MANNING

CSS IN DEPTH

SECOND EDITION

MEAP

Keith J. Grant

CSS in Depth, Second Edition

1 Cascade, specificity, and inheritance

This chapter covers

- The six criteria that make up the cascade
- The difference between the cascade and inheritance
- How to control which styles apply to which elements
- Common misunderstandings about shorthand declarations
- Working with new and upcoming CSS features

CSS is unlike a lot of things in the world of software development. It's not a programming language in the conventional sense, but it does require abstract thought. It's not purely a design tool, but it does require some creativity. It provides a deceptively simple declarative syntax, but if you've worked with it on any large projects, you know it can grow into unwieldy complexity.

When you need to learn to do something in conventional programming, you can usually figure out what to search for (for example, "How do I find items of type x in an array?"). With CSS, it's not always easy to distill the problem down to a single question. Even when you can, the answer is often "it depends." The best way to accomplish something is often contingent on your particular constraints and how precisely you'll want to handle various edge cases.

While it's helpful to know some "tricks" or useful recipes you can follow, mastering CSS requires an understanding of the principles that make those practices possible. This book is full of examples, but it is primarily a book of principles.

Part 1 begins with the most fundamental principles of the language: the cascade, the box model, and the wide array of unit types available. Most web developers know about the cascade and the box model. They know about pixel units and may have heard that they "should use ems instead." The truth

is, there's a lot to these topics, and a cursory understanding of them gets you only so far. If you're ever to master CSS, you must first know the fundamentals, and know them deeply.

I know you're excited to start learning the latest and greatest CSS has to offer. That is the exciting stuff. But first, we'll go back to the fundamentals. I'll quickly review the basics, which you're likely already familiar with, and then dive deep into each topic. My aim is to strengthen the foundation on which the rest of your CSS is built.

In this chapter, I'll begin with the C in CSS—the cascade. I'll articulate how it works, then show you how to work with it practically. I'll then look at a related topic, inheritance. I'll follow that with a look at shorthand properties and some common misunderstandings surrounding them.

Together, these topics are all about applying the styles you want to the elements you want. There are a lot of "gotchas" here that often trip up developers. A good understanding of these topics will give you better control over making your CSS do what you want it to do. With any luck, you'll also better appreciate and even enjoy working with CSS.

1.1 The cascade

Fundamentally, CSS is about declaring rules: Under various conditions, you want certain things to happen. If this class is added to that element, apply these styles. If element X is a child of element Y, apply those styles. The browser then takes these rules, figures out which ones apply where, and uses them to render the page.

When you look at small examples, this process is usually straightforward. But as your stylesheet grows, or the number of pages you apply it to increases, your code can become complex surprisingly quickly. There are often several ways to accomplish the same thing in CSS. Depending on which solution you use, you may get wildly different results when the structure of the HTML changes, or when the styles are applied to different pages. A key part of CSS development comes down to writing rules in such a way that they're predictable.

The first step toward this is understanding how exactly the browser makes sense of your rules. Each rule may be straightforward on its own, but what happens when two rules provide conflicting information about how to style an element? You may find one of your rules doesn't do what you expect because another rule conflicts with it. Predicting how rules behave requires an understanding of the cascade.

To illustrate, you'll build a basic page header like one you might see at the top of a web page (figure 1.1). It has the website title atop a series of teal navigational links. The last link is colored orange to make it stand out as a sort of featured link.

Note

to print book readers Many graphics in this book are best viewed in color. I have taken steps to ensure they are understandable when printed in black and white, but at times you may find it helpful to refer to the eBook for color graphics. To get your free eBook in PDF, ePub, and Kindle formats, go to https://www.manning.com/books/css-in-depth to register your print book.

As you build this page header, you'll probably be familiar with most of the CSS involved. This will allow us to focus on aspects of CSS you might take for granted or only partially understand.

Figure 1.1 Page heading and navigation links you will build in this chapter

To begin, create an HTML document and a stylesheet named styles.css. Add the code in listing 1.1 to the HTML.

Note

A repository containing all code listings in this book is available for download at https://github.com/CSSInDepth/css-in-depth-2. The repository has all CSS embedded with the corresponding HTML in a series of HTML files.

Listing 1.1 Markup for the page header

```
<!doctype html>
<html lang=”en-US”>
<head>
  <link href="styles.css" rel="stylesheet" type="text/css" />
</head>
<body>
  <header class="page-header">
    <h1 id="page-title" class="title">Wombat Coffee
Roasters</h1>    #A
    <nav>
      <ul id="main-nav" class="nav">    #B
        <li><a href="/">Home</a></li>
        <li><a href="/coffees">Coffees</a></li>
        <li><a href="/brewers">Brewers</a></li>
        <li><a href="/specials" class="featured">Specials</a>
</li>    #C
      </ul>
    </nav>
  </header>
</body>
</html>
```

When two or more rules target the same element on your page, the rules may provide conflicting declarations. The next listing shows how this is possible. It shows three rulesets, each specifying a different font style for the page title. The title can’t have three different fonts at one time. Which one will it be? Add this to your CSS file to see.

