrome, Opera, or Samsung Internet to view it.

Showing Both Faces with the `backface-visibility` Property

By default, the back face of an element is a mirror image of its front face. With stacked or overlapping elements, the reverse side is always visible to the viewer, regardless of which side sits at the top of the stack.

Sometimes, however, we don’t want this back side to be visible. Let’s return to the card metaphor mentioned in the introduction to this chapter. This time we’ll use the playing card pictured below.

With any card, we only want one side to be visible to the user at a time. To manage the visibility of an object's back side, we can use the `backface-visibility` property.

The initial value of `backface-visibility` is `visible`. Rear faces will always be shown. But if we want to hide a visible back face, we can use `backface-visibility: hidden` instead.

Let's create our playing card. First our HTML:

```
<div class="card">
    <div class="side front">
        <div class="suit">&clubs;</div>
    </div>
    <div class="side back"></div>
</div>
```

In the markup above, we've set up front and back sides for a `card` container. Here's our card CSS:

```
.card {
    border: 1px solid #ccc;
    height: 300px;
    position: relative;
    transition: transform 1s linear;
    transform-style: preserve-3d;
    width: 240px;
}
```

The important part to notice here is `transform-style: preserve-3d`. Again, we'll need this property to prevent the flattening that occurs by default when we use the `transform` property. Now let's set up the CSS for the front and back sides of our cards:

```
/* Applies to both child div elements */
.side {
    height: inherit;
    left: 0;
    position: absolute;
    top: 0;
    width: inherit;
}
.front {
    transform: rotateY(180deg);
}
.back {
    background: rgba(204, 204, 204, 0.8);
}
.suit {
    line-height: 1;
    text-align: center;
    font-size: 300px;
}
```

Both sides are absolutely positioned, so they'll stack according to their source order. We've also flipped the `.front` sides around the Y-axis by 180 degrees. When it's all put together, your card should look a bit like the one pictured below.

Both sides of the card are visible at the same time. Let’s revise our CSS slightly. We’ll add `backface-visibility: hidden` to our `.side` rule set:

```
.side {
    backface-visibility: hidden;
    height: inherit;
    left: 0;
    position: absolute;
    top: 0;
    width: inherit;
}
```

Now, `div.front` is hidden. If you see a gray box and no club symbol, it's working as expected.

Prefixes for Safari

Safari versions 12 and earlier require the `-webkit-` vendor prefix for the `backface-visibility` property.

The utility of `backface-visibility: hidden` becomes a little clearer when we flip `div.card`. Let's add a `.flipped` class to our CSS:

```
.flipped {
    transform: rotateY(180deg);
}
```

Now when we flip our card over (pictured below), we see `div.front`, and only `div.front`.

The image below shows two cards before being flipped. The card on the left has a `backface-visibility` value of `hidden`, while the one on the right has a value of `visible`.

And in the next image, we can see these same cards after the `flipped` class is added—that is, `<div class="card flipped">`.

Conclusion

Whew! That was a lot to take in. I hope that, after reading this chapter, you've learned how to:

- affect page layout and stacking order with transforms
- calculate the current transform matrix
- apply 2D transform functions that rotate, translate, and skew objects
- use 3D transforms to create the illusion of depth and dimension

In the next chapter, we'll look at CSS visual effects—including gradients, blend modes, and filters.

Chapter 9: Visual Effects

CSS includes several features for creating the kinds of visual effects that used to require image-editing software such as Photoshop. Filter effects give us the ability to add true drop shadows, and to blur or desaturate images, while blend modes let us combine layers.

In this chapter, we'll take a look at three of these features:

- the `background-blend-mode` and `mix-blend-mode` properties of the Compositing and Blending specification
- filter effects
- masking and clipping

Browsers have long supported these features—fully or in part. Adoption of them by CSS authors has been a bit slower, however, presumably because many still use image editing software to create these effects. Using CSS, however, gives us flexibility: changing a header image or creating a new icon variant doesn't require your design team to create a new asset.

Let's dig in!

Blend Modes

Blend modes make it possible to combine background colors and images using effects commonly found in graphics software such as Photoshop. Defined modes include `multiply`, `screen`, `overlay`, and `color-dodge`.

"Blend modes" is the colloquial name for the `background-blend-mode` and `mix-blend-mode` properties. These properties are outlined in the Compositing and Blending specification. Although the CSS Working Group manages most CSS specifications, the World Wide Web Consortium's CSS-SVG Effects Task Force manages this one. Blend modes apply to HTML and SVG elements. Here we'll focus on using them with HTML.

Blend modes describe how to visually combine layers. Both `mix-blend-mode` and `background-blend-mode` accept the same values, but work slightly differently:

- `mix-blend-mode` blends *foreground* layers with layers that they overlap.
- `background-blend-mode` mixes *background* images and colors. It accepts a single mode or a list of modes, and this list gets applied in the same order as `background-image`.

Let's look at some examples.

mix-blend-mode

As mentioned above, `mix-blend-mode` merges foreground layers with layers that are stacked below it. A layer may be a background color or image for a parent element, for the root element, or elements that sit beneath the targeted element in the stack. Consider the following HTML and CSS:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>mix-blend-mode</title>
    <link rel="stylesheet" href="mix-blend-mode.css">
</head>
<body>
    <div>
        <p>This is a paragraph that's positioned below the image in the stack. </p>
        <img src="dahlia.jpg">
    </div>
```

```
</body>
</html>
```

Our `mix-blend-mode.css` file looks like this:

```
div {
    background-image: linear-gradient(to left, #f30, #fc0);
    width: 60vw;
    margin: 3rem auto;
    position: relative;
}
p {
    position: absolute;
    color: black;
    margin: 4rem;
    font-size: 6.4rem;
}
img {
    display: block;
    height: auto;
    width: 100%;
}
```

The image below shows the output of the code we have so far.

Photo by dewdrop157 from Pixabay.

Contrast Issues

Notice that the text pictured above is hard to read. That's because there's insufficient contrast between the text and the image. In a real-world project, you should add a background color or gradient to the `<p>` element to make it legible. Also ensure that there's sufficient contrast between foreground and background colors.

Since the `<p>` element is absolutely positioned, the text overlays the photo. Notice too that the photo fills the entire width and height of its parent container so we can't see its background gradient. Now let's add `mix-blend-mode` to the `img` rule set. We'll use the `difference` mode:

```
img {
    display: block;
    height: auto;
    width: 100%;

    mix-blend-mode: difference;
}
```

Now the photograph has moved to the top of the layer stack (as pictured below), but its pixels have been blended with the paragraph's text and the background image of its parent `<div>`.

Adding `mix-blend-mode: difference` causes the photograph, text, and background of its parent to blend in a specific way. Using a value other than `normal` for `mix-blend-mode` creates a new stacking context for the element. The new stack or group then gets blended and composited with the stacking context that contains the element.

Stacking

Refer to Chapter 5, "Layouts", for a refresher on stacking contexts.

Although this looks like chaos on screen, it's predictable once you understand how each blend mode works. Blend modes use an RGB color system, where the red, green, and blue component of each color is expressed as a value between 0 and 1. We can convert from colors expressed using the CSS `rgb`/`rgba` functions by dividing each channel's value by 255. For example, `rgb(255, 0, 255)` becomes `RGB(1, 0, 1)`, and `rgb(52, 62, 28)` becomes `RGB(0.203, 0.243, 0.109)`.

With the exception of `normal`, each blend mode uses a mathematical formula to determine what color to paint each pixel within the overlapping region (`normal` indicates that no blending should occur). The specification outlines how to calculate the output for each blend mode. For `difference`, that formula is:

```
pixel color = bottom layer color - top layer color
```

In other words, **pixel color** is the absolute value of the difference between the red, blue, and green channels of the bottom and top layers.

Layer Names

I've used the terms "bottom layer" and "top layer" for clarity. The specification calls the bottom layer the "destination layer" or "backdrop", and it calls the top layer the "source layer".

Let's return to the photo of dahlias in the example above. The color of the pixel at (0,0) of the photograph is `rgb(185, 40, 58)`. If we express each channel as a value between 0 and 1, this becomes `RGB(0.725, 0.157, 0.227)`.

The color of the pixel at (0,0) in the background gradient of `div` is `rgb(255, 204, 0)`, or `RGB(1, 0.8, 0)`. Let's calculate the difference between each channel:

```
RED:   1.0 - 0.725 =  0.275
GREEN: 0.8 - 0.157 =  0.643
BLUE:  0.0 - 0.227 = -0.227
```

The color at (0,0) of our blended layers is `RGB(0.275, 0.643, 0.227)` or `rgb(70, 164, 58)`. When used with `difference`, black doesn't produce a change in pixel color; its red, blue, and green channels are all zero (`rgb(0,0,0)`). White, on the other hand, inverts the color of a pixel.

Taming Blend Modes

To learn more about how the `difference` and `exclusion` blend modes work, read Ana Tudor's article "Taming Blend Modes: `difference` and `exclusion`".

Safari versions 14.0.2 and under behave a little differently from other browsers. When the value of `background-image` or `background-color` isn't explicitly set, those versions of Safari use white or `rgb(255,255,255)` as the initial value. Firefox, Chrome, and Edge, on the other hand, use `rgba(0,0,0,0)`. This can lead to unexpected results for some blend modes. Let's look at an example using `exclusion`.

The `exclusion` blend mode is similar to the `difference` blend mode, with a slightly lower level of contrast. Here's the formula for calculating an exclusion blend mode:

```
pixel color = bottom layer + top layer - 2 × bottom layer × top layer
```

Say we have a pixel with color `rgb(128, 177, 230)`. If we express each channel as a proportion of 255, this becomes `RGB(0.502, 0.694, 0.902)`. First, we'll blend it with black, or `RGB(0, 0, 0)`:

```
RED:   0 + 0.502 - 2 × 0 × 0.502 = 0.502
GREEN: 0 + 0.694 - 2 × 0 × 0.694 = 0.694
BLUE:  0 + 0.902 - 2 × 0 × 0.902 = 0.902
```

Our pixel color remains unchanged at `RGB(0.502, 0.694, 0.902)` or `rgb(128, 177, 230)`. But if we blend it with white, or `RGB(1, 1, 1)`, something else happens:

```
RED:   1 + 0.502 - 2 × 1 × 0.502 = 0.498
GREEN: 1 + 0.694 - 2 × 1 × 0.694 = 0.306
BLUE:  1 + 0.902 - 2 × 1 × 0.902 = 0.098
```

Our blended pixel color is `RGB(0.498, 0.306, 0.098)` or `rgb(127, 78, 25)`.

In the image below, the picture on the left reflects what happens when you use `mix-blend-mode: exclusion` and the initial value of `background-color` in Firefox, Chrome, and Edge. The picture on the right reflects Safari's behavior. Adding `background-color: white` to the element's parent ensures consistent behavior across browsers.

Photo by Aaron Burden from Unsplash.

`background-blend-mode`

As mentioned at the beginning of this section, the `background-blend-mode` property combines background colors and images. It accepts a mode or a list of modes. Arguments follow the ordering of `background-image`, meaning that `background-blend-mode` can accept a single blend mode or a list of blend modes.

The first image listed in a `background-image` list is the topmost layer, drawn closest to the user. Other images in the list stack towards the bottom in the order in which they're listed. In the following example, our `background-image` list contains two values. However, because the top gradient uses opaque colors, it's the only background image that's visible:

```
body {
    background-image: linear-gradient( 90deg, rgb(255,0,0), rgb(0,0,255)),
                      linear-gradient(270deg, rgb(127,255,0), rgb(255,255,0));
}
```

The following image illustrates that just the topmost layer is visible, which is the first image listed.

Each item in a `background-blend-mode` list gets applied in the same order. If there's only one item in the list, it's repeated for every item in the `background-image` list. Let's add a blend mode to the CSS above:

```
body {
    background-image: linear-gradient( 90deg, rgb(255,0,0), rgb(0,0,255)),
                      linear-gradient(270deg, rgb(127,255,0), rgb(255,255,0));
    background-blend-mode: multiply;
}
```

Since `background-blend-mode` has one item in its list, the browser will use `multiply` for both background images.

The `multiply` blend mode sets the color of a pixel to the product of the top and bottom layers. In this case, the pixel at (0,0) is the product of red (`RGB(1,0,0)`) and yellow (`RGB(1,1,0)`), which works out to `RGB(1,0,0)`, as illustrated below. Multiplying our gradient layers creates a gradient that transitions from red to black.

Let’s look at another example using multiple backgrounds and multiple blend modes:

```
div {
    height: 733px;
    width: 1152px;
    background-size: 100% auto;
    background-repeat: no-repeat;
    background-position: 0 0, -70px -140px, center right;

    /* List of multiple backgrounds */
    background-image: linear-gradient(to bottom, #f90d, #f30d, #f0fd 105%),
                      url('strawberry-trio.jpg'),
                      url('strawberry-and-milk.jpg');
    background-blend-mode: saturation, darken, screen;
}
```

In this example, we’ve combined a gradient and two photographs of strawberries. The following image shows the separate layers.

Strawberry image (center) by [Hal Gatewood] from Unsplash. Strawberry on spoon image by [wallner] from Pixabay.

Each layer has a different blend mode. The `saturation` mode applies to the gradient layer, while `saturation` and `darken` apply to the `strawberry-trio.jpg` and `strawberry-and-milk.jpg` layers respectively. You can see the result below.

You may have noticed here that we haven't defined a background color. Let's change that:

```
div {
    height: 733px;
    width: 1152px;
    background-size: 100% auto;
    background-repeat: no-repeat;
    background-position: 0 0, -70px -140px, center right;

    background-image: linear-gradient(to bottom, #f90d, #f30d, #f0fd 105%),
                      url('strawberry-trio.jpg'),
                      url('strawberry-and-milk.jpg');
    background-blend-mode: saturation, darken, screen;