Listing 1.2 Conflicting declarations

```
h1 {         #A
  font-family: serif;
}


#page-title {   #B
  font-family: sans-serif;
```

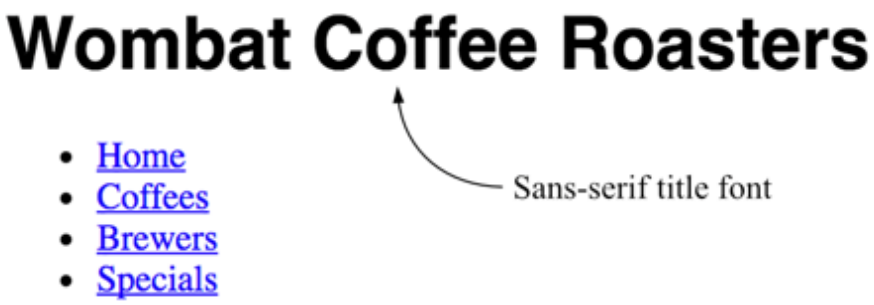
```
}


.title {    #C
  font-family: monospace;
}
```

Rulesets with conflicting declarations can appear one after the other, or they can be scattered throughout your stylesheet. Either way, given your HTML, these all target the same element.

All three rulesets attempt to set a different font family to this heading. Which one will win? To determine the answer, the browser follows a set of rules, so the result is predictable. In this case, the rules dictate that the second declaration wins, due to the id in its selector; the title will have a sans-serif font (figure 1.2).

The *cascade* is the name for this set of rules. It determines how conflicts are resolved, and it's a fundamental part of how the language works. Although most experienced developers have a general sense of the cascade, parts of it are sometimes misunderstood.

Figure 1.2 The ID selector wins over the other rulesets, producing a sans-serif font for the title.



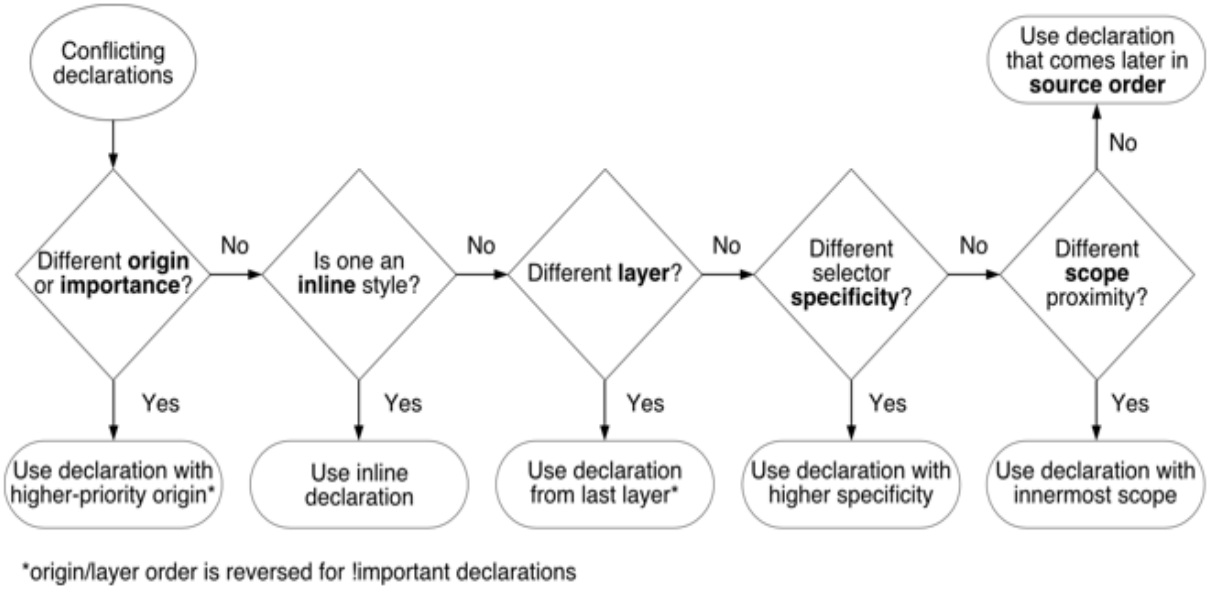
Let's unpack the cascade. When declarations conflict, the cascade considers six criteria in the following order to resolve the difference:

1. *Stylesheet origin*—Where the styles come from. Your styles are applied in conjunction with the browser's default styles.
2. *Inline styles*—Whether a declaration is applied to an element via the HTML `style` attribute or via a CSS selector.

3. *Layer*—Styles can be defined in layers, each with a different priority.
4. *Selector specificity*—Which selectors take precedence over which.
5. *Scope proximity*—Whether the styles are scoped to a portion of the DOM.
6. *Source order*—Order in which styles are declared in the stylesheet.

Some of these criteria are impacted by the use of the `!important` annotation, which we will look at later in the chapter. Figure 1.3 shows how the rules are applied at a high level.

Figure 1.3 High-level flowchart of the cascade showing declaration precedence



These rules allow browsers to behave predictably when resolving any ambiguity in the CSS.

Layers and *scope* are new additions to CSS that provide more explicit control over the cascade. We will look at these in greater depth in chapters 8 and 9.

Without any layers or scoped styles in your stylesheets, the remaining four aspects of the cascade continue to behave as they have for years, and that is where I'll focus first. I'll step you through them one at a time: origin, inline styles, specificity, and source order.

A quick review of terminology

Depending on where you learned CSS, you may or may not be familiar with all the names of the various parts of CSS syntax. I won't belabor the point, but because I'll be using these terms throughout the book, it's best to be clear about what they mean.

Following is a line of CSS. This is called a *declaration*. The following declaration comprises a *property* (`color`) and a *value* (`black`):

```
color: black;
```

Properties aren't to be confused with *attributes*, which are part of the HTML syntax. For example, in the element `<a href="/">`, `href` is an attribute of the `a` tag.

A group of declarations inside curly braces is called a *declaration block*. A declaration block is preceded by a selector (in this case, `body`):

```
body {
  color: black;
  font-family: Helvetica;
}
```

Together, the selector and declaration block are called a *ruleset*. A ruleset is also called a *rule*—although, it's my observation that rule is rarely used so precisely and is usually used in the plural to refer to a broader set of styles.

Finally, *at-rules* are language constructs beginning with an "at" symbol, such as `@import` rules or `@media` queries.

1.1.1 Stylesheet origin

The stylesheets you add to your web page aren't the only ones the browser applies. There are three different types, or origins, of stylesheets. The styles you add to your page are called *author* styles. There are also *user* styles, which are customizations added by the end user, and *user-agent* styles, which are the browser's default styles. User-agent styles have lower priority,

so your styles override them. User styles are rare, but if present, they have a priority between user-agent styles and author styles.

Note

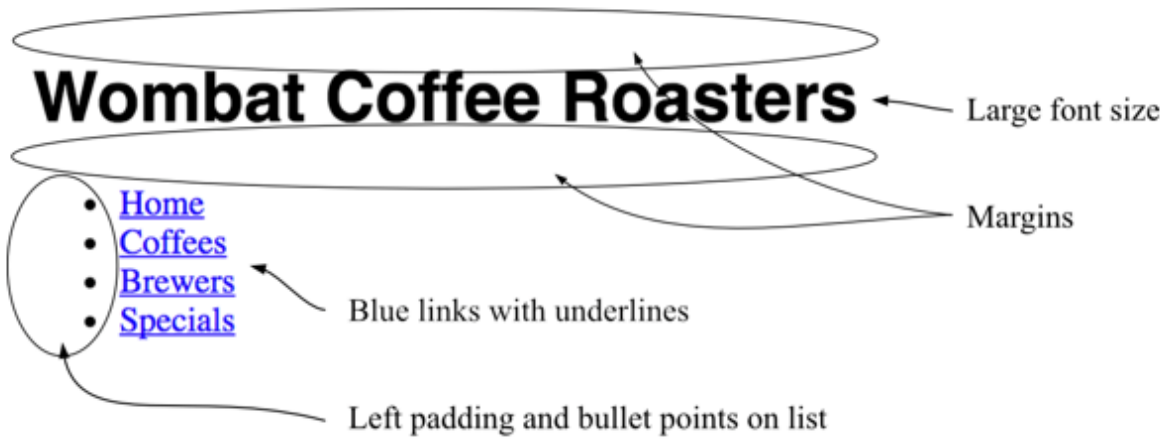
Only certain browsers allow users to define a user stylesheet. User styles are rarely used and beyond your control, so you generally don't need to worry about them when styling a page.

User-agent styles vary slightly from browser to browser, but generally they do the same things: headings (`<h1>` through `<h6>`) and paragraphs (`<p>`) are given a top and bottom margin, lists (`<ol>` and `<ul>`) are given a left padding, and link colors and default font sizes are set.

User-agent styles

Let's look again at the example page (figure 1.4). The title is sans-serif because of the styles you added. A number of other things are determined by the user-agent styles: the list has a left padding and a `list-style-type` of `disc` to produce the bullets. Links are blue and underlined. The heading and the list have top and bottom margins and the heading has a large font size.

Figure 1.4 User-agent styles set defaults for your web page header.



After user-agent styles are considered, the browser applies your styles—the author styles. This allows declarations you specify to override those set by

the user-agent stylesheet. If you link to several stylesheets in your HTML, they all have the same origin: the author.

The user-agent styles set things you typically want, so they don't do anything entirely unexpected. When you don't like what they do to a certain property, set your own value in your stylesheet. Let's do that now. You can override some of the user-agent styles that aren't what you want so your page will look like figure 1.5.

Figure 1.5 Author styles override user-agent styles because they have higher priority.

In the following listing, I've removed the conflicting font-family declarations from the earlier example and added new ones to set colors and override the user-agent margins and the list padding and bullets. Edit your stylesheet to match these changes.

Listing 1.3 Overriding user-agent styles

```
h1 {
  color: #2f4f4f;
  margin-bottom: 10px;   #A
}

#main-nav {
  margin-top: 10px;        #A
  list-style: none;        #B
  padding-left: 0;    #B
}

#main-nav li {
  display: inline-block;   #C
}

#main-nav a {
```

```
  color: white;                #D
  background-color: #13a4a4; #D
  padding: 5px;                #D
  border-radius: 2px;         #D
  text-decoration: none;           #D
}
```

If you've worked with CSS for long, you're probably used to overriding user-agent styles. When you do, you're leveraging the origin part of the cascade. Your styles will always override the user-agent styles because the origins are different.