    background-color: green; /* Now we have a background color */
}
```

Now our element has a greenish tint, as you can see below. If the element has a background color, that layer becomes part of the blend.

Blend modes are limited to a local stacking context. Adding a background color to an ancestor of `div` would not affect the blending result of `div`.

Blend Mode Values

There are sixteen values that you can use with the `mix-blend-mode` and `background-blend-mode` properties. The following image is a partly transparent PNG image of a lemon set against a background gradient of its container that transitions from magenta to purple to blue, with no blend mode applied.

Sliced lemon photo by Louis Hansel from Unsplash.

The following table illustrates the result of each `mix-blend-mode` value when added to the `img` element.

Property	Behavior	Effect

Property	Behavior	Effect
`color`	Creates a color with the hue and saturation of the source or top layer color and the luminosity of the backdrop or bottom layer color, preserving the gray levels of the backdrop	

Property	Behavior	Effect
`color-burn`	Darkens the color of the bottom layer's pixel to reflect that of the top or source layer. (White produces no change)	

Property	Behavior	Effect
`color-dodge`	Brightens the color of the bottom layer’s pixel to reflect that of the top or source layer. (Black produces no change)	

Property	Behavior	Effect
`darken`	Replaces the bottom layer’s color with that of the top layer when the top or source layer is darker	

Property	Behavior	Effect
`difference`	The absolute value of the difference between the bottom layer’s color and the top layer’s color	

Property	Behavior	Effect
`exclusion`	Similar to `difference` in appearance, but creates a lower contrast	

Property	Behavior	Effect
`hard-light`	When the top layer's color value is less than or equal to 0.5, it multiplies the colors. Otherwise, it screens them. Produces an effect that's similar to shining a harsh spotlight on the bottom layer or backdrop color	

Property	Behavior	Effect
`hue`	Creates a color with the hue of the top (or source) layer’s color and the saturation and luminosity of the bottom layer or backdrop color	

Property	Behavior	Effect
`lighten`	Replaces the bottom layer’s color with that of the top layer when the top or source layer is lighter	

Property	Behavior	Effect
`luminosity`	Creates a color with the luminosity of the source (or top layer) color and the hue and saturation of the backdrop color. The effect is the inverse of `color`	

Property	Behavior	Effect
`multiply`	Multiplies the top layer by the bottom layer and replaces the color in the bottom layer (or destination)	

Property	Behavior	Effect
`overlay`	Colors in the top layer are mixed with those in the bottom layer or backdrop while preserving the backdrop’s highlights and shadows. The inverse of `hard-light`	

Property	Behavior	Effect
`saturation`	Creates a color with the saturation of the top layer’s color and the hue and luminosity of the bottom layer’s color. When an area of a backdrop is pure gray (such as saturation = 0), the colors don’t change	

Property	Behavior	Effect
`screen`	Multiplies the complements of the bottom layer and top layer values, and complements the product	

Property	Behavior	Effect
`soft-light`	Darkens or lightens based on the color of the top or source layer. Channel values less than or equal to 0.5 tend to darken source colors. Channel values greater than 0.5 lighten them. Produces an effect that's similar to shining a diffused spotlight on the bottom layer or backdrop color	

Refer to the specification for the formulas used to calculate the pixel color for each blend mode.

Filter Effects

With filter effects, we can blur objects, change them from full color to grayscale or sepia tone, modify their hue, and invert their colors. As with blend modes, CSS filter effects can be used with HTML or SVG documents. Most examples in this section use HTML elements.

Filter effects have two parts:

- **filter primitives**, which are SVG tags that must be used as children of the SVG `filter` element
- **filter functions**, which are used with the CSS `filter` and `backdrop-filter` properties

Most filter functions have a filter primitives equivalent, but not every filter primitive has an equivalent filter function. We'll focus on filter functions in this section, since they're most applicable to CSS.

SVG Primitives

SVG primitives are beyond the scope of this book. If you'd like to learn more, though, the Mozilla Developer Network has an excellent SVG element reference that includes documentation about filter primitive elements.

Filter functions can be used with the `filter` or `backdrop-filter` properties. The `filter` property affects foreground elements, while `backdrop-filter` affects the layers *behind* the element to which it's applied.

You'll see examples of both in this section. First, let's meet the filter functions.

Meet the CSS Filter Functions

There are 12 filter functions, each of which has an SVG filter primitive counterpart. Both `filter` and `backdrop-filter` accept these functions as part of their filter function list. The table below shows how each filter function affects the following photograph.

Keel-billed toucan photo by Zdeněk Macháček from Unsplash.

Function name	Effect	Initial value	Result
`blur()`	Applies a Gaussian blur to the input image or element. Must be a length value.	`0px`	

Function name	Effect	Initial value	Result
`brightness()`	Uses a linear multiplier to make an image or element appear brighter or dimmer, and its value must be greater than 0. Values may be a percentage or a decimal value. 0 or 0% turns the layer completely black. 1 or 100% creates no change. Values greater than 1 or 100% create an image that's brighter than its input.	1 or 100%	

Function name	Effect	Initial value	Result
`contrast()`	Changes the contrast of the input. Values may be a percentage or a decimal value. 0 or 0% turns the image gray. 1 or 100% leaves it unchanged. Values greater than 100% or 1 create increasing amounts of contrast.	1 or 100%	
`drop-shadow()`	Applies a Gaussian blur drop shadow to the element. Arguments are similar to those of `box-shadow`, but instead of the third argument being the blur radius, it's the standard deviation (<x-offset> <y-offset> <standard	0 0 0 transparent	

Function name	Effect	Initial value	Result
	deviation> <shadow color>).		

Function name	Effect	Initial value	Result
`grayscale()`	Converts the image to grayscale or black and white. Values may be a percentage or a decimal value. Accepts a value between 0 and 1 or 0% and 100%. Values greater than 100% are permitted, but are clamped to 100%.	1 or 100%	

Function name	Effect	Initial value	Result
`hue-rotate()`	Accepts an angle measurement in degree, radian, gradian, or turn units. An angle of 0 degrees or multiples of 360 degrees don’t create a visual change. However, animating to or from a multiple of 360 changes the hue a sufficient number of times.	0deg	

Function name	Effect	Initial value	Result
`invert()`	Accepts a value between 0 and 1 or 0% and 100%. Values may be a percentage or a decimal value. Values greater than 100% are permitted, but are clamped to 100%.	1 or 100%	

Function name	Effect	Initial value	Result
`opacity()`	Changes the transparency of an element. Values may be a percentage or a decimal value. Accepts a value between 0 and 1 or 0% and 100%. Values greater than 100% are permitted, but are clamped to 100%. Note that this isn’t an alias or alternative to the `opacity` property. They’re separate properties. Using both will increase the transparency of the target object.	1 or 100%	

Function name	Effect	Initial value	Result
`saturate()`	Affects the saturation or color intensity of a layer. Values may be a percentage or a decimal value. 0 or 0% is completely unsaturated, which is typically rendered as gray. 100% creates no change. Values greater than 1 or 100% increase saturation, creating super-saturated or over-saturated layers.	1 or 100%	

Function name	Effect	Initial value	Result
`sepia()`	Converts the image to sepia tones. Values may be a percentage or a decimal value. Accepts a value between 0 and 1 or 0% and 100%. Values greater than 100% are permitted, but are clamped to 100%.	1 or 100%	

Filter functions can be used alone or in combination. For instance, we can combine the sepia and drop shadow filters:

```
img {
    filter: sepia( 1 ) drop-shadow( 3px 3px 8px #0da8cc );
}
```

This creates the effect shown in the image below.

I should probably mention here that the `drop-shadow()` function works differently from the `box-shadow` property, despite its similar syntax. Firstly, the drop shadow filter *doesn't* accept a list of shadows. Passing a shadow list, as in the example below, won't work. The browser will ignore your entire filter rule:

```
div {
    /* Unsupported. Will not work */
    filter: drop-shadow(0px 0px 3px #0c0, -2px -2px 3px #333 );
}
```

Secondly, the `drop-shadow` function, as the specification explains, is "a blurred, offset version of the input image's alpha mask drawn in a particular color, composited below the image". In other words, when a layer contains transparent areas, the drop shadow filter follows the contours of the transparency. It doesn't create a shadow for the element's box.

The image above illustrates this difference. In both cases, these images are PNG images with 100% alpha transparency. The image on the left uses `filter: drop-shadow()`, and the shadow follows the shape of the lemon. In the image on the right, the shadow follows the dimensions of the image.

Using `backdrop-filter`

Where `filter` affects foreground elements, `backdrop-filter` affects elements or layers that sit behind the element to which `backdrop-filter` is applied. This can be a layer created using `background-image`, or a positioned sibling element.

Browser support for `backdrop-filter` is less robust than for `filter`. Major Chromium-based browsers—including Chrome, Edge, Opera, and Samsung Internet—support it by default. However, Safari still requires a `-webkit-` prefix (such as `-webkit-backdrop-filter`).

Firefox supports `backdrop-filter` without a vendor prefix. At the time of writing (and confirmed in Firefox 91 and below), that support is still considered experimental and needs to be enabled. To tinker with this feature in Firefox, edit your `about:config` settings and change the `layout.css.backdrop-filter.enabled` and `gfx.webrender.all` settings to `true`.

In the image below, we see a paragraph with `backdrop-filter` applied. The `backdrop-filter` property affects the background image layer, not the paragraph itself.

Flamingo photo by [Alejandro Contreras](#) from Unsplash.

Take a look at the flamingo image above. Notice that the area of the flamingo that sits beneath the pink-colored box is both blurred and blue, but the text isn't. Here's the CSS to create that effect:

```
div {
    background-image: url('flamingo.jpg');
    background-size: contain;
    width: 600px;
    height: 600px;
    display: flex;
    align-items: center;
    justify-content: center;
    margin: auto;
}
p {
    margin: 0;
    align-items: center;
    padding: 6rem 2rem;
  /*
   * Set a background color as a fallback.
   */
    background: hsla( 22.4, 44.9%, 63.7%, .9 );
}
@supports (
    ( backdrop-filter: blur( 8px ) hue-rotate( 180deg ) ) or
    ( -webkit-backdrop-filter: blur( 8px ) hue-rotate( 180deg ) )
) {
    p {
      /*
       * Undo the background color. The initial value of `background-color` is
       * `transparent`.
       */
      background-color: initial;
      backdrop-filter: blur( 8px ) hue-rotate( 180deg );
    }
}
```

In order for `backdrop-filter` to work, the top layer needs to be at least partly transparent. If you use a background color, it should have an alpha transparency value of less than 1 (such as `hsla(300, 100%, 50%, .5)` or `#636a`). A transparent background color, as shown here, also works.

Firefox doesn't yet support `background-filter` by default. Adding a background color to `p` prevents the text in this example from becoming unreadable in Firefox and older versions of other browsers. How did I arrive at `hsla(22.4, 44.9%, 63.7%, .9)`? First, I used the eyedropper tool of an image editor to select a pixel color (`hsl( 202.4, 43%, 64.9%)`). Next, I subtracted 180 degrees to match the `hue-rotation` value (202.4 - 180). Lastly, I adjusted its transparency. This gives us a fallback effect that's similar to our backdrop filter, as shown in the image below.

Don’t forget to set a fallback when using `backdrop-filter`, particularly if you’re layering text on top of an image.

Wrapping the `backdrop-filter` rule set in an `@supports` block lets us undo the background color *only* if the browser supports the `backdrop-filter` property.

More on `@supports`

Learn more about `@supports` and how it works in Chapter 10, “Applying CSS Conditionally”.

How Filter Effects Affect Layout

As with `transform` and `opacity`, `filter` turns the element to which it’s applied into a containing block when its value is something other than `none`. It also creates a new, local stacking context. The image below shows the same markup before and after a `filter: hue-rotate(45deg);` declaration is added to the `<div>` element.

Using `filter` still creates a containing block and stacking context, even if the value of the `filter` property doesn't create a change in the element's appearance. In other words, `div {filter: blur(0px);}` would have the same impact on a layout as `filter: hue-rotate(45deg)`, but wouldn't change the color of the elements.

The `backdrop-filter` property works similarly. Adding `backdrop-filter` to an element contains its `absolute` and `fixed` positioned descendants. But it doesn't affect the stacking context of the element or elements that comprise the backdrop.

So far, we've covered blend modes and filter effects. We'll close out the chapter by looking at two more effects: clipping and masking.

Clipping and Masking

Clipping and masking are ways of concealing or revealing portions of a layer or document. Both are defined by the CSS Masking Module Level 1 specification.

Although they're defined by the same specification, clipping and masking work slightly differently.

- **Clipping** is a bit like using a cookie cutter. It uses a closed vector path, shape, or polygon to trim areas of a layer. Parts of the layer that lie outside the vector path are hidden. Parts that are inside the path are visible.
- **Masking** works more like peeled away masking tape, where painted or filled areas of the masking image expose the layers underneath.

In practical terms, the big difference is that clipping uses the `clip-path` property, and masking uses the `mask` shorthand or the `mask-*` longhand properties. Masking is more complicated to use, but it's also a little more flexible.

In some cases, you may want to wrap your clipping and masking rule sets in an `@supports` block.

The `clip-path` Property

The `clip-path` property accepts a basic shape or the URL of an SVG clip path. In the "Shapes" section of Chapter 5, "Layouts", we saw that there are five basic shape functions defined by the CSS Shapes specification:

- `inset()` (used to create rectangles)
- `circle()`
- `ellipse()`
- `polygon()`
- `path()` (which must be an SVG path data string)

Let's look at an example using the `ellipse()` shape function. Our markup is simple—just an `<img>` element:

```
<img src="milk-and-strawberry-in-a-spoon.jpg">
```

And here's our CSS:

```
img {
    display: block;
    height: 95vh;
    margin: 2rem auto;
    width: auto;

    /* Creates an oval shape that effectively crops the image */
    clip-path: ellipse( 50% 50% at 50% 50% );
}
```

Areas of the photograph that fall outside the clipping shape aren't painted to the screen, as seen in the image below.

Using `clip-path` with Polygons

The `clip-path` property also accepts a polygon shape as a value. Polygons are closed shapes made from straight lines. The `polygon()` function accepts a comma-separated list of coordinates for each point that makes up the polygon. Let's change our CSS to use a hexagonal polygon:

```
img {
    display: block;
    height: 95vh;
    margin: 2rem auto;
    width: auto;

    /* Creates a hexagon clipping area */
    clip-path: polygon(25% 0%, 75% 0%, 100% 50%, 75% 100%, 25% 100%, 0% 50%);
}
```

The following image shows the result of using a hexagonal polygon as a clip path.

Polygons can be tricky to create. Bennett Feely's Clippy is perhaps the best web-based tool for creating polygon clip paths. Firefox's developer tools also include a rudimentary polygon shape editor.

Creating More Complex Clipping Regions with `path()`

For more complex shapes made of curves and arcs alone or in combination, we need to use the `path()` function. `path()`, unlike `polygon()`, accepts an SVG data path (the value of the `d` attribute of a path element) as its argument:

```
img {
    display: block;
    height: 95vh;
    margin: 2rem auto;
    width: auto;

    /* Creates a blob-shaped clipping area */
    clip-path: path('m 104.3412,-94.552373 a 235.57481,242.55224 0 0 0 -235.57475,242.551213
235.57481,242.55224 0 0 0 235.57475,242.55309 235.57481,242.55224 0 0 0 35.90179,-2.83462
116.28209,116.28209 0 0 0 -0.3821,9.42037 A 116.28209,116.28209 0 0 0 256.14311,513.4198
116.28209,116.28209 0 0 0 363.17355,442.59305 288.39157,288.39157 0 0 0 645.41323,671.74187
288.39157,288.39157 0 0 0 933.80515,383.34996 288.39157,288.39157 0 0 0 645.41323,94.957809
288.39157,288.39157 0 0 0 587.24283,100.88816 116.28209,116.28209 0 0 0 587.69599,90.618827
116.28209,116.28209 0 0 0 471.41379,-25.663306 116.28209,116.28209 0 0 0 355.88656,77.404732
230.96342,158.23655 0 0 0 328.66979,73.946711 235.57481,242.55224 0 0 0 104.34119,-94.552373
Z');
}
```

The result is pictured below.

The easiest way to create path data is by creating an SVG image that contains a `path` element and grabbing value of the `path` element's `d` attribute. Using the `path()` function lets us create more complex shapes that include curves and arcs, in addition to straight lines.

Notice here that the points of the path exceed the bounds of the element being clipped. Parts of the image we've used to clip the photograph extend outside of the `<img>` element's bounds. Because `path()` uses coordinates and pixels, there isn't a way to scale it proportionally to fit an element's dimensions as can be done with other basic shapes.

This also means that `path()` isn't responsive. It won't change with the size of the viewport or the container. Basic shapes, however, *can* be responsive if you use percentages for the coordinates of each vertex.

Clipping paths don't affect the geometry of an element; they only affect its rendering. DOM functions such as `getClientRects()` return the dimensions of the entire element, not the clipped region.

Clipping paths *do*, however, affect the interactive area of an element. If, for example, you apply a `clip-path` to an `<a>` element, only the area within the bounds of the clip path will receive pointer and mouse events.

Using `clip-path` with URLs

Most current browser versions also support using SVG URLs as values. However Chromium-based browsers and Safari require the SVG to be inline. To date, Firefox is the only browser that also supports external SVG images. Of course, this may have changed between the writing of this paragraph and your reading of it.

However, we can't use just *any* SVG image. The image needs to contain a `clipPath` element, which should have one or more shapes, paths, or lines as children. An SVG `clipPath` must also have an `id` attribute, which is how we'll reference it. Here's an example of using an SVG `clipPath` element:

```
<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 400 400">
    <clipPath id="star">
    <path
        transform="translate(200, 0) scale(1.8)"
        style="fill: #000;"
        d="M 328.05692,385.18211 203.81502,322.90671 82.079398,389.94936 102.91396,252.54413
1.5342416,157.48399 138.65261,134.83828 197.73212,9.0452444 261.64137,132.45466 l
137.89287,17.31576 -97.62028,98.91692 z" />
    </clipPath>
</svg>
```

To use this image as a clip path, we'll need to use the `url()` function:

```
img {
    display: block;
    height: 95vh;
    margin: 2rem auto;
    width: auto;

    /* Creates a star-shaped clipping area */
    clip-path: url(#star);
}
```

You can see the result in the image below.

External SVG images work similarly. We just need to add the external URL of the image, such as `url(https://example.com/star.svg#svg).`

Origin Issues

When used with `clip-path`, external SVG are subject to cross-origin restrictions. You'll need to serve the image from the same origin, or set the appropriate `Access-Control-*` headers. See Chapter 6, "Working with Text", for more on dealing with cross-origin restrictions.

Using an inline SVG image will affect the layout of other items on your page, even though it's rendered invisibly. Luckily, there's an easy workaround: set the dimensions and position of the SVG element:

```
svg.clipper {
    position: absolute;
    top: 0px;
    left: 0px;
    width: 0px;
    height: 0px;
}
```

Since the contents of a clip path are only rendered when the clip path gets applied, it's still available to the document. However, the CSS above prevents the root `<svg>` element from taking up space.

Using `display: none` won't work. It prevents a box from being generated altogether. Similarly, `visibility: hidden` not only hides the root SVG element and its children, but also the layers you want to clip. Positioning the clipping path offscreen, or setting its height and width to zero avoids those issues.

Clipping defines a contiguous region of a layer that will be painted to the screen. Masked areas, on the other hand, don't need to be contiguous. Unlike `clip-path`, they can be also be resized and positioned.

Masking

Masking is likely familiar to you if you've ever worked with graphics editors such as Photoshop, Sketch, or Glimpse. It's an effect created when the painted areas of one layer reveal portions of another layer, as illustrated below.

Masks can be defined using the `mask` shorthand property, or the `mask-*` longhand properties. The behavior of masking properties mimics those of `background` properties. Masking, however, also affects foreground layers, and creates a new stacking context for those layers.

At the time of writing, Firefox has the most complete support for CSS masking. Firefox supports all masking properties, without a prefix. Chromium- and WebKit-based browsers support prefixed versions of all masking properties, with the exception of `mask-mode`. Firefox also includes support for the `-webkit-` prefixed subset of properties. The table below details `mask` property support across the three browser engine families.

Mask property	Firefox	Chromium	WebKit
`-webkit-mask`	Yes	Yes	Yes
`-webkit-mask-clip`	Yes	Yes	Yes
`-webkit-mask-composite`	Yes	No	No
`-webkit-mask-image`	Yes	Yes	Yes
`-webkit-mask-mode`	No	No	No
`-webkit-mask-origin`	Yes	Yes	Yes
`-webkit-mask-position`	Yes	No	No
`-webkit-mask-repeat`	Yes	Yes	Yes
`-webkit-mask-size`	Yes	Yes	Yes
`mask`	Yes	No	No
`mask-clip`	Yes	No	No
`mask-composite`	Yes	No	No
`mask-image`	Yes	No	No
`mask-mode`	Yes	No	No
`mask-origin`	No	No	No
`mask-position`	Yes	No	No
`mask-repeat`	Yes	No	No
`mask-size`	Yes	No	No

Tracking Browser Support

To keep track browser of support for each property, refer to the property's page on MDN Web Docs or caniuse.com (which uses MDN's browser compatibility data for some properties).

The CSS Masking Module Level 1 specification also defines a set of `mask-border-*` properties. As of this writing, however, support for these properties is still in the experimental phase. Safari, Chrome, and Edge use non-standard `-webkit-mask-box-*` properties. Firefox lacks any support. Since support and standardization are still in the works, we won't cover them here.

Creating a Mask with `mask-image`

To create a mask, use `mask-image`. Its value should be either a CSS image created with a gradient function, or the URL of an SVG `<mask>` element.

In SVG, the `<mask>` element is a container for other shapes that form the contours of the mask. It's similar to `clipPath`. The following code shows a simple example of a star-shaped mask:

```
<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 400 400">
  <mask id="star">
    <path style="fill: #000;" d="M 328.05692,385.18211 203.81502,322.90671
82.079398,389.94936 102.91396,252.54413 1.5342416,157.48399 138.65261,134.83828
197.73212,9.0452444 261.64137,132.45466 l 137.89287,17.31576 -97.62028,98.91692 z" />
  </mask>
</svg>
```

We can then reference this mask like so:

```
img {
    height: 95vh;
    margin: auto;
    display: block;
    width: auto;

    mask-image: url('masks.svg#star');
}
```

At least, that's how it's *supposed* to work, according to the specification. In practice, referring to an SVG `<mask>` element only works in Firefox. You can, however, use an SVG or PNG image as a masking image. Gradients created using the `linear-gradient()` function also work, but radial and conic gradients don't (as of this writing).

For our first masking example, we'll use the `star.png` shown below. It has a filled, star-shaped area where *alpha=1*, and it's surrounded by transparent pixels where *alpha=0*.

Here's our CSS:

```
img {
    height: 95vh;
    margin: auto;
    display: block;
    width: auto;

   -webkit-mask-image: url('star.png'); /* Chromium and WebKit browsers */
           mask-image: url('star.png');
}
```

As you can see in the image below, the *painted* or *filled* areas of `star.png` reveal the photograph below it. Transparent areas of `star.png` hide the layer (or layers) below it.

In this example, our mask image includes some areas that are 100% opaque and some that are 100% transparent. However, the rendering of masked layers may be affected by the alpha transparency or color of the masking layer.

Managing Mask Processing with `mask-mode`

Take a look at the image below, where a linear gradient transitions from an opaque to a transparent color. Notice how the image fades into the background, particularly toward the bottom.

This effect is created by using a linear gradient as the masking image:

```
img {
    mask-image: linear-gradient( to bottom, #000, #fff0 90% );
}
```

Our gradient transitions from opaque black, where *alpha = 1*, to white with an alpha transparency of 0. Where the mask image is 100% opaque, the toucan photograph is also fully opaque. As the gradient transitions from opaque to transparent, the image fades in opacity, approaching 0% at the bottom of the image.

This is an **alpha mask**. With alpha masks, the browser uses the alpha channel—and only the alpha channel—of the masking image to determine the pixel colors of the final painted layer. Portions of a layer that sit beneath the transparent portions of a mask image are invisible.

CSS (and SVG) also supports *luminance* masks. **Luminance masks** use the values of each color channel (red, green, and blue), in addition to the alpha channel, to calculate the color of the pixels painted to the screen. With luminance masks, areas of a layer that are overlapped by black or transparent colors will be invisible. Other areas of the layer will be more or less opaque based on the luminance of the colors in the masking layer.

We can change the behavior of a masking layer using the `mask-mode` property. It accepts one of three values: `alpha`, `luminance`, or `match-source`. Its initial value is `match-source`.

When the mask image is an SVG `<mask>` element and the value of `mask-mode` is `match-source`, the browser uses the computed value of the `<mask>` element's `mask-type` property. However, when the mask is an image, such as a gradient or PNG, the mask is processed as an alpha mask.

Let's look at an example using the linear gradient and photograph shown below, where a striped gradient is used to illustrate the difference between alpha and luminance masking.

Firstly, let's set up our linear gradient mask image. Although the CSS below doesn't include prefixed properties, don't forget to include the `-webkit-` prefix (such as `-webkit-mask-image`) when using this in the real world:

```
body {
    background: #000;
}
img {
    height: 95vh;
    margin: auto;
    display: block;
    width: auto;

    mask-image: linear-gradient(#f006 0 33.33%, #ff0a 33.33% 66.66%, #080 66.66% 100%);
    mask-mode: alpha;
}
```

Our gradient consists of a transparent red stripe (`#f006`), followed by a band of transparent yellow (`#ff00a`), and a bright, opaque green. We've also set a background color of black for our document. The image below shows the result of using a linear gradient with transparent colors as a mask image.

In this example, `mask-mode` is `alpha`, so each band of the gradient mask only changes the opacity of the photograph. If we change the value of `mask-mode` to `luminance`, however, the bright yellow band of the gradient is much brighter than the red band at top, and the green band is slightly darker than the yellow band. In fact, the red band is almost as black and opaque as the document's background color.

Luminance refers to the perceived lightness of a color. Yellow has a higher luminance than green. This particular green, at full opacity, has a higher luminance than our semi-transparent red. As a result, the yellow band is the brightest of the three.

Calculating Luminance

Although there are formulas for calculating the relative and absolute luminance of RGB(A) colors, it’s a lot easier to use the relative luminance and contrast ratio from Planetcalc.com.

Opaque white, of course, has the highest luminance value of all. Changing opaque blue to white creates that bright, full-color band across the bottom of the photo, as shown in the image below. Opaque white results in opaque pixels when the value of `mask-mode` is `luminance`.

Making Mask Images Repeat (or Not) with `mask-repeat`

The behavior of `mask-image` is a bit like that of `background-image`. For example, mask images repeat horizontally and vertically to fill the dimensions of the element to which they're applied.

We can prevent a mask image from repeating with the `mask-repeat` property. Let's update our `mask-image` CSS from earlier:

```
img {
    height: 95vh;
    margin: auto;
    display: block;
    width: auto;

    -webkit-mask-image: url('star.png');
            mask-image: url('star.png');

    -webkit-mask-repeat: no-repeat;
            mask-repeat: no-repeat;
}
```

This gives us the result shown below.

As you may have figured out, `mask-repeat` behaves a bit like `background-repeat`. In fact, it accepts the same values (up to two):

- `repeat-x`: repeats the mask image horizontally
- `repeat-y`: repeats the mask image vertically
- `repeat`: repeats the mask image in both dimensions
- `space`: repeats the mask as often as it can be repeated without being clipped, then spaces them out
- `round`: repeats the whole mask image, scaling the image up or down if necessary
- `no-repeat`: prevents the mask image from repeating

When `repeat-x` and `repeat-y` are used as `mask-image` values, it's the same as using `mask-image: repeat no-repeat` and `mask-image: no-repeat repeat`, respectively.

Resizing Mask Images with `mask-size`

Mask images, similarly to background images, can be resized using the `mask-size` property. Let's change our CSS a little bit. In the code below, we've resized the mask image and changed how it repeats:

```
img {
    height: 95vh;
    margin: auto;
    display: block;
    width: auto;
    -webkit-mask-image: url('star.png');
            mask-image: url('star.png');

    -webkit-mask-repeat: round;
            mask-repeat: round;

    -webkit-mask-size: 5vw auto;
            mask-size: 5vw auto;
}
```

Setting the `mask-size` property means that it can repeat more often across each dimension, as shown below.

As with `background-size`, permitted values for `mask-size` include length and percentages, as well as the `cover` and `contain` keywords.

It's also possible to change the `mask-position` and `mask-origin` of a masking image. These properties share values and behavior with their `background-position` and `background-origin` counterparts.

Using Multiple Mask Images

As mentioned earlier in this section, `mask-image` supports multiple masks, like its `background-image` counterpart. We'll switch from PNG images to SVG images for the examples in this section. We'll use `circle.svg` and `square.svg`, shown below.

The gray areas of the images above are just to show the position of the circle and square within the bounds of the SVG document. Those areas are transparent.

Let’s change our CSS to use multiple mask images:

```
img {
    height: 95vh;
    margin: auto;
    display: block;
    width: auto;

    -webkit-mask-repeat: no-repeat;
            mask-repeat: no-repeat;

    -webkit-mask-image: url('circle.svg'), url('square.svg');
            mask-image: url('circle.svg'), url('square.svg');
}
```

Multiple masking images follow the same ordering as `background-image`. The first image in the list becomes the topmost layer mask.

Here, the circle and square images overlap to form a single shape, as shown above. We can shape how masking layers are visually combined using the `mask-composite` property.

Managing Mask Layer Compositing with `mask-composite`

Compositing is the process of combining separate image layers or sources into a single visual layer. With mask images, the initial or default behavior is to add the layers together: `mask-composite: add`. The `mask-composite` property accepts one of four values: `add`, `subtract`, `intersect`, or `exclude`. The table below shows the effect of each property when the order of `mask-image` is `url('circle.svg'), url('square.svg')`.

Property	What it does	Example

Property	What it does	Example
`add`	Source, or top layer is placed *over* the destination or bottom layer. This is the initial value.	

Property	What it does	Example
`subtract`	Destination or bottom layer is subtracted from the top or source layer.	

Property	What it does	Example
`intersect`	Only the areas of the top layer that overlap the destination or bottom layer are painted.	

Property	What it does	Example
`exclude`	The non-overlapping areas of the source and destination layers are painted.	

The order of the layers in `mask-image` matters in some cases. Changing the order of our mask images to `mask-image: url('square.svg'), url('circle.svg');`, for example, changes the effect of `mask-composite: subtract`. The image below shows this difference. Since `square.svg` is now the top layer, `circle.svg` gets subtracted from it.

Using the `mask` Shorthand Property

You may find it easier to use the `mask` shorthand property. Let’s rewrite our `mask-composite` example using the `mask` shorthand:

```
img {
    mask: no-repeat url('circle.svg'), no-repeat url('square.svg');
}
```

Notice that we’ve repeated the `mask-repeat` value for each image in our `mask` list. If we wanted to get fancy and change the size and position of `square.svg`, we could use the following:

```
img {
    mask: no-repeat url('circle.svg'), no-repeat url('square.svg') 0 0 / 200px 200px;
}
```

The CSS above moves our square to the top-left corner of the container, and resizes it to `200px` by `200px`, as pictured below.

Longhand properties are more verbose, but clearer. Shorthand properties save bytes, but potentially at the expense of readability.

Conclusion

Now you’re all up to speed on visual effects! After reading this chapter, you should now know:

- how to use blend modes and predict the value of a blended pixel
- how to make layers appear in grayscale or sepia tone using filter effects
- how to use `clip-path` and masking, and when you might choose one over the other

In the next chapter, we’ll look at applying CSS conditionally.

Chapter 10: Applying CSS Conditionally

Conditional CSS refers to CSS rules that are applied when a condition is met. A condition may be a CSS property and value combination, as with the `@supports` rule. A condition may test for a browser window condition such as width, as with the `@media` rule. Or a condition may be a device feature such as hover capability or pointer input, as with some newer features of `@media`. We'll discuss all of the above in this chapter.

Both `@media` and `@supports` are described by the CSS Conditional Rules Module Level 3 specification. The `@media` rule—which you probably know as *media queries*—is fully defined by the Media Queries specification.

Media Queries and `@media`

The `@media` rule is actually a long-standing feature of CSS. Its syntax was originally defined by the CSS 2 specification back in 1998. Building on the media types defined by HTML 4, `@media` enabled developers to serve different styles to different media types—such as `print` or `screen`.

The Media Queries Level 3 specification extended the `@media` rule to add support for media *features* in addition to media types. Media features include window or viewport width, screen orientation, and resolution. Media Queries Level 4 added **interaction media features**, a way to apply different styles for pointer device quality—that is, the fine-grained control of a mouse or stylus versus the coarseness of a finger. Media Queries Level 5 adds features for `light-level` and `scripting`, along with user preference media features such as `prefers-reduced-motion` and `prefers-reduced-transparency`.

Alas, most of the media types defined by HTML 4 are now obsolete. Only `all`, `screen`, `print`, and `speech` are currently defined by any specification. Of those, only `all`, `screen` and `print` have widespread browser support.

We'll briefly discuss them in the examples that follow. As for media features, we'll focus on what's available in browsers today.

Media Query Syntax: The Basics

Media query syntax seems simple, but sometimes it's a bit counterintuitive. In its simplest form, a media query consists of a media type, used alone or in combination with a media condition—such as `width` or `orientation`. A simple, type-based media query for screens looks like this:

```
@media screen {
    /* Styles go here */
}
```

CSS style rules are nested within this `@media` rule set. They'll only apply when the document is displayed on a screen, as opposed to being printed:

```
@media screen {
    body {
        font-size: 20px;
    }
}
```

In the example above, the text size for this document will be `20px` when it's viewed on a desktop, laptop, tablet, mobile phone, or television.

If No Media Type Is Specified

When no media type is specified, it's the same as using `all`.

We can apply CSS to one or more media types by separating each query with a comma. If the browser or device meets *any* condition in the list, the styles will be applied. For example, we could limit styles to screen and print media using the following:

```
@media screen, print {
    body {
        font-size: 16px;
    }
}
```

The real power of media queries, however, comes when you add a media feature. **Media features** interrogate the capabilities of the device or conditions of the viewport.

A media feature consists of a property and a value, separated by a colon. The query *must* also be wrapped in parentheses. Here's an example:

```
/* Note that this is the equivalent of @media all (width:
480px) */
@media ( width: 480px ) {
    nav li {
        display: inline-block;
    }
}
```

Now `nav li` will have a `display` value of `inline-block` only when the width of the viewport is equal to 480 pixels. Let's use the `and` keyword to make a more specific media query:

```
@media screen and ( width: 480px ) {
    nav li {
        display: inline-block;
    }
}
```

These styles will be used only when the output device is a screen and its width is `480px`. Notice here that the media type *is not* enclosed by parentheses, but the media feature—`(width: 480px)`—is.

But the query above has a small problem. If the viewport is wider than `480px` or narrower than `480px`—and not *exactly* `480px`—these styles won't be applied. What we need instead is a *range*.

Range Media Features and `min-` and `max-` Prefixes

A more flexible media query might test for a minimum or maximum viewport width. We can apply styles when the viewport is *at least* this wide, and *no more than* that wide. Luckily for us, the Media Queries Level 3 specification defines the `min-` and `max-` prefixes for this purpose. These prefixes establish the lower or upper boundaries of a feature range.

Let’s update our previous code:

```
@media ( max-width: 480px ) {
    nav li {
        display: block;
    }
}
```

In this example, `nav li` will have a `display` property value of `block` from a viewport width of `0`, *up to and including* a maximum viewport width of `480px`.

We can also define a media query range using `min-` and `max-`, along with the `and` keyword. For example, if we wanted to switch from `display: block` to `display: flex` between `481px` and `1600px`, we might do the following:

```
@media ( min-width: 481px ) and ( max-width: 1600px ) {
    nav ul {
        display: flex;
    }
}
```

If both conditions are true—that is, the viewport width is at least `480px`, but not greater than `1600px`—our styles will apply.

Not all media feature properties support ranges with `min-` and `max-`. The table below lists those that do, along with the type of value permitted for each.

Property	Description	Value type
`aspect-ratio`	The ratio of viewport `width` to `height`	ratio (such as `1024/768` or `16:9`)
`color`	Number of bits per color component of the device; `0` when the device is not a color device	integer
`color-index`	Minimum number of colors available on the device	integer
`height`	Height of the viewport or page box	length

Property	Description	Value type
`monochrome`	Number of bits per pixel in a monochrome frame buffer	integer
`resolution` (see note below)	Describes the pixel density of a device	resolution (`dpi`, `dpcm`, and `dppx`) units
`width`	Width of the viewport or page box	length

Safari and `resolution`

Safari doesn't yet support the `resolution` media feature. Instead, it still uses the non-standard, vendor-prefixed `-webkit-device-pixel-ratio` feature. For `min`/`max` ranges, use `-webkit-min-device-pixel-ratio` and `-webkit-max-device-pixel-ratio`.

Firefox versions 63 and above support comparison operators such as `>` and `<=` in addition to the `min` and `max` syntax for ranges. Instead of `@media (min-width: 480px) and (max-width: 1600px)`, we could write this query as follows:

```
@media ( width >= 480px ) and ( width <= 1600px ) {
    nav li {
        display: block;
    }
}
```

That's a little clearer than `@media ( min-width: 480px ) and ( max-width: 1600px )`. Unfortunately, this syntax isn't yet supported by most browsers. Stick with `min-` and `max-` for now.

Discrete Media Features

There's a second type of media feature: the discrete type. **Discrete media features** are properties that accept one of a set—or a predefined list—of values. In some cases, the set of values is a Boolean—either `true` or `false`. Here's an example using the `orientation` property. The example adjusts the proportional height of a logo when in `portrait` mode:

```
@media screen and ( orientation: portrait ) {
    #logo {
        height: 10vh;
        width: auto;
    }
}
```

The `orientation` feature is an example of a discrete media feature. It has two supported values, `portrait` and `landscape`. Minimum and maximum values don't make much sense for these properties. The table below lists discrete media features that are currently available in major browsers.

Property	Description	Acceptable values
`any-hover`	Ability of *any* connected input mechanism to have a hover state as determined by the user agent	`none`, `hover`
`hover`	Ability of the *primary* input mechanism to have a hover state as determined by the user agent	`none`
`any-pointer`	Presence and accuracy of *any* pointing device available to the user	`none`, `coarse`, `coarse`
`pointer`	Presence and accuracy of the *primary* pointing device as determined by the user agent	`none`, `coarse`, `coarse`
`grid`	Whether the device is grid (such as a teletype terminal or phone with a single fixed font) or bitmap. Don't confuse this with CSS Grid layout	Boolean (see note below)
`orientation`	Describes behavior for whatever is larger out of width or height. When the width is greater than height, the orientation is `landscape`. When the inverse is true, the orientation is `portrait`	`portrait`, `landscape`
`prefers-reduced-motion`	Defines styles when the user has disabled animations for their operating system	`no-preference`, `reduce`

Property	Description	Acceptable values
`prefers-color-scheme`	Set styles when the user has indicated that they prefer a color scheme for their operating system	`light`, `dark`

Boolean Feature Queries

Boolean feature queries have an unusual syntax. You can use either `0` or `1` as a value, or just the feature itself. In the case of `grid`, this would be `@media (grid)` or `@media (grid: 1)`.

Other discrete media features include `overflow-block` and `overflow-inline`, which describe the behavior of the device when content overflows in the block or inline direction (think electronic billboards or slide shows). Eventually, we may also see support for a `scripting` feature which tests for JavaScript support.

One discrete media feature we can use now is `hover` (along with `any-hover`). The `hover` media feature query allows us to set different styles based on whether or not the primary input mechanism supports a `:hover` state. The `any-hover` feature works similarly, but applies to any input mechanism, not just the primary one. It's a discrete feature type, and has just two valid values:

- `none`: the device has no hover state, or has one that's inconvenient (for example, it's available after a long press)
- `hover`: the device has a hover state

Consider the case of radio buttons and checkbox form controls on touchscreens. Touchscreen devices typically have an on-demand hover state, but may lack one completely. Adult-sized fingers are also fatter than the pointers of most mouse or track pad inputs. For those devices, we might want to add more padding around the label, making it easier to tap:

```
@media screen and ( hover: on-demand ) {
    input[type=checkbox] + label {
        padding: .5em;
    }
}
```

Another media feature that's well supported by browsers is the `pointer` media feature (and `any-pointer`). With `pointer`, we can query the presence and accuracy of a pointing device for the *primary* input mechanism. The `any-pointer` property, of course, tests the presence and accuracy of *any pointer* available as an input mechanism. Both media features accept one of the following values:

- `none`: the device's primary input mechanism is not a pointing device
- `coarse`: the primary input mechanism is a pointing device with limited accuracy
- `fine`: the device's primary input mechanism includes an accurate pointing device

Devices with pointing inputs include stylus-based screens or pads, touchscreens, mice, and track pads. Of those, touchscreens are generally less accurate. Stylus inputs, on the other hand, are very accurate—but, like touchscreens, they lack a hover state. With that in mind, we might update our `hover` query from earlier so that we only add padding when the `pointer` is `coarse`:

```
@media screen and ( hover: none ) and ( pointer: coarse ) {
    input[type=checkbox] + label {
        padding: .5em;
    }
}
```

Multiple Device Inputs

Don't assume that a *primary* input is the *only* input for a device, or even that it's the main input for the user. In fact, interaction media features don't account for keyboards at all. In his article "Interaction Media Features and Their Potential (for Incorrect Assumptions)", Patrick H. Lauke explains the limits of the `pointer`/`any-pointer` and `hover`/`any-hover` media features.

Most operating systems include a set of accessibility and user preference settings that control features like the animation and transparency of windows, or system-wide theming preferences. Level 5 of the Media Queries specification defines several features for querying user-preference settings: `prefers-reduced-motion`, `prefers-color-scheme`, `prefers-contrast`, `prefers-reduced-transparency`, `prefers-reduced-data` and `forced-colors`. Of these, only `prefers-reduced-motion` and `prefers-color-scheme` have widespread support across browsers and operating systems.

Using `prefers-reduced-motion` to Improve the Experience of People with Vestibular and Seizure Disorders

As mentioned in Chapter 7, "Transitions and Animations", large-scale animations can create sensations of dizziness and nausea for people with vestibular disorders. Flickering animations can cause seizures for people with photosensitive epilepsy.

Seizures and dizziness don't make for a very good user experience. At the same time, animation can improve usability for users who aren't affected by vestibular disorders. As a way to balance improved usability for some while preventing debilitating conditions in others, WebKit proposed a `prefers-reduced-motion` media feature. It has two possible values: `no-preference` and `reduce`.

Rationale

"Responsive Design for Motion", a blog post from the WebKit team, explains the team's rationale for proposing the `prefers-reduced-motion` query, as well as how to use it.

With `prefers-reduced-motion`, we can provide an alternative animation or disable it altogether, as shown in the following example:

```
/* Starting state */
.wiggle {
    animation: wiggling 3s ease-in infinite forwards
alternate;
}
@media screen and ( prefers-reduced-motion: reduce ) {
    .wiggle {
        animation-play-state: paused;
    }
}
```

If the user's preference is to reduce motion, the `.wiggle` animation will be disabled.

When used without a value, `prefers-reduced-motion` is true. In other words, removing `reduce` from the above media query gives it an equivalent meaning:

```
@media screen and ( prefers-reduced-motion ) {
    .wiggle {
        animation-play-state: paused;
    }
}
```

Even when the user has chosen to reduce motion, your animations won't be disabled unless you add CSS to accommodate that preference. You may instead wish to enable transitions and animations only when the user *hasn't* indicated a preference:

```
/* @media screen and not ( prefers-reduced-motion ) also
works */
@media screen and ( prefers-reduced-motion: no-preference ) {
    .wiggle {
        animation: wiggling 3s ease-in infinite forwards
alternate running;
    }
}
```

Chrome versions 73 and below, Firefox versions 62 and below, Edge versions 18 and below, and Safari versions 10.1 and below don't support `prefers-reduced-motion`. Consider adding a user interface element that lets site visitors disable animations in those browsers. Don't forget to follow WCAG guidelines when creating animations and transitions.

Respecting Users Color Preferences with `prefers-color-scheme`

Some operating systems offer the ability to select a dark theme for the interface. We can use the `prefers-color-scheme` feature to add support for this preference in web pages and applications. This feature has two possible values: `light` and `dark`.

```
@media ( prefers-color-scheme: dark ) {
    /* Styles here */
}
```

The `prefers-color-scheme` feature works well with custom properties (see Chapter 4). For example, you might use a different color palette for each color scheme, and use custom properties to define each color:

```
/* Styles when there's no preference */
:root {
    --background: #ccc;
    --foreground: #333;
    --button-bg:  #505;
    --button-fg:  #eee;
    --link:       #909;
    --visited:    #606;
}
/* Update colors for the background, foreground, and buttons
*/
@media screen and ( prefers-color-scheme: light ) {
   :root {
        --background: #fff;
        --foreground: #000;
        --button-bg:  #c0c;
        --button-fg:  #fff;
   }
}
@media screen and ( prefers-color-scheme: dark ) {
    :root {
        --background: #222;
        --foreground: #eee;
        --button-bg:  #808;
        --button-fg:  #fff;
        --link:       #f0f;
        --visited:    #e0f;
    }
}
```

When creating themes for use with `prefers-color-scheme`, don't forget to check whether your foreground and background colors have sufficient contrast. Firefox, Chrome and Edge have robust accessibility checking tools built into their developer tools. Deque provides axe, a free developer tools browser extension for Firefox, Chrome and Edge that includes checks for color contrast. Tools such as Lea Verou's Contrast Ratio also work well.

To develop and test for dark mode, you'll first need to enable it:

- Windows: go to **Settings** > **Personalization** > **Colors** > **Choose your color**
- macOS: go to **System Preferences** > **General** > **Appearance**
- Ubuntu: go to **Settings** > **Appearance** > **Window colors**

Chrome and Edge also allow users to set a preference at the browser level by enabling the Force Dark Mode for Web Contents setting. You can find this setting at `chrome://flags/#enable-force-dark` and `edge://flags/#enable-force-dark`, respectively. Keep in mind that even when Force Dark Mode is enabled, `matchMedia('(prefers-dark-mode)').matches` may return `false` for some operating systems.

In Firefox, you can simulate light (via the sun-shaped icon) and dark (via the crescent-moon–shaped icon) color scheme support in the web inspector panel.

A third value, `no-preference`, has been removed from the specification due to a lack of browser support. You may, however, come across articles or code samples that include it. Don't use it in new projects.

Nesting `@media` Rules

It's also possible to nest `@media` rules. Here's one example where it might be useful to nest media queries:

```
@media screen {
    @media ( min-width: 320px ) {
        img {
            display: block;
            width: 100%;
            height: auto;
        }
    }
    @media ( min-width: 640px ) {
        img {
            display: inline-block;
            max-width: 300px;
        }
    }
}
```

In this example, we’ve grouped all our `screen` styles together, with subgroups for particular viewport widths.

Working around Legacy Browser Support with `only`

As mentioned in the beginning of this chapter, `@media` has been around for a while. However, the syntax and grammar of `@media` has changed significantly from its original implementation. As the Media Queries Level 4 specification explains, the original error-handling behavior:

> would consume the characters of a media query up to the first non-alphanumeric character, and interpret that as a media type, ignoring the rest. For example, the media query `screen and (color)` would be truncated to just `screen`.

To avoid this, we can use the `only` keyword to hide media queries from browsers that support the older syntax. The `only` keyword must precede a media query, and affects the entire query:

```
@media only screen and ( min-resolution: 2dppx ) {
    /* Styles go here */
}
```

`only` the Lonely

The `only` keyword tells the browser that these styles should be applied only when the following condition is met. The good news is that the older error-handling behavior is mostly an edge case among browsers that are in use today. For most current browsers and current web users, using the `only` keyword is unnecessary. I've included it here for completeness.

Negating Media Queries

You can also negate a media query using the `not` keyword. The `not` keyword must come at the beginning of the query, before any media types or features. For example, to hide styles from `print` media, you might use the following:

```
@media not print {
    body {
        background: url( 'paisley.png' );
    }
}
```

If we wanted to specify low-resolution icons for lower-resolution devices instead, we might use this snippet:

```
@media not print and ( min-resolution: 1.5dppx ) {
    .external {
        background: url( 'arrow-lowres.png' );
    }
}
```

Notice here that `not` comes before and negates the *entire* media query. You can't insert `not` after an `and` clause. Arguments such as `@media not print and not (min-resolution: 2dppx)` or `@media screen and not (min-resolution: 2dppx)` violate the rules of media query grammar. However, you can use `not` at the beginning of each query in a media query list:

```
@media not ( hover: hover ), not ( pointer: coarse ) {
    /* Styles go here */
}
```

Styles within this grouping rule would be applied when the device doesn't have a hover state or when the pointing device has fine-grained accuracy.

Other Ways to Use Media Queries

Thus far, we've talked about `@media` blocks within stylesheets, but this isn't the only way to use media types and queries. We can also use them with either

`@import` or the `media` attribute. For example, to import a stylesheet `typography.css` when the document is viewed on screen or printed, we could use the following CSS:

```
@import url( typography.css ) screen, print;
```

But we can also add a media query to an `@import` rule. In the following example, we're serving the `hi-res-icons.css` stylesheet only when the device has a minimum pixel density of `2dppx`:

```
@import url( hi-res-icons.css ) ( min-resolution: 2dppx );
```

HTTP/1.1

For browsers and servers that still use HTTP/1.1, `@import` adds an additional HTTP request and blocks other assets from downloading. Use it with care!

Another way to use queries is with the `media` attribute, which can be used with the `<style>`, `<link>`, `<video>`, and `<source>` elements. In the following example, we'll only apply these linked styles if the device width is 480 pixels wide or less:

```
<link rel="stylesheet" href="styles.css" type="text/css"
media="screen and (max-width: 480px)">
```

Performance Considerations

In every browser tested, the stylesheet will be requested and downloaded, even when the media query doesn't apply. However, linked assets within that stylesheet (for example, background images defined with `url()`) won't be.

Precedence

If your linked stylesheets also contain media queries, these will take precedence over the value of the `media` attribute.

We can also use the `media` attribute with the `<source>` element to serve different files for different window widths and device resolutions. What follows is an example using the `<source>` element and `media` attribute with the `<picture>` element:

```
<picture>
    <source srcset="image-wide.jpg" media="( min-width:
1024px )">
    <source srcset="image-med.jpg" media="( min-width: 680px
)">
    <img src="image-narrow.jpg" alt="Adequate description of
the image contents.">
</picture>
```

Content-driven Media Queries

A current common practice when using media queries is to set `min-width` and `max-width` *breakpoints* based on popular device sizes. A **breakpoint** is the width or height that triggers a media query and its resulting layout changes. Raise your hand if you've ever written CSS that resembles this:

```
@media screen and ( max-width: 320px ) {
    ⋮
}
@media screen ( min-width: 320px ) and ( max-width: 480px ) {
    ⋮
}
@media screen ( min-width: 481px ) and ( max-width: 768px ) {
    ⋮
}
@media screen ( min-width: 769px ) {
    ⋮
}
```

These work for a large number of users. But device screen widths are more varied than this. Rather than focus on the most popular devices and screen sizes, try a content-centric approach.

Don't Use `device-width` with Media Queries

Avoid using `device-width` (including `min`/`max`) altogether for media queries. High DPI devices in particular may report a device width that doesn't match its actual pixel capability.

A content-centric approach to media queries sets breakpoints based on the point at which the layout starts to show its weaknesses. One strategy is to start small, which is also known as a **mobile-first** approach. As Bryan Reiger puts it, “the absence of support for `@media` queries is in fact the first media query”.

You can do a lot to create a flexible, responsive layout *before* adding media queries. Then, as you increase the viewport width or height, add styles that take advantage of the additional real estate. For example, how wide is the browser window when lines of text become too long to read comfortably? That can be the point at which your layout switches from a single-column layout (as illustrated in the first image below) to a two-column layout (shown in the second image).

There are two advantages to this approach. First, your site will still work on older mobile browsers that lack support for media queries. The second reason is just as important: this approach prepares your site for a wider range of screen widths and resolutions.

Using Media Queries with JavaScript

Media queries also have a JavaScript API, better known as `matchMedia()`. If you're not versed in JavaScript, don't worry. We'll keep the examples short so they're easier to understand. The API for media queries is actually defined by a different specification, the CSSOM View Module. It's not CSS, strictly speaking, but since it's closely related to `@media`, we'll cover it.

The `matchMedia()` method is a property of the `window` object. That means we can refer to it using `window.matchMedia()` or just `matchMedia()`. The former is clearer, since it indicates that this is a native JavaScript method, but the latter saves a few keystrokes. I'm a lazy typist, so I'll use `matchMedia()` in the examples that follow.

Use `matchMedia()` to test whether a particular media condition is met. The function accepts a single argument, which must be a valid media query.

Why use a media query with JavaScript rather than CSS? Perhaps you'd like to display a set of images in a grid on larger screens, but trigger a slide show on small screens. Maybe you want to swap the `src` value of a `<video>` element based on the screen size or resolution. These are cases for using `matchMedia()`.

Here's a simple example of `matchMedia` in action. This code checks whether the viewport width is greater than or equal to `45em`:

```
var isWideScreen = matchMedia( "(min-width: 45em)" );
console.log( isWideScreen.matches ); // Logs true or false to
console
```

Using `matchMedia()` creates a `MediaQueryList` object. Here, that object is stored in the `isWideScreen` variable. Every `MediaQueryList` object contains two properties:

- `media`, which returns the media query argument that was passed to `matchMedia()`
- `matches`, which returns `true` if the condition is met and `false` otherwise

Since we want to know whether it's true that the browser window is at least `45em` wide, we need to examine the `matches` property.

`MediaQueryList.matches` will return `false` when either:

- the condition isn't met at the time `matchMedia()` is invoked
- the syntax of the media query is invalid
- the browser doesn't support the feature query

Otherwise, its value will be `true`.

Here's another example of using `matchMedia`. We'll update the source of a `<video>` element based on the size of the current viewport and resolution:

```
if( matchMedia( "( max-width: 480px ) and ( max-resolution:
1dppx )" ) {
    document.querySelector('video').src = 'smallvideo.mp4';
}
```

If the condition doesn't match—or the browser doesn't support the `resolution` feature query—the value of `src` won't change.

Error Checking with `not all`

Typically, the value of the `media` property is the media query we've tested. But maybe you forgot to include the parentheses around your feature query (a syntax error). Or perhaps the query uses a `pointer` feature query, but the browser is yet to support it. In both of those cases, the browser will return a `not all` value. This is media query speak for "this doesn't apply to any media condition".

In cases where the media query is a list—that is, when it contains multiple conditions—the value of `matchMedia().media` will also contain multiple values. If part of that query list is invalid or unsupported, its value will be `not all`. Here's an example:

```
var mq = matchMedia( "( hover: none ), ( max-width: 25em )"
);
```

In browsers lacking support for the `hover: none` media feature query, the value of `mq.media` will be `not all, (max-width: 25em)`. In browsers that do support it, the value of `mq.media` will be `(hover: none), (max-width: 25em)`. Let's look at another example:

```
var mq = matchMedia( "min-resolution: 1.25dppx, ( max-width:
25em )" );
```

In this example, the value of `mq.media` will also be `not all, ( max-width: 25em )`. In this case, however, it's because our first feature query uses the wrong syntax. Remember that media feature queries need to be enclosed in parentheses. The argument should be `matchMedia( "( min-resolution: 1.25dppx ), ( max-width: 25em )" );` instead.

Listening for Media Changes

Media conditions aren't necessarily static. Conditions can change when the user resizes the browser or toggles between portrait and landscape mode. Luckily, there's a mechanism for monitoring and responding to changes in our document's environment: the `addEventListener()` method.

The `addEventListener()` method is a standard method of the Document Object Model. It accepts two arguments: the event type, and a callback function. The callback function gets invoked every time an event of the specified type occurs. Changes to the document's environment are always `change` events.

Let's add a class name when our document enters landscape orientation. The first step is to create a `MediaQueryList` object using `matchMedia` and a media query:

```
var isLandscape = matchMedia( "( orientation: landscape )" );
```

Step two is to define our callback function. The callback function receives an object as its only argument. In Chrome, Safari and Microsoft Edge, this will be a `MediaQueryListEvent` object. In Firefox (verified in versions 90 and below), it's a `MediaQueryList` object, which is a holdover from an earlier version of the specification. There isn't much difference between them, and the code below works with both object types:

```
const toggleClass = function ( mediaquery ) {
    if ( mediaquery.matches ) {
        document.body.classList.add( 'widescreen' );
    } else {
        document.body.classList.remove( 'widescreen' );
    }
}
```

Media query events aren't very smart. They're fired any time the value of `MediaQueryList.matches` changes, regardless of whether or not the condition is `true`. This means we need to examine the value of the `MediaQueryListEvent.matches` or `MediaQueryListEvent.media` property. In this case, if the value of `mediaquery.matches` is `true`, we'll add a class name to our `<body>` element. Otherwise, we'll remove it.

Finally, let's add this event listener to our `MediaQueryList` object with `addEventListener`:

```
isLandscape.addEventListener( 'change', toggleClass );
```

To remove a listener, use `removeEventListener` as shown:

```
isLandscape.removeEventListener( toggleClass );
```

Early versions of the CSSOM View specification defined `addListener` and `removeListener` methods. These methods were separate mechanisms, removed from the DOM event queue. This changed in the Level 4 specification. Both functions are now deprecated, but older browsers still support them.

One workaround for this is to test whether the browser supports `addEventListener` when used with a `MediaQueryList` object:

```
if( typeof isLandscape.addEventListener === 'function' ) {
    isLandscape.addEventListener( 'change' , toggleClass );
} else {
    isLandscape.addListener( toggleClass );
}
```

You can use a similar check for `removeEventListener` and `removeListener`.

Testing for Property Support with Feature Queries

Feature queries let us apply CSS rules when the browser supports a particular property and value combination. As with media queries, feature queries consist of two parts: the `@supports` CSS rule, and a DOM-based API for use with JavaScript.

Why might we use `@supports`? Here's a scenario: as originally specified, `display` allowed four possible values: `block`, `inline`, `list-item`, and `none`. Later specifications added `table-*` values, `flex`, and `grid`. With `@supports`, we can define CSS rules that will be applied only when the browser supports `display: grid`:

```
@supports ( display: grid ) {
    .gallery {
        display: grid;
        grid-template-columns: repeat( 4, auto );
    }
}
```

To define a condition, wrap the property and value you'd like to test in a set of parentheses as shown. Both portions are required. A condition such as `@supports (hyphens)` won't work. You *can*, however, use a CSS keyword such as `unset` or `initial` as part of the test—such as `@supports (hyphens: initial)`.

To combine conditions, use the `and` keyword. For example, if you wanted to apply styles when both the `text-decoration-color` and `text-decoration-style` are supported, you could use the following:

```
@supports ( text-decoration-color: #c09 ) and ( text-
decoration-style: double ) {
    .title {
        font-style: normal;
        text-decoration: underline double #f60;
    }
}
```

The `@supports` syntax also allows disjunctions using the `or` keyword. Disjunctions are especially useful for testing vendor-prefixed property support. Older versions of WebKit-based browsers require a vendor prefix for flexible box layout support. We can augment our `@supports` condition to take that into account:

```
@supports ( display: flex ) or ( display: -webkit-flex ) {
    nav ul {
        display: -webkit-flex;
        display: flex;
    }
}
```

Finally, we can also define a collection of styles if a condition isn't supported by using the `not` keyword:

```
@supports not ( display: grid ) {
    nav {
        display: flex;
    }
}
```

The `not` keyword can only be used to negate one condition at a time. In other words, `@supports not (text-decoration-color: #c09) and (text-decoration-style: double)` is not valid. But you *can* combine two tests into a single condition by using an outer set of parentheses: `@supports not ((text-decoration-color: #c09) and (text-decoration-style: double))`.

Very old browsers, such as Internet Explorer, lack support for both `@supports` and properties you might wish to query, such as `float: inline-start`. For those browsers, we can leverage CSS error handling and the cascade instead. CSS ignores rules that it can't parse, and the last-defined rule wins. Below is an example using the `float` property:

```
img {
    float: left;          /* Browsers that support the old
float values */
    float: inline-start; /* Browsers that support newer
logical values */
}
```

Using error handling and the cascade often works well enough that you can forgo using `@supports` altogether. You may, however, need to use `@supports` to isolate and override declarations supported by both older and newer browsers. Consider the following CSS:

```
nav ul {
    text-align: center;
    padding: 0;
}
nav li {
    display: inline-block;
    min-width: 20rem
}
nav li:not( :last-child ) {
    margin: 0 1.5rem 0 0;
}
@supports ( display: grid ) {
    nav ul {
        display: grid;
        grid-template-columns: repeat( auto-fit, 20rem );
        justify-content: center;
        gap: 1.5rem;
    }

    /* Undo all of the styles from above */
    nav li:not( :last-child ),
    nav li {
        display: initial;
        margin: 0;
        min-width: unset;
    }
}
```

In this case, we've used `@supports` to remove the margin for `nav li` only when the browser supports CSS Grid. Browsers that lack support for feature queries ignore the entire block.

Determining Selector Support with `selector()`

Originally designed to test support of properties and values, the CSS Conditional Rules Module Level 4 specification expands the syntax of `@supports` to include selectors using `selector()`. Here's an example:

```
@supports selector( :blank ) {
    input:not(:blank):invalid {
        background: pink;
    }
}
```

In browsers that support the `:blank` pseudo-class (and to date, no browser does), `<input>` elements that contain invalid data but are not blank will have a pink background.

Remember that CSS ignores rules and selectors that it doesn't understand. In other words, you probably don't need to use `selector()`. If you do, make sure your site degrades gracefully.

`CSS.supports` DOM API

Feature queries also have an API: `CSS.supports()`. `CSS.supports()` always returns a Boolean (`true` or `false`) value depending on whether or not the browser supports that property and value combination.

`CSS.supports()` accepts a parentheses-wrapped CSS declaration as its argument. For example:

```
CSS.supports( '( text-decoration: underline wavy #e91e63 )'
);
```

If the browser supports this syntax for `text-decoration`, `CSS.supports` returns `true`. Otherwise, it returns `false`.

We can test multiple conditions using conjunctions (the `and` keyword) or disjunctions (the `or` keyword). `CSS.supports` also allows negation using the `not` keyword. For example, we can test whether a browser supports `display: -webkit-flex` or `display: flex` using the following:

```
CSS.supports( '( display: -webkit-flex ) or ( display: flex
)' );
```

Most browsers treat parentheses as optional when testing a *single* property and value combination (versions of Microsoft Edge 18 and under are an exception). When testing support for multiple conditions, each one *must* be wrapped in parentheses, as we've done here. Failing to do so means that `CSS.supports()` may return a false negative.

You can also use `CSS.supports` with `selector()` to test selector support, as shown below:

```
const canUseIs = CSS.supports( 'selector( :is() )' );
console.log( canUseIs ); // Logs true or false
```

Enclose the entire condition in quotes to prevent a JavaScript `ReferenceError` that `selector` is `undefined`.

Selectors that use functional notation—`:is()`, `:where()` and `:has()`—should include parentheses. When testing support for `:not()`, you'll also need to include an argument (it's optional for the other selectors):

```
CSS.supports( 'selector( :not() )'); // Returns false

CSS.supports('selector( :not( :last-child ) )'); // Returns
true
```

Although this feature isn't well-documented, it is widely supported in browsers.

Understanding the Cascade for `@supports` and `@media`

Using `@supports` or `@media` doesn't increase the specificity or importance of a rule. Normal cascade rules apply, meaning that styles defined after an `@supports` or `@media` block will override rules within the block. Consider the following CSS:

```
@supports ( text-decoration: underline wavy #c09 ) {
    .title {
        font-style: normal;
        text-decoration: underline wavy #c09;
    }
}
.title {
    font-style: italic;
}
```

All elements with a `title` class will be both italicized and underlined. The subsequent `font-style: italic;` line overrides the `font-style: normal;`. That's not what we want here. Instead, we need to flip the order of our rule sets, so that `font-style: normal` takes precedence over `font-style: italic`:

```
.title {
    font-style: italic;
}
@supports ( text-decoration: underline wavy #c09 ) {
    .title {
        font-style: normal;
        text-decoration: underline wavy #c09;
    }
}
```

Both `@supports` and `@media` work best when used to progressively enhance a site. Define your base styles—the styles that every one of your target browsers can handle. Then use `@supports` or `@media` to override and supplement those styles.

Conclusion

Both `@media` and `@supports` are powerful and flexible ways to progressively enhance your CSS and serve a range of devices. Now that you've reached the end of this chapter, you should know how to use:

- `@media` to create flexible layouts for a range of devices and inputs
- `window.matchMedia()` and the `addEventListener`/`removeEventListener` methods to call JavaScript based on a media query
- `@supports` and the `CSS.supports()` API to progressively enhance documents

In the next chapter, we’ll learn about two scroll-related CSS properties and features: Scroll Snap, and the `scroll-behavior` property.

Chapter 11: CSS and Scrolling

Smooth scrolling previously required the use of JavaScript to calculate the speed and timing of a scrolling operation. Developers of a certain age may remember using the `animate()` method of the jQuery JavaScript library, or the MooTools `Fx.Scroll()` class to make a page scroll to a given location.

Similarly, carousels and slide shows often required JavaScript libraries. Perhaps you've tried a slide show such as Flickity or FlexSlider that uses JavaScript to animate the position of a scrolling container by updating its `transform` value.

In this chapter, we'll look at two CSS features—the `scroll-behavior` property, and CSS Scroll Snap—that make it possible to create jump links that scroll smoothly, and carousels that require minimal amounts of JavaScript, if any.

Scrolling and Vestibular Disorders

Smooth scrolling may cause dizziness for people who have vestibular disorders. Use the `prefers-reduced-motion` media feature and `@media` to reduce or disable scrolling for visitors who've indicated a preference.

Dump the Jump: Smooth Internal Links with `scroll-behavior`

The `scroll-behavior` property manages the behavior of scrolling when caused by navigation, or the invocation of a CSSOM scrolling method—`scroll()`, `scrollTo()`, `scrollBy()`, or `scrollIntoView()`. Other methods of scrolling, such as with a pointer device, aren't affected by the `scroll-behavior` property.

Initially, the value of `scroll-behavior` is `auto`. This value causes an **instant scroll**. An instant scroll is less of a *scroll* and more of a *jump* to the desired location within the document. You can also use `smooth`, which causes what we generally think of as scrolling behavior: the content glides to the requested point.

What Is a Scrolling Box?

Smooth scrolling only works when the element has a “scrolling box”. An element or a viewport has a **scrolling box** when:

- the element or viewport has a scrolling mechanism
- the element overflows its content area *and* the used value of the `overflow-x` or `overflow-y` property is something other than `hidden` or `clip`

Here’s an example:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>scroll-behavior</title>

        <style>
            ul {
                display: flex;
                list-style: none;
                padding: 1rem;
                gap: 1rem;
                justify-content: center;
            }

            .scroll-container {
                width: 50%;
                margin: auto;

                /*
                 * When the container has a fixed height, its
contents may
                 * overflow it
                 */
                height: 30rem;

                /*
                 * Can also use `overflow` as a property.
Value can also be
                 * `scroll`
                 */
                overflow-y: auto;

                scroll-behavior: smooth;
            }

            /*
             * Remember, motion can make some people dizzy!
             */
            @media screen and ( prefers-reduced-motion ) {
                .scroll-container {
                    scroll-behavior: auto;
                }
            }

            .scroll-container div {
                /*
                 * Total height of child divs are large
enough or contain
                 * enough content to overflow their parent
container
                 */

                height: 50vh;
            }
        </style>
    </head>
    <body>
        <nav>
            <ul>
                <li><a href="#A">A</a></li>
                <li><a href="#B">B</a></li>
                <li><a href="#C">C</a></li>
                <li><a href="#D">D</a></li>
            </ul>
         </nav>

        <div class="scroll-container">
            <div id="A">A</div>
            <div id="B">B</div>
            <div id="C">C</div>
            <div id="D">D</div>
        </div>
    </body>
</html>
```

Clicking any of the navigation links triggers a smooth, gliding-style scroll to the linked `<div>`.