Note

You may notice I used ID selectors in this code. There are reasons to avoid doing this, which I'll cover in a bit.

Important declarations

There's an exception to the style origin rules: declarations that are marked as important. A declaration can be marked important by adding `!important` to the end of the declaration, before the semicolon:

```
color: red !important;
```

Declarations marked `!important` are treated as a higher-priority origin, so they will always override normal (non-important) styles from any origin. All taken together, the overall order of preference, in decreasing order, is this:

1. Important user-agent
2. Important user
3. Important author
4. Normal author
5. Normal user
6. Normal user-agent

Declarations from origins earlier in this list will always win over declarations from later origins. Also notice that the order of important

origins is inverted from that of normal origins—we'll see this concept again when we get to cascade layers in chapter 8.

Note

Transitions and keyframe animations (covered in later chapters) also create two additional origins—These effects create "virtual" rules for dynamically changing values that receive special consideration in the cascade so they behave predictably.

The cascade independently resolves conflicts for every property of every element on the page. For instance, if you set a bold font on a paragraph, the top and bottom margin from the user-agent stylesheet can still apply. The `!important` annotation is an interesting quirk of CSS, which we'll come back to again shortly.

1.1.2 Inline styles

If conflicting declarations can't be resolved based on their origin, the browser next considers whether they are added to an element via inline styles. When you use an HTML style attribute to apply styles, the declarations are applied only to that element. These will override any declarations applied from your stylesheet or a `<style>` tag. Inline styles have no selector because they are applied directly to the element they target.

In your page, let's make the featured Specials link in the navigation menu to be orange, as shown in figure 1.6. I'll evaluate several ways you can accomplish this, beginning with inline styles in listing 1.4.

Figure 1.6 Applying inline styles overrides the styles applied using selectors.

To see this in your browser, edit your page to match the code given here. (You'll undo this change in a moment.)

Listing 1.4 Inline styles overriding declarations applied elsewhere

```
<li>
  <a href="/specials" class="featured"
    style="background-color: orange;">   #A
    Specials
  </a>
</li>
```

To override inline declarations in your stylesheet, you'll need to add an `!important` to the declaration, shifting it into a higher-priority origin. If the inline styles are marked important, then nothing can override them. It's preferable to do this from within the stylesheet. Undo this change, and we'll look at better approaches.

1.1.3 Selector specificity

Specificity is a feature of CSS that is often not readily apparent when learning the language, but understanding it is essential. You can go a long way without an understanding of stylesheet origin because most of the styles will be ones you add, all belonging to the author origin. But if you don't understand specificity, it will almost certainly cause problems for you.

If the previous aspects of the cascade cannot resolve a conflict between declarations, the browser seeks to resolve them by looking at the specificity of their selectors. For instance, a selector with two class names has a higher specificity than a selector with only one. If one declaration sets a background to orange, but another with higher specificity sets it to teal, the browser applies the teal color.

To illustrate, let's see what happens if you try to turn the featured link orange with a simple class selector. Update the final part of your stylesheet so it matches the code given here.

Listing 1.5 Selectors with different specificities

```
#main-nav a {           #A
  color: white;
  background-color: #13a4a4;        #B
  padding: 5px;
  border-radius: 2px;
  text-decoration: none;
}

.featured {         #C
  background-color: orange;
}
```

It doesn’t work! All the links remain teal. Why? The first selector here is more specific than the second. It’s made up of an ID and a tag name, whereas the second is made up of a class name. There’s more to this than merely seeing which selector is longer, however.

Different types of selectors also have different specificities. An ID selector has a higher specificity than a class selector, for example. In fact, a single ID has a higher specificity than a selector with any number of classes. Similarly, a class selector has a higher specificity than a tag selector (also called a type selector).

The exact rules of specificity are:

- If a selector has more IDs, it wins (that is, it’s more specific).
- If that results in a tie, the selector with the most classes wins.
- If that results in a tie, the selector with the most tag names wins.

Consider the selectors shown in the following listing (but don’t add them to your page). These are written in order of increasing specificity.

Listing 1.6 Selectors with increasing specificities

```
html body header h1 {  #1
  color: blue;
}

body header.page-header h1 {   #2
  color: orange;
}

.page-header .title {    #3
```

```
  color: green;
}

#page-title {        #4
  color: red;
}
```

The most specific selector here is #4, with one ID, so its color declaration of red is applied to the title. The next specific is #3, with two class names. This would be applied if the ID selector #4 were absent. Selector #3 has a higher specificity than selector #2, despite its length: two classes are more specific than one class. Finally, #1 is the least specific, with four element types (that is, tag names) but no IDs or classes.

Note

Pseudo-class selectors (for example, `:hover`) and attribute selectors (for example, `[type="input"]`) each has the same specificity as a class selector. The universal selector (`*`) and combinators (`>`, `+`, `~`) have no effect on specificity.

If you add a declaration to your CSS and it seems to have no effect, often it's because a more specific rule is overriding it. Many times developers write selectors using IDs, without realizing this creates a higher specificity, one that is hard to override later. If you need to override a style applied using an ID, you have to use another ID.

It's a simple concept, but if you don't understand specificity, you can drive yourself mad trying to figure out why one rule works and another doesn't.

A notation for specificity

A common way to indicate specificity is in a number form, often with commas between each number. For example, "1,2,2" indicates a specificity of one ID, two classes, and two tags. IDs, having the highest priority, are listed first, followed by classes, then tags.

The selector `#page-header #page-title` has two IDs, no classes, and no tags. We can say this has a specificity of 2,0,0. The selector `ul li`, with two

tags but no IDs or classes, has a specificity of 0,0,2. Table 1.1 shows the selectors from listing 1.6.

Table 1.1 Various selectors and their corresponding specificities

Selector	IDs	Classes	Tags	Notation
`html body header h1`	0	0	4	0,0,4
`body header.page-header h1`	0	1	3	0,1,3
`.page-header .title`	0	2	0	0,2,0
`#page-title`	1	0	0	1,0,0

It now becomes a matter of comparing the numbers to determine which selector is more specific. A specificity of 1,0,0 takes precedence over a specificity of 0,2,2 and even over 0,10,0 (although I don't recommend ever writing selectors as long as one with 10 classes), because the first number (IDs) is of the higher priority.

Occasionally, people use a four-number notation with a 0 or 1 in the most significant digit to represent whether a declaration is applied via inline styles, because inline styles were originally defined as a subset of specificity in earlier versions of the CSS specification. In this case, an inline style has a notation of 1,0,0,0. This would override styles applied via selectors, which could be indicated as having specificities of 0,1,2,0 (one ID and two classes) or something similar.

Specificity considerations

When you tried to apply the orange background using the `.featured` selector, it didn't work. The selector `#main-nav a` has an ID that overrides the class selector (specificities 1,0,1 and 0,1,0). To correct this, you have some options to consider. Let's look at several possible fixes.

The quickest fix is to add an `!important` to the declaration you want to favor. Change the declaration to match that given here.

Listing 1.7 Possible fix one

```
#main-nav a {
  color: white;
  background-color: #13a4a4;
  padding: 5px;
  border-radius: 2px;
  text-decoration: none;
}

.featured {
  background-color: orange !important;   #A
}
```

This works because the `!important` annotation raises the declaration to a higher-priority origin. Sure, it's easy, but it's also a naive fix. It may do the trick now, but it can cause you problems down the road. If you start adding `!important` to multiple declarations, what happens when you need to override something already set to important? When you give several declarations an `!important`, then the origins match and the regular specificity rules apply. This ultimately will leave you back where you started.

Let's find a better way. Instead of trying to get around the rules of selector specificity, let's try to make them work for us. What if you raised the specificity of your selector? Update the rulesets in your CSS to match this listing.

Listing 1.8 Possible fix two

```
#main-nav a {          #A
  color: white;
  background-color: #13a4a4;
```

```
  padding: 5px;
  border-radius: 2px;
  text-decoration: none;
}

#main-nav .featured {         #B
  background-color: orange; #C
}
```

This fix also works. Now, your selector has one ID and one class, giving it a specificity of 1,1,0, which is higher than `#main-nav a` (a specificity of 1,0,1), so the background color orange is applied to the element.

You can still make this better, though. Instead of raising the specificity of the second selector, let's see if you can lower the specificity of the first. The element has a class as well: `<ul id="main-nav" class="nav">`, so you can change your CSS to target the element by its class name rather than its ID. Change `#main-nav` to `.nav` in your selectors as shown here.

Listing 1.9 Possible fix three

```
.nav {               #A
  margin-top: 10px;
  list-style: none;
  padding-left: 0;
}

.nav li {             #A
  display: inline-block;
}

.nav a {        #B
  color: white;
  background-color: #13a4a4;
  padding: 5px;
  border-radius: 2px;
  text-decoration: none;
}

.nav .featured { #C
  background-color: orange;
}
```

You've brought the specificity of the selectors down. The orange background is high enough to override the teal.

As you can see from these examples, specificity tends to become a sort of arms race. This is particularly the case with large projects. It is generally best to keep specificity low when you can, so when you need to override something, your options are open.

Tip

You can also reduce selector specificity by using the `:where()` pseudo-class. It always has zero specificity. For example, `:where(.nav)` is equivalent to the selector `.nav`, but it has a specificity of 0,0,0, and `.nav :where(a)` is equivalent to `.nav a`, but its specificity is 0,1,0. See Appendix A for more on `:where()`.

1.1.4 Source order

The final step to resolving the cascade is source order. If all other criteria are the same, then the declaration that appears later in the stylesheet—or appears in a stylesheet included later on the page—takes precedence.

This means you can manipulate the source order to style your featured link. If you make the two conflicting selectors equal in specificity, then whichever appears last wins. Let's consider the fourth option shown in the following listing.