For non-root elements, the element must have a fixed height, set using the `height` or `max-height` properties in order for `scroll-behavior` to have an effect. The value of `overflow` should also be `scroll` or `auto`. The root `<html>` element takes on the dimensions of the viewport. Content that exceeds the viewport dimensions creates both an overflow and a scrolling mechanism. As a result, `scroll-behavior: smooth` works for the `<html>` element without any additional properties.

The `<body>` element, however, is a bit of a special case. Body elements rarely generate a scrolling box, in part because content rarely overflows them. Body elements grow and shrink with the dimensions of their child content. That means if you apply `scroll-behavior` to a `<body>` element, you'll also need to set the `height`, `width`, and `overflow` values of its parent. Applying `scroll-behavior` to the root element, on the other hand, works similarly, and requires fewer bytes.

Unfortunately, this version of the CSSOM specification didn't include a way for developers to have control over how the page scrolls beyond `smooth` and "not-smooth" (better known as the `auto` value). We can't use an easing function, or set a scroll duration.

Instead, `scroll-behavior: smooth` uses a browser-defined timing function and duration. Depending on the browser, `scroll-behavior` may also follow conventions of the operating system on which it runs. If you want to shape *how* the scroll works—for example, whether the timing function is `linear` or `ease-in`, or how many milliseconds it should take—you'll need to use JavaScript.

Scroll Snap

Sometimes the inability to control what happens when scrolling ends is the problem you'd like to solve. For example, it can be tricky to create a slide show that always centers a photo without doing a lot of potentially intensive

DOM operations. This is where CSS Scroll Snap shines. Scroll Snap makes it easy to set the alignment of content within a scroll container.

Scroll Snap versus Scroll Snap Points

You may have heard of an earlier, similar feature known as “Scroll Snap Points”. Scroll Snap Points was deprecated in favor of Scroll Snap. You may see some old web tutorials that reference the earlier specification. One clear way to tell whether the tutorial is up to date is to look for the presence of properties with a `scroll-snap-points` prefix, or mentions of `scroll-snap-type-x` and `scroll-snap-type-y`. The MDN Web Docs have a guide to Scroll Snap browser compatibility.

Creating a Scroll Snap Container

Again: an element is a scroll container when the total width and/or height of its children (along with any margins or gaps) overflows its horizontal or vertical dimension. To make it a scroll *snap* container, we need to add the `scroll-snap-type` property. Here’s an example. First, let’s look at our markup:

```
<div class="scroll-container">
    <p>A</p>
    <p>B</p>
    <p>C</p>
    <p>D</p>
    <p>E</p>
    <p>F</p>
    <p>G</p>
    <p>H</p>
    <p>I</p>
</div>
```

This markup is simple. We have a containing `<div>` element, with a class name of `scroll-container`. Its children are all `<p>` elements. We we could just as well use `<div>` or `<img>` elements as child elements. Here’s the CSS:

```
.scroll-container {
    display: flex;
    gap: 2rem;
    /*
     * These two properties force the element to be a
scrolling
     * container. The value of overflow should be auto or
scroll
     */
    width: 100vw;
    overflow-x: auto;

    /*
     * Makes the element a Scroll Snap container
     */
    scroll-snap-type: x mandatory;
}

.scroll-container p {
    /*
     * Makes every flex item exactly 40vw. The total width of
     * the children overflows the width of the container
     */
    flex: 0 0 40vw;

    /*
     * Where to snap after scrolling
     */
    scroll-snap-align: none center;
}
```

The `scroll-snap-type` property indicates two things: the Scroll Snap axis and its strictness. In this example, we've used `scroll-snap-type: x mandatory`. That first value, `x`, means that our container will scroll horizontally; `mandatory` indicates that the container *must* snap to a position when it isn't being scrolled.

You *must* specify the scroll axis for `scroll-snap-type`. In addition to `none` (the initial value), and CSS global values, the value of `scroll-snap-type` can be any of the following:

- `inline`: scrolls in the inline direction—horizontally for languages written and read from left to right or right to left, and vertically for vertical writing modes
- `block`: scrolls in the block direction—vertically for languages written and read from left to right or right to left, and horizontally for vertical writing modes

- `x`: scrolls horizontally
- `y`: scrolls vertically
- `both`: scrolls both horizontally and vertically

Since the scroll axis is a mandatory value, it must be listed first. The Scroll Snap strictness value, on the other hand, is optional. Its value can be one of the following:

- `none`: no snapping occurs
- `proximity`: *may* snap to a position after a scrolling operation completes, at the discretion of the user agent
- `mandatory`: *must* snap to a snap position when there's no scrolling operation in progress

If you don't indicate the strictness of a Scroll Snap, the browser uses `proximity`. Using `mandatory`, however, ensures that the browser snaps to the position you indicate.

The `scroll-snap-type` property sets the scrolling behavior on the containing element. For the snap to work, we also need to indicate how child elements should be aligned within the container when a scrolling operation completes.

Aligning Scrolled Elements with `scroll-snap-align`

As shown in the example above, Scroll Snap also requires the `scroll-snap-align` property. This property gets applied to the children of a Scroll Snap container. It accepts up to two values: the first indicates the snapping alignment along the block axis, while the second indicates the snapping alignment along the inline axis:

```
.scroll-container p {
    scroll-snap-align: none center;
}
```

The CSS above says to the browser: "Don't snap vertical scrolling, and snap horizontal scrolling to the center position of each child element." The image below shows the effect. Our child element is centered within the Scroll Snap container.

The following properties are valid `scroll-snap-align` values, and the property accepts up to two snap alignment positions:

- `none`: initial value; don’t snap scroll in either direction
- `start`: snap to the start of the element’s box

- `end`: snap to the end of the element’s box
- `center`: snap to the middle of the element’s box

When `scroll-snap-align` has a single position, that value is used for both axes. In other words, `scroll-snap-align: center` is the same as `scroll-snap-align: center center`. Whether `start` and `end` align to the right or left edge depends on the writing mode of the document.

Let’s change the value of `scroll-snap-align` from `none center` to `none start`. Now, instead of our child element being centered within the snapport, its starting edge is aligned to the starting edge of the container, as pictured below.

Using `scroll-snap-align: none end` or `scroll-snap-align: end`, on the other hand, aligns the ending edge of our child element to the ending edge of the container.

Don’t Break Scrolling!

Avoid using `hidden` with the overflow properties. Yes, it eliminates scrollbars. Unfortunately, `overflow: hidden` also breaks keyboard and pointer device scrolling in Firefox and Safari. Using arrow or `Page Up`/`Page Down` keys won't work. Neither will scrolling via a trackpad, mouse, or swipe gesture. Using `overflow: hidden` also breaks gesture and pointer device scrolling in Chromium-based browsers.

Instead, use the `scrollbar-width` property to hide scrollbars. The `scrollbar-width` property accepts three possible values: `none`, `thin`, and `auto`, which is the initial value. Adding `scrollbar-width: none` to a Scroll Snap container hides the scrollbar while maintaining the ability to scroll the container.

To date, Firefox (versions 64 and later) is the only browser to support `scrollbar-width`. At the time of writing, Chromium's support is in progress. Until it's ready, use the `::-webkit-scrollbar` pseudo-class and `display: none` to hide scrollbars in both Chromium and WebKit-based browsers. Here's an example:

```
.scroll-container::-webkit-scrollbar {
    display: none;
}
.scroll-container {
    display: flex;
    width: 70vw;
    overflow-x: auto;
    scroll-snap-type: inline mandatory;
    scroll-padding: 0rem;
    scrollbar-width: none;
}
```

If you do hide scrollbars, offer some visual indication that there's more content available and provide an alternative way to scroll through it. You may, for example, add **Back** and **Forward** buttons and define a click handler that uses the `Element.scrollBy()` function. For example:

```
const moveIt = ( evt ) => {
    /* If this is not a scrolling control, do nothing */
    if( evt.type === 'click') {
        if( ! evt.target.classList.contains('scroll-trigger')
) return;
    }

    /* Divide by 2 to reduce the distance scrolled */
    let xScrollBy = ( scrollContainer.clientWidth / 2 );

    /* Negate the scrollBy value if the back arrow was
clicked */
    if( evt.target.dataset.direction === 'back' ) {
        xScrollBy = xScrollBy * -1;
    }

    const scrollContainer = document.querySelector('.scroll-
container');

    /*
     scrollBy can accept arguments in the form (x-distance,
y-distance ),
     or a dictionary that contains `top`, `left`, and/or
`behavior`
     properties.

     Instead of setting `behavior` as an option, we could
also apply
     the CSS `scroll-behavior` property to `.scroll-
container`.
    */
    const scrollByOptions = {
        left: xScrollBy,
        behavior: 'smooth'
    }
    scrollContainer.scrollBy( scrollByOptions );
}

/* Take advantage of event delegation */
document.body.addEventListener( 'click', moveIt );
```

This ensures that your site's visitors know that there's more to view.

When using navigation buttons, as shown here, you'll also need to indicate how the scroll should behave. Browsers treat a button click more like a link click than a scroll wheel; the scrolling behavior will be an instant scroll, instead of a smooth one. To fix this, add `scrolling-behavior: smooth` to the container element, or include `behavior: 'smooth'` in the dictionary parameter that you pass to the scrolling function.

Optimizing the Scroll Viewing Area with `scroll-padding`

Sometimes you'll want to ensure that your content isn't obscured by a fixed or absolutely positioned element. For instance, you may have controls that are positioned at the bottom of the container, as shown in the image below, which depicts a Scroll Snap container with scrolling controls for the container. The controls are absolutely positioned, within an element that contains both the Scroll Snap container and buttons.

This is when the `scroll-padding` property comes in handy. As the specification explains, the `scroll-padding` property defines the optimal viewing region of a scrollport. It adds space within the scrolling *container*, but doesn't change its dimensions. This is true even if the computed value of its `box-sizing` property is `border-box`.

Scrollports and Snapports

A **scrollport** is the viewable area of a scrolling container. The **snapport** is the scrollport plus any offsets specified with `scroll-padding`.

Let's look at an example. We'll use markup from earlier in this section, and add additional elements for our controls:

```
<div class="slideshow">
    <p class="controls">
        <button type="button" data-
direction="back">Up</button>
        <button type="button" data-
direction="forward">Down</button>
    </p>

    <div class="scroll-container">
        <p>A</p>
        <p>B</p>
        <p>C</p>
        <p>D</p>
        <p>E</p>
        <p>F</p>
        <p>G</p>
        <p>H</p>
    </div>
</div>
```

Let's pair the above markup with the CSS shown below:

```
.slideshow {
    position: relative;
}
.controls {
    position: absolute;
    bottom: 0;
    z-index: 1;
    display: flex;
    gap: 2rem;
    justify-content: center;
    width: 100%;
}
.scroll-container {
    width: 70vw;
    height: 50vh;
    margin: auto;
    overflow-y: auto;
    scroll-snap-type: block mandatory;
}
.scroll-container p {
    margin: 0;
    height: 80%;
    scroll-snap-align: end none;
}
```

This gives us the layout we saw pictured at the beginning of this section. Without `scroll-padding`, each child element gets partially obscured by the controls at the end of each scrolling operation. You can see this in the image below.

Let’s add a `scroll-padding` declaration to our scroll container:

```
.scroll-container {
    width: 70vw;
    height: 50vh;
    margin: auto;
    overflow-y: auto;
    scroll-snap-type: block mandatory;

    /* Total height of the control container and its vertical
margin */
    scroll-padding: 0 0 68px;
}
```

Now at the end of each scrolling operation, the end of the child element aligns with the edge of the Scroll Snap container, plus 68 pixels.

As you may have gathered from its syntax, `scroll-padding` is a shorthand property for physical longhand properties. Values follow the same order and syntax as `margin` or `padding`: top, right, bottom, left. You can use lengths or percentages for values, but negative values are invalid.

All Scroll Containers

The `scroll-padding` property applies to scroll containers of every type, not just Scroll Snap containers.

If you only want to set padding along a single edge, use physical properties instead. The physical longhand `scroll-padding` properties are as follows:

- `scroll-padding-top`
- `scroll-padding-right`
- `scroll-padding-bottom`
- `scroll-padding-left`

For padding that adjusts with the writing mode, use flow-logical longhand properties:

- `scroll-padding-inline-start`
- `scroll-padding-block-start`
- `scroll-padding-inline-end`
- `scroll-padding-block-end`

Like other logical properties, which edge these properties affect depends on the document's writing mode. In languages written horizontally from left to right, `scroll-padding-inline-start` is the left edge of the container and `scroll-padding-block-start` is its top edge. For horizontal, right-to-left languages, `scroll-padding-inline-start` is the right edge. For vertical, right-to-left languages, `scroll-padding-block-start` is the right edge. For vertical, left-to-right languages, it's the left.

Using `scroll-padding` only affects a scrolling axis when there's something to scroll. Adding left or right scroll padding has no effect when using `scroll-snap-type: y`. Similarly, `scroll-padding-top` makes no difference when the scroll direction is horizontal.

Shifting Box Alignment with `scroll-margin`

The `scroll-margin` property, on the other hand, applies to the children of a scroll container. It adjusts the area of the box that gets aligned to the snapport.

It doesn’t change the dimensions of the box to which it’s applied, but instead shifts its alignment position by the provided length. Let’s revisit our CSS from the `scroll-snap-align` section:

```
.scroll-container p {
    scroll-snap-align: none center;
}
```

Without `scroll-margin`, each child of a Scroll Snap container will be centered within the snapport, as pictured below.

Let’s add a right `scroll-margin` value of 200 pixels:

```
.scroll-container p {
    scroll-snap-align: none center;
    scroll-margin: 0 200px 0 0;
}
```

Now the center of our child element is shifted to the left by 200 pixels, due to this extra right margin, as shown below.

Values for `scroll-margin` must be lengths. Negative values are perfectly valid, and `scroll-margin` uses the same ordering as `margin`. Let’s change our right `scroll-margin` value to a negative value.

```
.scroll-container p {
    scroll-snap-align: none center;
    scroll-margin: 0 -200px 0 0;
}
```

Now the center of our child element has been pulled 200 pixels to the right, as pictured below. This is, in effect, the same as using a positive `scroll-margin-left` value.

Much like `scroll-padding`, the `scroll-margin` property is a shorthand for the physical longhand properties. Values are ordered the same way as `margin` and `scroll-padding` too: top, right, bottom, and left. You can also specify a margin value based on the inline or block direction:

- `scroll-margin-top`
- `scroll-margin-right`
- `scroll-margin-bottom`
- `scroll-margin-left`

Logical longhand properties are:

- `scroll-margin-inline-start`
- `scroll-margin-block-start`
- `scroll-margin-inline-end`
- `scroll-margin-block-end`

Here, too, `scroll-margin` only affects the margin of a box along the axis of the overflowing content. If the scrolling direction is vertical, adding `scroll-margin-left`, `scroll-margin-right`, or either of the `scroll-margin-inline-*` properties, won't affect box alignment within the scrollport.

To date, there isn't a way to change the Scroll Snap timing function using CSS. If you want that level of control, you'll still need to use JavaScript.

Conclusion

You've made it to the end of this chapter! You should now know:

- how to create smooth scrolling jump links without JavaScript
- how to create a carousel or slide show using a minimal amount of JavaScript

There's an area of CSS that's a little more experimental than the features we've discussed in the book so far: using CSS with SVG. SVG (which stands for Scalable Vector Graphics) uses markup to describe how images should be rendered onscreen. Because SVG is markup, we can use many CSS properties with SVG elements. We can also express some SVG attributes using CSS properties. We'll dig into the details in the next chapter.

Chapter 12: SVG

So far, we've talked about using CSS with HTML, but we can also use CSS with SVG, or *Scalable Vector Graphics*. SVG is a markup format for describing flat, two-dimensional images. Because it's a markup language, it has a Document Object Model, and can be used with CSS.

By using CSS with SVG, we can change the appearance of SVG based on user interaction. Or we can use the same SVG document in multiple places, and show or hide portions of it based on the width of the viewport.

All major browser engines support the SVG 1.1 specification, and they have done for years. Support for features of SVG 2, on the other hand, is still a work in progress. Some of what we'll discuss here has limited browser support at the time of writing. That may have changed by the time you're reading this. Keep an eye on the Chromium meta issue—Implement SVG2 features—to track development progress in Chromium-based browsers. Watch the Support SVG 2 features meta issue to follow Firefox's implementation work, and WebKit's Implement SVG 2 meta issue for Safari. Issue trackers can be unpleasant to navigate, but for now they're the best way to track SVG 2 support.

Before we go any further, however, let's talk about what SVG is and why you should use it.

Vector Images versus Raster Images

Most of the images currently used on the Web are raster images, also known as bitmap images. **Raster images** are made up of pixels on a fixed grid, with a set number of pixels per inch. JPEG, WebP, GIF, and PNG are all examples of raster image formats.

Raster images are *resolution dependent*. A 144 PPI (pixels-per-inch) PNG image looks great on a device with a 144 PPI display resolution. When viewed on a higher resolution, 400 PPI display, however, that same image can look fuzzy. Raster images also have fixed dimensions and look best at their original size. Scaling a 150 by 150 pixel image up to 300 by 300 pixels distorts it.

Instead of using pixels on a grid, vector image formats describe the primitive shapes—circles, rectangles, lines, or paths—that make up an image, and their placement within the document's coordinate system. As a result, vector images are *resolution independent*, and retain their quality regardless of display resolution or display dimensions.

Resolution independence is the biggest advantage of SVG. We can scale images up or down with no loss of quality. The same image looks great on both high and low PPI devices. That said, SVG is poorly suited to the amount of color data required for photographs. It's best for drawings and shapes. Use it in place of PNG and GIF images, and as a more flexible replacement for icon fonts.

Another advantage of SVG is that it was designed to be used with other web languages. We can create, modify, and manipulate SVG images with JavaScript. Or, as we'll see in this chapter, we can style and animate SVG using CSS.

Associating CSS with SVG Documents

Using CSS with SVG is a lot like using it with HTML. We can apply CSS using the `style` attribute of an SVG element, group CSS within a document using the `<style>` element, or link to an external stylesheet. The pros and cons of each method are the same as when using CSS with HTML.

Using the `style` Attribute

Here's a simple SVG document where the code creates a black circle:

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg"
viewBox="0 0 200 200" enable-background="new 0 0 200 200">
    <circle cx="101.3" cy="96.8" r="79.6" />
</svg>
```

The image below shows how that code renders in a browser.

Let’s give our circle a pink fill using CSS and the `style` attribute:

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg"
viewBox="0 0 200 200" enable-background="new 0 0 200 200">
    <circle cx="101.3" cy="96.8" r="79.6" style="fill: #f9f"
/>
</svg>
```

The effect of this is shown below.

Here's one difference between using CSS with HTML and using it with SVG: property names. Many of the CSS properties that we use with HTML documents aren't compatible with SVG, and vice versa. We'll come back to this point later in the chapter.

Using the `style` attribute isn't the best way to use CSS, of course. Doing so limits the ability to reuse those styles across multiple elements or documents. Instead, we should use inline or linked CSS.

Embedding CSS in SVG Documents

Instead of using the `style` attribute, we can use the `<style>` element:

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg"
viewBox="0 0 200 200" enable-background="new 0 0 200 200">
    <style type="text/css">
        circle {
            fill: #0c0;
        }
    </style>
    <circle cx="101.3" cy="96.8" r="79.6" />
</svg>
```

Embedding CSS in an SVG document lets us reuse those styles for multiple elements within the same document, but it prevents that CSS from being shared across multiple documents. That's fine for logos and icons. But if you're creating something like a library of chart styles, an external CSS file is a better bet.

Using a standard text editor, you can also add CSS to SVG images created with software such as Sketch, Inkscape, or Illustrator. Doing so won't affect your ability to edit the image with the drawing application, but if you edit the file using image software, the application may rewrite or remove your CSS.

Linking from SVG to an External CSS File

As with HTML, linking to an external CSS file makes it possible to share styles across several SVG documents. To link an external CSS file, add `<?xml-stylesheet ?>` to the beginning of your SVG file:

```
<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet href="style.css" type="text/css"?>
<svg version="1.1" xmlns="http://www.w3.org/2000/svg"
viewBox="0 0 200 200" enable-background="new 0 0 200 200">
    <circle cx="101.3" cy="96.8" r="79.6" />
</svg>
```

Using the `<link>` Element

Alternatively, use the HTML `<link>` element. If you do use this method, you'll need to include the `xmlns` namespace attribute, as shown below:

```
<link href="style.css" type="text/css" rel="stylesheet"
xmlns="http://www.w3.org/1999/xhtml" />
```

Older Browsers and `<link>`

Some older browsers need the `<link>` element to be enclosed by `<defs>` or `<g>` tags.

The `<link>` element isn't an SVG element. It belongs to HTML and XHTML. XHTML is a variant of HTML that's parsed according to the rules of XML markup. According to the rules of XML, we can borrow elements and their behavior from other XML dialects, such as XHTML. To do so, however, we need to tell the browser which namespace the element belongs to using the `xmlns` attribute.

Using `@import`

We can also link to an external stylesheet by using `@import` inside `<style>` and `</style>` tags:

```
<style type="text/css">
@import('style.css');
</style>
```

This method functions similarly to the `<link>` method.

SVG and the `<img>` Element: Limitations

Linking from SVG files to external assets, including CSS files, doesn't work with the `<img>` element. This is a security limitation of the `<img>` element that's baked into browsers.

If you'd like to use linked CSS with your SVG images, you'll need to do either of these two things:

- use the `<style>` element in your SVG document to place your CSS inline
- use an `<iframe>` or `<object>` element (see note below)

Using `<iframe>` and `<object>`

Craig Buckler's tutorial "How to Add Scalable Vector Graphics to Your Web Page" discusses using `<iframe>` and `<object>` in detail.

In general, you should use `<iframe>` over `<object>`. However, the `<object>` element can be the child of an `<a>` element, while `<iframe>` can't. Using `<iframe>` or `<object>` also makes the SVG document tree available to the parent document's document tree. This means that we can use JavaScript to interact with it (for example, with `document.querySelector('iframe').contentDocument`).

Inline SVG and External Assets

When adding SVG to HTML, the browser won't load external assets referenced by the SVG document. We can, however, link to CSS for our SVG document from the `<head>` of our HTML document:

```
<head>
    ⋮
    <link href="svg.css" type="text/css" rel="stylesheet" />
</head>
```

SVG elements within HTML documents also become part of the HTML document tree. If you're using inline SVG, it's perfectly fine to combine your HTML-related and SVG-related CSS in the same stylesheet.

Differences between SVG and HTML

While SVG and HTML are both markup languages, there are two significant differences between them that affect how they work with CSS:

- SVG doesn't adhere to the CSS box model
- SVG lacks a positioning scheme

SVG Doesn't Adhere to the CSS Box Model

When used with HTML, CSS layout follows the rules of the CSS box model. SVG, on the other hand, uses coordinates for layout. It adheres to what may be best understood as a "shape model".

SVG shapes aren't limited to rectangular boxes. As a result, most box-model–related properties don't apply to SVG elements. You can't, for instance, change the `padding` or `margin` of an SVG element. Nor can you use the `box-sizing`, `box-shadow`, `outline`, or `border-*` properties. Grid layout, floats, and Flexbox also don't work.

You can, however, use CSS to set or change a range of SVG properties and attribute values. The full list is outlined in the SVG 2 specification, although support in most browsers is incomplete. Some CSS properties, such as `filter`, can be used with SVG or HTML. We'll discuss a few of them in this chapter, within the context of specific techniques.

SVG Lacks a Positioning Scheme

When CSS is used with HTML, element boxes can:

- exist within a normal flow
- be removed from normal flow with the `float` property
- be removed from normal flow with the `position` property

The CSS specification refers to these as *positioning schemes*. Positioning schemes don't exist in SVG. The `position` property has no effect on SVG elements. Neither do properties such as `top`, `left` and `bottom`, which depend on elements being positioned. You also can't float elements within an SVG document.

Instead, SVG uses a coordinate system for element placement. To create a `<circle>`, for example, you need to set its center point coordinates using the `cx` and `cy` attributes, and set a radius length using the `r` attribute. A polygon consists of a series of point coordinates and line segments drawn between them. In other words, you can define where an element will be drawn to the SVG canvas, but you can't "position" them in the CSS sense of the word.

Related to positioning schemes, SVG also lacks the idea of `z-index` and stacking contexts. SVG elements are instead stacked according to their source order. Those that appear later in the document sit towards the top of the stack. If you want to change the stacking order of SVG elements, you'll need to move them around in the source or use JavaScript to reorder them in the DOM tree.

SVG 2 and `z-index`

The SVG 2 specification *does* define behavior for `z-index` and stacking contexts in SVG documents, but most browsers don't yet support it.

In fact, most CSS 2.1 properties don't apply to SVG documents. Exceptions include animations and transforms, `display`, `overflow`, `visibility`, `filter`, and a few font and text-related properties. Instead, you'll have to use SVG-specific styling properties with SVG documents. Most of these properties can also be expressed as SVG element attributes.

Styling SVG Elements

Here's a simple example of how to style SVG elements using CSS. First our SVG document, which is a stand-alone file:

```
<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet href="styles.css" type="text/css" ?>
<svg version="1.1" xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" x="0px" y="0px"
viewBox="0 0 497 184" enable-background="new 0 0 497 184"
xml:space="preserve">
    <polygon id="star" points="77,23.7 98.2,66.6 145.5,66.5
111.2,106.9,119.3,154 77,131.8 34.7,154 42.8,106.9 8.5,67.5
55.8,66.6 "/>
```

```
    <circle id="circle" cx="245" cy="88.9" r="67.5"/>
</svg>
```

This markup creates the image shown below.

Although we can’t use most CSS properties with SVG documents, we can use CSS to change an element’s color. Let’s make our star yellow:

```
#star {
    fill: hsl( 44, 100%, 50% );
}
```

You’ll often see the `fill` attribute used with SVG tags—for example, `<circle fill="rgb( 255, 185, 0 )" cx="3" cy="10" r="100">`—but it’s also a property that can be used with CSS.

We can also use CSS to adjust an element’s `stroke`, which is the outline of an SVG shape. A shape’s stroke exists, even if no `stroke` properties are set. Let’s give our circle a dark blue, dashed border that’s ten pixels wide. We’ll also set its `fill` property to `cornflowerblue`:

```
circle {
    fill: cornflowerblue;
    stroke: darkblue;
    stroke-width: 10;
    stroke-dasharray: 10, 15;
    stroke-linecap: round;
}
```

Together this gives us the result below.

Using SVG Attributes as CSS Properties

We can also use CSS to set the coordinate values of some shape elements: `<rect>`, `<circle>`, and `<ellipse>`. Typically, we'd use SVG attributes for these elements:

```
<svg viewBox="0 0 400 400"
xmlns="http://www.w3.org/2000/svg">
    <rect x="20" y="200" width="300" height="300" fill="#f60"
/>
</svg>
```

However, SVG 2 redefined some SVG attributes as geometry properties. This means we can use CSS to set their values:

```
<svg viewBox="0 0 400 400"
xmlns="http://www.w3.org/2000/svg">
    <style type="text/css">
      rect {
          x: 20px;
          y: 50px;
          width:  300px;
          height: 300px;
          fill: #f60;
      }
    </style>
    <rect />
</svg>
```

Coordinate properties (`x` and `y`), center coordinate properties (`cx` and `cy`), and radius properties (`rx`, `ry`, and `r`), can be set using CSS. So can `width` and `height`. Units are optional for SVG attributes. CSS values, on the other hand, *require* units. Both lengths and percentages are valid for the properties mentioned here, but be aware that lengths work a bit differently with SVG documents. Remember that the S in SVG stands for *scalable*. The computed size of an SVG element also depends on:

- the computed `width` and `height` of the root SVG element
- the value of the root element's `viewBox` attribute
- any scaling transforms applied to the element or its ancestors

In other words, the corners of our `<rect>` element are `(20, 50)`, `(20, 320)`, `(350, 320)`, and `(20, 350)` within the SVG coordinate system. However, the *actual* dimensions may be larger or smaller, depending on the factors above.

Not every SVG attribute is available via CSS—at least not in every browser. For example, Chrome and Edge support using the CSS `path()` function to set path data, or the `d` attribute:

```
path {
    d: path("M 454.45223,559.21474 -304.96705,163.45948
417.4767,-296.33928 Z");
}
```

As of this writing, they are the only browsers that do. Work to add support in Firefox and WebKit has not yet begun.

For other shape elements, the SVG 2 specification is downright inconsistent. To date, you *must* use element attributes to set the properties of `<line>`, `<polyline>`, and `<polygon>` elements.

That said, we aren't limited to using type (or element) selectors to set properties. We could, for instance, define small, medium, and large circles using class selectors:

```
<svg viewBox="0 0 400 400"
xmlns="http://www.w3.org/2000/svg">
    <style type="text/css">

    .small {
        cx: 20px;
        cy: 20px;
        r:  20px;
        fill:  #0c0;
      }

      .medium {
        cx: 80px;
        cy: 80px;
        r:  60px;
        fill:  #fc0;
      }

      .large {
        cx: 220px;
        cy: 220px;
        r:  120px;
        fill: #00f;
      }

    </style>
    <circle class="small" />
    <circle class="medium" />
    <circle class="large" />
</svg>
```

Regardless of the selector, using CSS syntax to specify properties also makes it easy to animate them. We'll take a look at how to do this in the next section.

Animating and Transitioning SVG CSS Properties

Using CSS with SVG becomes more interesting when we add transitions and animations to the mix. The process is just like animating HTML elements with CSS, but with SVG-specific properties. Let's create a twinkling star effect using the following SVG document:

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg" x="0px"
y="0px" viewBox="0 0 497 184" xml:space="preserve">
    <defs>
        <link href="twinkle.css" type="text/css"
rel="stylesheet" xmlns="http://www.w3.org/1999/xhtml"/>
    </defs>
    <polygon class="star" points="77,23.7 98.2,66.6
145.5,66.5 111.2,106.9 119.3,154 77,131.8 34.7,154 42.8,106.9
8.5,67.5 55.8,66.6 "/>
    <polygon class="star twinkle" points="77,23.7 98.2,66.6
145.5,66.5 111.2,106.9 119.3,154 77,131.8 34.7,154 42.8,106.9
8.5,67.5 55.8,66.6 "/>
</svg>
```

Our document contains two star-shaped polygon elements, each with a class name of `star`. To create the twinkling effect, we'll animate the first one. Here's our CSS:

```
@keyframes twinkle {
    from {
        fill-opacity: .4;
    }
    to {
        fill-opacity: 0;
        transform: scale( 2 );
    }
}
```

```
.star {
    fill: rgb( 255,195,0 );
    transform-origin: 50% 50%;
}
.twinkle {
    animation: twinkle 1.5s infinite forwards ease-in;
}
```

Here we've used the SVG-specific property `fill-opacity`. As with CSS, if we can interpolate the value of an SVG styling property, we can animate or transition it. You can see two different points of the animation in the image below.

Let’s look at another example. This time we’ll create a drawing effect by transitioning the `stroke-dasharray` property. Here’s our SVG document:

```
<svg version="1.1"
xmlns="http://www.w3.org/2000/svg"xmlns:xlink="http://www.w3.
org/1999/xlink" x="0px" y="0px"
        viewBox="0 0 200 200" enable-background="new 0 0 200
200">
    <circle fill="transparent" stroke-width="16" cx="101.3"
cy="96.8" r="79.6"/>
</svg>
```

The `stroke-dasharray` property accepts a comma-separated list of length or percentage values that create a dashed pattern. Odd-numbered values determine the dash length. Even-numbered values determine the gap length. A `stroke-dasharray` value of `5, 10` means that the stroke will be `5px` long with a gap of `10px` between each dash. A value of `5, 5, 10` alternates `5px` and `10px` dash lengths with `5px` gaps in between.

We can use `stroke-dasharray` to create a drawing effect by starting with a zero dash length and a large gap, and ending with a large dash length and a dash gap of zero. Then we’ll transition between the two. Here’s what our CSS looks like:

```
circle {
    transition: stroke-dasharray 1s ease-in;
    fill: transparent;
    stroke-dasharray: 0, 500;
}
.animate {
    stroke-dasharray: 500, 0;
}
```

At the beginning of the transition, our stroke is invisible because the dash length is `0` and our gap is `500`. But when we add the `animate` class to our circle, we shift the dash length to `500` and eliminate the gap. The effect is a bit like drawing a circle with a pair of compasses. Why 500? It’s the smallest value that worked to create this particular effect.