Listing 1.10 Possible fix four

```
.nav a {
  color: white;
  background-color: #13a4a4;
  padding: 5px;
  border-radius: 2px;
  text-decoration: none;
}

a.featured {                    #A
  background-color: orange;
}
```

In this solution, the specificities are equal. Source order determines which declaration is applied to your link, resulting in an orange featured button.

This addresses your problem but, potentially, also introduces a new one: although a featured button inside the nav looks correct, what happens if you want to use the `featured` class on another link elsewhere on the page, outside of your nav? You'll get an odd blend of styles: the orange background, but not the text color, padding, or border radius of the navigational links (figure 1.7).

Figure 1.7 The `featured` class used outside the nav produces odd results.

Listing 1.11 shows the markup that creates this behavior. There's now an element targeted only by the second selector, but not the first, which produces an undesirable result. You'll have to decide whether you want this orange button style to work outside of the nav, and if you do, you'll need to make sure all the desired styles apply to it as well.

Listing 1.11 Featured link outside of nav

```
<header class="page-header">
  <h1 id="page-title" class="title">Wombat Coffee Roasters</h1>
  <nav>
    <ul id="main-nav" class="nav">
      <li><a href="/">Home</a></li>
      <li><a href="/coffees">Coffees</a></li>
      <li><a href="/brewers">Brewers</a></li>
      <li><a href="/specials" class="featured">Specials</a></li>
    </ul>
  </nav>
</header>
<main>
```

```
  <p>
    Be sure to check out
    <a href="/specials" class="featured">our specials</a>.   #A
  </p>
</main>
```

With no other information about your needs on this site, I'd be inclined to stick with fix number three (listing 1.9). Ideally on your websites, you'll be able to make some educated guesses about your needs elsewhere. Perhaps you know that you are likely to need a featured link in other places. In that case, perhaps fix four (listing 1.10) would be what you want, with the addition of styles to support the `featured` class elsewhere on the page.

Very often in CSS, as I said earlier, the best answer is "it depends." There are many paths to the same end result. It's worth considering several options and thinking about the ramifications of each. When facing a styling problem, I often tackle it in two phases: First figure out what declarations will get it looking right. Second, think through the possible ways to structure the selectors and choose the one that best fits your needs.

Link styles and source order

When you began studying CSS, you may have learned that your selectors for styling links should go in a certain order. That's because source order affects the cascade. This listing shows styles for links on a page in the "correct" order. Add this to your stylesheet.

Listing 1.12 Link styles

```
a:link {
  background-color: blue;
  color: white;
  text-decoration: none;
  padding: 2px;
}
a:visited {
  background-color: purple;
}

a:hover {
  background-color: transparent;
```

```
  color: blue;
  text-decoration: underline;
}

a:active {
  color: red;
}
```

The cascade is the reason this order matters: given the same specificity, later styles override earlier styles. If two or more of these states are true of one element at the same time, the last one can override the others. If the user hovers over a visited link, the hover styles take precedence. If the user activates the link (that is, clicks it) while hovering over it, the active styles take precedence.

A helpful mnemonic to remember this order is LoVe/HAte—link, visited, hover, active. Note that if you change one of the selectors to have a different specificity than the others, this will break down and you may get unexpected results.

Cascaded values

The browser follows these steps to resolve every property for every element on the page. A declaration that "wins" the cascade is called a *cascaded value*. There is at most one cascaded value per property per element. A particular paragraph (`<p>`) on the page can have a top margin and a bottom margin, but it can't have two different top margins or two different bottom margins. If the CSS specifies different values for one property, the cascade will choose only one when rendering the element. This is the cascaded value.

Cascaded value

A value for a particular property applied to an element as a result of the cascade.

If a property is never specified for an element, it has no cascaded value for that property. The same paragraph, for instance, may not have a border or padding specified.

The cascade is getting easier to work with

Historically, there are two common rules of thumb for working with the cascade. First, don't use IDs in your selectors. Second, don't use `!important`. These both make it difficult to override styles in the future should the need arise.

These two rules can be good advice, but things are changing. There are many situations where using them can be acceptable—using layers and scope, specifically. However, it's important to never use them as a knee-jerk reaction to win a specificity battle. In Part III, we will examine modern tools for controlling the cascade and I will show you examples where it is appropriate to break these rules. But now that you're clear on how the cascade behaves, we can press on.

An important note about importance

If you're creating a JavaScript module for distribution (such as an NPM package), I strongly urge you not to apply styles inline via JavaScript if it can be avoided. If you do, you're forcing developers using your package to either accept your styles exactly or use `!important` for every property they want to change.

Instead, include a stylesheet in your package. If your component needs to make style changes dynamically, it's almost always preferable to use JavaScript to add and remove classes to the elements. Then users can use your stylesheet, and they have the option to edit it however they like without battling specificity.

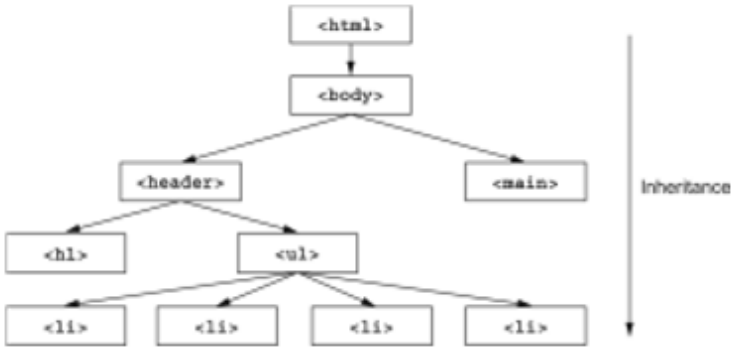
1.2 Inheritance

There's one last way that an element can receive styles—inheritance. The cascade is frequently conflated with the concept of inheritance. Although the two topics are related, you should understand each individually.

If an element has no cascaded value for a given property, it may inherit one from an ancestor element. It's common to apply a `font-family` to the

<body> element. All the ancestor elements within will then inherit this font; you don't have to apply it explicitly to each element on the page. Figure 1.8 shows how inheritance flows down the DOM tree.

Figure 1.8 Inherited properties are passed down the DOM tree from parent nodes to their descendants.



Not all properties are inherited, however. By default, only certain ones are. In general, these are the properties you'll want to be inherited. They are primarily properties pertaining to text: `color`, `font`, `font-family`, `font-size`, `font-weight`, `font-variant`, `font-style`, `line-height`, `letter-spacing`, `text-align`, `text-indent`, `text-transform`, `white-space`, and `word-spacing`.

A few others inherit as well, such as the list properties: `list-style`, `list-style-type`, `list-style-position`, and `list-style-image`. The table border properties, `border-collapse` and `border-spacing`, are also inherited; note that these control border behavior of tables, not the more commonly used properties for specifying borders for non-table elements. (We wouldn't want a passing its border down to every descendant element.) This is not quite a comprehensive list, but very nearly.

You can use inheritance in your favor on your page by applying a font to the body element, allowing its descendant elements to inherit that value (figure 1.9).

Figure 1.9 Applying a `font-family` to the body allows all descendant elements to inherit the same value.

Add this code to the top of your stylesheet to apply this principle to your page.

Listing 1.13 Applying font-family to a parent element

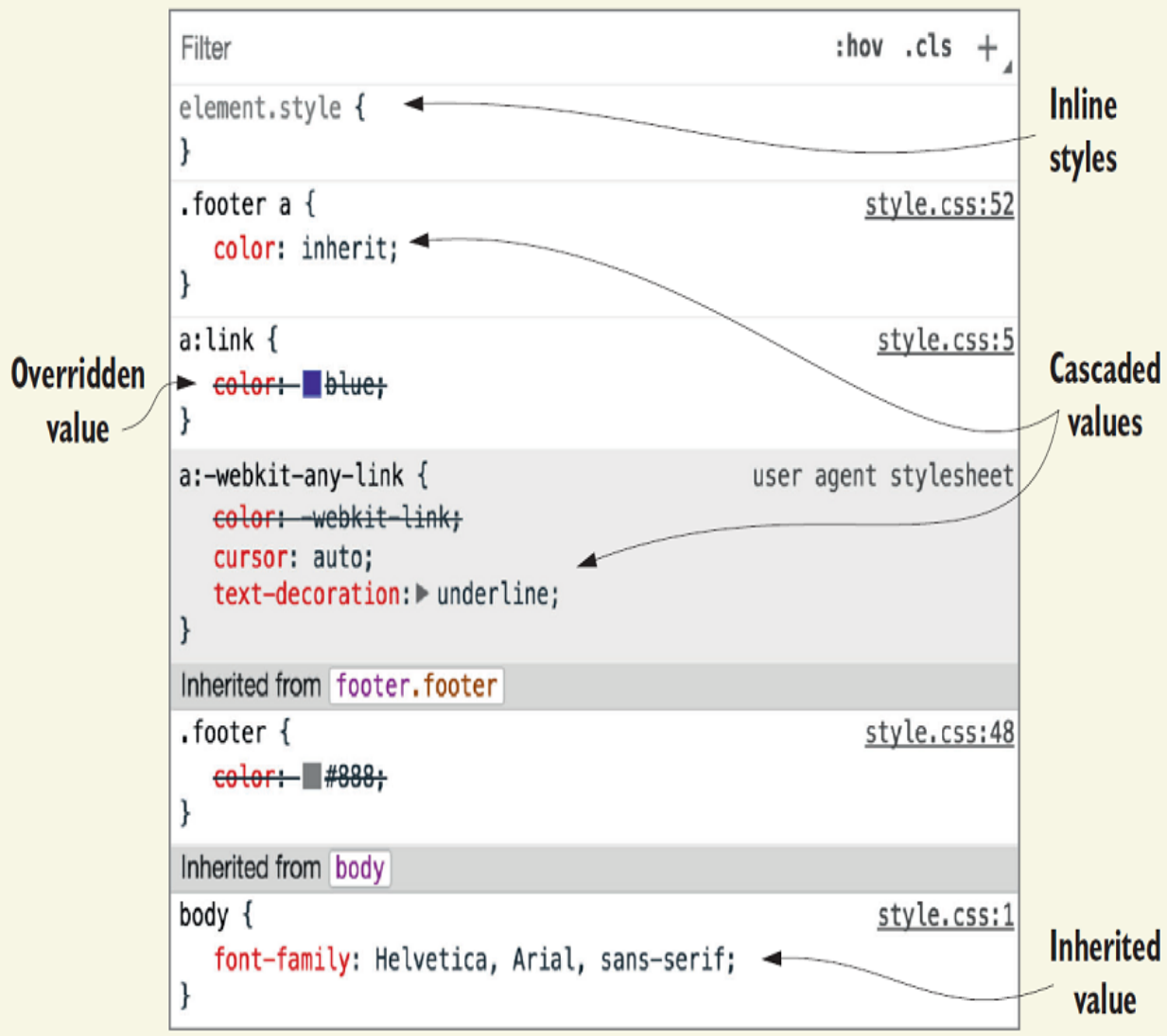
```
body {
  font-family: sans-serif;  #A
}
```