An Animated Path Future

Remember our example of defining a path via CSS from the previous section? Someday, we may be able to animate paths in every browser, using CSS:

```
path {
    d: path("M357.5 451L506.889 192.25H208.111L357.5 451Z");
    transition: d 1s ease-in-out;
}
.straighten {
    d: path("M357.5 8871L406 -10113.75H208.111L357.5 351Z");
}
```

To date, however, only Chromium-based browsers such as Google Chrome and Microsoft Edge support animating path definitions in this way. To make this work in other browsers, use a JavaScript library such as GreenSock and its MorphSVGPlugin. In addition to its cross-browser compatibility, GreenSock and the MorphSVGPlugin also make it much easier to morph between two shapes regardless of the number of points in each.

Using SVG with Media Queries

With HTML documents, we might show, hide, or rearrange parts of the page based on the conditions of the viewport. If the browser window is 480 pixels wide, for example, we might shift our navigation from a horizontal one to a vertical, collapsible list. We can do something similar with media queries and SVG documents. Consider a logo, such as that of the fictitious Hexagon Web Design & Development pictured below.

Without media queries, this SVG logo would simply stretch or shrink to fit the viewport or its container. But with media queries, we can do more clever things.

Let’s distinguish between the HTML document viewport and the SVG document viewport. When SVG is inline, the HTML viewport and the SVG viewport are one and the same. The SVG document behaves like any other HTML element. On the other hand, when an SVG document is linked—as with the `<iframe>`, `<object>` or `<img>` elements—we’re dealing with the SVG document viewport.

Media queries work in both cases, but when the SVG document is linked, its viewport is independent of its HTML document. In that case, the size of the browser window doesn’t determine the size of the SVG viewport. Instead, the viewport size is determined by the dimensions of the `<object>`, `<iframe>`, or `<img>` element. Take the abridged SVG document that follows as an example (a full demonstration of this technique, including the complete source of this SVG document, is available in the code archive.):

```
<svg version="1.1" id="HexagonLogo"
xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" x="0px" y="0px"
viewBox="0 0 555 174" xml:space="preserve">
    <defs>
        <style type="text/css">
        /* CSS goes here */
        </style>
    </defs>
    <g id="hex">
        <polygon id="hexagonbg" points="55.2,162 10,86.5
55.2,11 145.5,11 190.7,86.5 145.5,162  "/>
        <path id="letterH" fill="#FFFFFF"
d="M58,35.5h33v35.2h18.4V35.5 h33.2v103.4h-33.2v-
38.3H91v38.3H58V35.5z M77.5,126.5V87.3h45.6v39.2h4V47.9h-
4v35.6H77.5V47.9h-4v78.6H77.5z"/>
    </g>

    <g id="word-mark">
        <g id="hexagon-word">
            ...
        </g>
        <g id="web-design-and-dev">
            ...
        </g>
    </g>
</svg>
```

In smaller viewports, let’s show just the H in a hexagon symbol:

```
@media (max-width: 320px) {
    [id=word-mark] {
        display: none;
    }
}
```

Now, whenever our SVG’s container is less than or equal to `20em`, only the symbol portion of our logo will be visible.

To trigger this view from the HTML document, set the width of the SVG container:

```
<iframe src="hexlogo.svg" style="width: 320px; border:0">
</iframe>
```

As you may have noticed from looking at the image above, our SVG image retains its intrinsic dimensions even though part of it has been hidden. This, unfortunately, is a limitation of SVG. To fix it, we need to change the `viewBox` attribute of the SVG document, but only when the viewport is below a certain size. This is a great use case for `matchMedia` (which is discussed in Chapter 10, "Applying CSS Conditionally").

The `viewBox` attribute, as its name suggests, determines the viewable area of an SVG element. By adjusting it, we can determine which part of an SVG image fills the viewport. What follows is an example using `matchMedia` and a media query to update the `viewBox` attribute:

```
<script type="text/javascript">
const svg = document.querySelector( 'svg' );

/* Store the original value in a variable */
const originalViewBox = svg.getAttribute( 'viewBox' );

/* Define our media query and media query object */
const mq = matchMedia( '( max-width: 320px )' );

/* Define the handler */
const updateViewBox = () => {
    if (mq.matches) {
        /* Change the viewBox dimensions to show the hexagon
*/
        svg.setAttribute( 'viewBox', '0 0 200 174' );
    } else {
        svg.setAttribute( 'viewBox', originalViewBox );
    }
}

svg.addEventListener( 'SVGLoad', updateViewBox );

/* Fire if the media condition changes */
mq.addEventListener( 'change', updateViewBox );
</script>
```

Now, whenever the SVG container is 320 pixels or less, the value of `viewBox` will be `"0 0 200 174"`. When it exceeds 320 pixels, `viewBox` gets restored to its initial value.

Since this technique uses either the `onload` event attribute or the `SVGLoad` event, it's a good idea to embed our CSS and JavaScript within the SVG file. When CSS is external, the `SVGLoad` event may fire before its associated CSS finishes loading.

Using Media Queries with `background-size`

SVG documents and media queries aren't limited to foreground images. We can also resize the SVG viewport using the CSS `background-size` property.

We'll start with this SVG document:

```
<?xml version="1.0" encoding="utf-8"?>
<svg version="1.1" xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" x="0px" y="0px"
viewBox="-20 -20 250 250" xml:space="preserve">
    <style type="text/css">
        circle {
            stroke: #000;
            stroke-width: 30;
            fill: #009688;
        }
        @media ( width: 100px ) {
            circle {
                fill: #673ab7;
            }
        }
        @media ( width: 300px ) {
            circle {
                fill: #ffc107;
            }
        }
    </style>
    </defs>
    <circle cx="100" cy="100" r="100" />
    <circle cx="100" cy="100" r="50" />
</svg>
```

This is a simple case. Our `<circle>` elements get a new `fill` color at specific viewport widths. When the viewport is 20 pixels wide, the `fill` value is teal. When it's 300 pixels wide, it's yellow.

To make this work, we have to use our SVG image as a background image and set the selector's `background-size` property. In this case, we'll use our image as a background for the `<body>` element and for `<li>` elements:

```
body, li  {
    background: url(circles.svg);
}
body  {
    background-color: #9c27b0;
    background-size: 300px auto;
}
li {
    background-position: left center;
    background-repeat: no-repeat;
    background-size: 1em auto;
    padding-left: 40px;
    font-size: 24px;
    margin: 1rem 0;
}
```

The result is pictured below.

Conclusion

Using SVG with CSS gives us more possibilities for flexible and adaptive documents. Upon completing this chapter, you should now know how to:

- use CSS to style SVG elements
- animate SVG properties
- employ CSS media queries and the `matchMedia` API to show and hide portions of an SVG document

Chapter 13: Conclusion

In this book, we've covered some of the finer points and broad strokes of CSS. In some ways, we've only scratched the surface.

With the CSS Working Group's switch to modularized specifications and shorter browser release cycles, new CSS features are created and implemented much more quickly. Attempting to keep up and stay ahead of the curve can leave your head spinning. Indeed, browsers support CSS features and properties such as generated content and scroll anchoring, which I haven't covered in this book.

So what's coming next? Quite a bit! One caveat: progress on features that seem to be on a fast track can stall over time. Browser vendors shift development priorities and sometimes entire rendering engines based on developer demand, performance, security, and business concerns. In rare cases—such as support for SVG fonts—vendors may remove support altogether.

The following list of upcoming features isn't comprehensive. It's more of a look at a few specifications and implementations in progress.

Nested Grids with `subgrid`

Applying `display: grid` to an element creates a grid formatting context, and turns its immediate child elements into grid items. Children of grid items, however, don't participate in the grid formatting context. Instead, they behave according to the rules of normal flow, as pictured below.

As the image above illustrates, neither child of Item 1 participates in the grid formatting context of its “grandparent” element. By specifying a subgrid, however, we can force our grandchild elements to line up with the grid tracks established by the grandparent element.

First, let’s look at the markup for the layout shown in the image above:

```
<div class="grid">
    <div class="grid-item-1">
        Item 1
        <div class="subgrid-item">Child of Item 1</div>
        <div class="subgrid-item">Child of Item 1</div>
    </div>
    <div class="grid-item-2">Item 2</div>
    <div class="grid-item-3">Item 3</div>
    <div class="grid-item-4">Item 4</div>
</div>
```

The markup is straightforward. Our grid container has four child elements, and its first child has two children of its own.

Here’s the CSS. I’ve removed non-essential declarations such as background colors:

```
.grid {
    gap: 2rem;
    display: grid;
    grid-template-columns: repeat(12, 1fr);
    grid-template-rows: repeat(2, 1fr);
}
/*Spans 10 columns */
.grid-item-1 {
    grid-column: 1 / 11;
}
 /* Spans two columns */
.grid-item-2 {
    grid-column: 11 / 13;
}
/* Spans six columns each */
.grid-item-3 {
    grid-column: 1 / 7;
}
.grid-item-4 {
    grid-column: 7 / 13;
}
```

```
.subgrid-item {
    font-size: .5em;
    padding: .5rem;
}
```

We haven’t yet defined a subgrid for this layout. Adding a subgrid requires adding two declarations to a grid item:

- `display: grid` (or `display: inherit`), which creates a grid formatting context for children of the grid item
- `grid-template-columns: subgrid` or `grid-template-rows: subgrid`

Let’s add a column subgrid for `.grid-item-1`. We’ll also make each child element take up five columns:

```
.grid-item-1 {
    grid-column: 1 / 11;

    /* Adopts the adopts the parent's grid tracks */
    display: grid;
    grid-template-columns: subgrid;
}
.subgrid-item:first-child {
    grid-column: 1 / 6;
}
.subgrid-item:last-child {
    grid-column: 6 / 11;
}
```

Elements within the subgrid align with the grid tracks of the parent grid container, as pictured below.

The child elements of Item 1 now align with the grid tracks of `div.grid`, minus any padding applied to `.grid-item-1`.

You may also have noticed from the screenshots that the Item 1 text wraps in our subgrid. Although that text isn't an element, it participates in the grid formatting context of its parent. As a result, its width is constrained to the width of a single grid track.

Unfortunately, Firefox (versions 71 and above) is the only browser that currently supports subgrid.

Creating Brick-like Layouts with `masonry`

Firefox is also the only browser that currently supports masonry-style grid layouts. It's still experimental at this stage. You'll need to enable it by changing the value of `layout.css.grid-template-masonry-value.enabled` to `true`. You can find this option in Firefox's `about:config` menu.

Masonry-style layouts, also known as [Pinterest-style layouts](), until now have required a JavaScript library such as [Masonry.js](). With the `masonry` grid template value, creating masonry-style layouts requires much less effort:

```
.grid {
    display: grid;
    gap: 1rem;

    /* Short hand for grid-template-rows / grid-template-
columns */
    grid-template: masonry / repeat(6, 1fr);
}
```

This creates the layout shown in the image below.

Rather than add strict tracks for rows (when `grid-template-rows: masonry`) or columns (when `grid-template-rows: masonry`), masonry creates a tightly packed layout.

Grid items shrink to the dimensions of their content. By default, they're arranged where there's available space, which may not match the source order. However, we can change this behavior using the `masonry-auto-flow` property. For example, adding `masonry-auto-flow: next` to the grid container forces items to be arranged in order, as pictured below, where `masonry-auto-flow: next` preserves the order of grid items in a masonry layout.

To experiment with `masonry` while ensuring backward compatibility, separate your `grid-template-rows` and `grid-template-columns` values. Remember: if a browser can't parse a declaration, that declaration gets discarded. Using `grid-template` for both values would make the entire rule fail. Instead, set `grid-template-rows` or `grid-template-columns` to `masonry`, and use the other property to define grid tracks:

```
.grid {
    display: grid;
    gap: 1rem;
    grid-template-columns: repeat(6, 1fr);
    grid-template-rows: masonry;
}
```

In browsers that don't yet support the `masonry` value, the CSS above creates a six-column grid layout. For 12 grid items, you'd see two rows.

Container Queries

Almost as soon as media queries landed in browsers, developers began asking for a way to change the layout of components based on the width of their container, instead of the browser viewport. A few developers have used JavaScript to create [responsive containers]() and [element queries]() that have a similar effect, but browser implementations have never gone beyond the specification phase.

In March of 2021, however, Chrome announced some movement in this space. Google released Chrome Canary with an experimental container queries implementation, based on a [draft specification]() for single-axis containment.

Work on the container queries feature is now part of the [CSS Containment]() specification. We may soon be able to create adaptable layouts using an `@container` rule with a syntax that's similar to `@media`:

```
.simple-input {
    contain: layout inline-size; /* Creates a containment
context for the inline axis */
}
.simple-input input,
.simple-input button {
    display: block;
    font: inherit;
    padding: 2px;
}
@container (min-width: 40rem) {
    .simple-input {
        display: flex;
        gap: 1rem;
    }
}
```

To experiment with container queries today, install Chrome or Chromium. Enable the feature by typing `chrome://flags/#enable-container-queries` in the address bar, and selecting **Enabled** from the **Enable CSS Container Queries** menu.

Both David A. Herron's "Container Queries: a Quick Start Guide" and the Mozilla Developer Network's "CSS Container Queries" are fantastic introductions to container queries. CodePen also has a growing collection of container query demos worth exploring. If poring over technical details is your thing, read Miriam Suzanne's *Container Query Proposal & Explainer*.

How to Follow Changes and Additions to CSS

Keeping track of all this can be overwhelming. Just when you think you're up to date on everything, you find a new spec that you didn't know existed, or an existing spec changes in a significant way. Because specifications and implementations are often in flux, keeping up with changes to CSS can be tough. But it is possible.

The World Wide Web Consortium manages a list of current specifications and their status. One of the best ways to become a CSS expert is to carefully read specifications. Specifications explain how features are *supposed* to work, and can help you recognize browser bugs, or understand what may be going wrong in your CSS.

If you'd like to track the development and discussion of CSS specifications, try the CSS Working Group's GitHub repository. It contains current drafts of specifications, and a list of issues that developers, browser vendors, and specification editors are working through. The CSS Working Group also has a

Twitter account if you'd just like to keep up with developments without getting into the proverbial weeds.

Tracking Browser Support

The Can I use site is the leader in the browser support tracking space. It tracks support for a range of CSS, HTML, SVG and JavaScript features in every major browser across several versions. Can I use shares documentation with the Mozilla Developer Network, another fantastic resource for tracking features as they land.

Chrome Platform Status tracks features as they appear in Google Chrome. Because they both use Chromium, Microsoft Edge hews closely to the feature set and release cycle of Google's Chrome. If Chrome supports a feature, there's a good chance Edge does as well. Other Chromium-based browsers, such as Samsung Internet, have a longer release cycle. Samsung includes the version of Chrome on which the current release is based as part of its release notes.

Apple is notoriously secret about its products. Safari, the web browser for macOS and iOS, is no exception. Safari is, however, based on WebKit, an open-source web browser engine. WebKit Feature Status is a great way to keep up with what's coming to Safari and other WebKit-based browsers.

If you prefer to weigh in on what features browsers should support, you can also follow and comment on issue tickets in the bug trackers of WebKit, Firefox, and Chromium. Developer interest can help vendors prioritize feature development.

Documentation and Tutorials

The Mozilla Developer Network is an amazing resource for web development more generally. Its CSS Reference is perhaps the best on the Web. Almost every property is documented, and each property's page includes examples of use, details whether it's experimental or production-ready, and links to the relevant specification.

For general CSS tricks, tips, and techniques, CSS-Tricks is an excellent resource. The site includes tutorials on CSS and other front-end development topics.

Stephanie Eckles' Modern CSS Solutions is perfect for experienced developers who are still trying to learn the CSS landscape. Its companion site SmolCSS includes snippets.

Newsletters are also a great way to keep track of new CSS features. Rachel Andrew keeps track of layout specifications and implementations in her long-running weekly CSS Layout News. Her newsletter also contains useful pointers to CSS and design-focused resources.

SitePoint.com, too, has a treasure trove of CSS-related material. Its HTML and CSS channel has lots of CSS tutorials, including topics such as Grid, CSS optimization, authoring tools, and progressive enhancement. If you need help, you can always ask a question in the SitePoint Forums.

And that's how this book ends. Of course, just reading this book isn't sufficient for becoming a true CSS master. The best way to achieve mastery is by putting knowledge into practice. My hope is that you've gained a better understanding of a range of CSS topics, including specificity, layout, and project architecture. These topics provide a solid foundation for your journey toward CSS mastery.