This is applied to the entire page by adding it to the body. But you can also target a specific element on the page, and it will only inherit to its descendant elements. The inheritance will pass from element to element until it's overridden by a cascaded value.

Use your DevTools!

A complicated nest of values inheriting and overriding one another can quickly become difficult to keep track of. If you're not already familiar with your browser's developer tools, get in the habit of using them.

DevTools provide visibility into exactly which rules are applying to which elements and why. The cascade and inheritance are abstract concepts; DevTools are the best way I know to get my bearings. Open them by right-clicking an element and choosing Inspect or Inspect Element from the context menu. Figure 1.x in an example of what you'll see.



The style inspector shows every selector targeting the inspected element, ordered by specificity. Below that are all inherited properties. This shows all of the cascade and inheritance for the element at a glance.

There are lots of subtle features to help you make sense of what's happening with an element's styles. Styles closer to the top override those below. Overridden styles are crossed out. The stylesheet and line number for each ruleset are shown on the right, so you can find them in your source code. This tells you exactly which element inherited which styles and where they originated. You can also type in the Filter box at the top to hide all but a certain set of declarations.

1.3 Special values

There are some special values that you can apply to any property to help manipulate the cascade: `inherit`, `initial`, `unset`, and `revert`. Let's take a look at these.

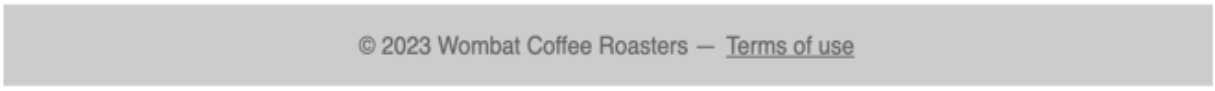
You may have noticed I provided slightly unusual styles for links back in listing 1.12: rather than a more traditional blue text, I specified white text and a blue background color. I did this intentionally to help illustrate the behavior of these keywords.

1.3.1 The inherit keyword

Sometimes, you'll want inheritance to take place when a cascaded value is preventing it. To do this, you can use the keyword `inherit`. You can override another value with this, and it will cause the element to inherit that value from its parent.

Suppose you add a light gray footer to your page. In the footer, there may be some links, but you don't want them to stand out too much because the footer isn't an important part of the page. So you'll make the links in the footer dark gray (figure 1.10).

Figure 1.10 The Terms of Use link when it inherits the gray text color



Add this markup to the end of your page. A normal page would have more content between this and the header, but this will serve the purpose.

Listing 1.14 Footer with a link

```
<footer class="footer">
  &copy; 2023 Wombat Coffee Roasters &mdash;
  <a href="/terms-of-use">Terms of use</a>
</footer>
```

Earlier, you applied some styles to all links on the page, and they will be applied to the Terms of Use link as well. To make the link in the footer gray, you'll need to override it. Add this code to your stylesheet to do that.

Listing 1.15 The inherit value

```
.footer {
  color: #666;              #A
  background-color: #ccc;
  padding: 15px 0;
  text-align: center;
  font-size: 14px;
}

.footer a {
  color: inherit;             #B
  background-color: transparent;  #C
  text-decoration: underline;
}
```

The second ruleset here overrides the link color, giving the link in the footer a cascaded value of `inherit`. Thus, it inherits the color from its parent, `<footer>`.

The benefit here is that the footer link will change along with the rest of the footer should anything alter it. (Editing the second ruleset can do this, or another style elsewhere could override it.) If, for example, the footer text on certain pages is a darker gray, then the link will change to match.

You can also use the `inherit` keyword to force inheritance of a property not normally inherited, such as border or padding, though you will likely not find many practical uses for this.

1.3.2 The initial keyword

Sometimes you'll find you have styles applied to an element that you want to undo, as you did with the background color in listing 1.15. You can do this by specifying the keyword `initial`. Every CSS property has an initial, or default, value. If you assign the value `initial` to that property, then it effectively resets to its default value. It's like a hard-reset of that value.

Instead of setting `background-color: transparent`, you can use `initial`. This is done in the following listing. Because `transparent` is the initial value for the background-color property, this accomplishes the same thing as the previous listing.

Listing 1.16 The initial value

```
.footer a {
  color: inherit;
  background-color: initial;   #A
  text-decoration: underline;
}
```

The benefit of this is you don't have to think about it as much. If you want to remove a border from an element, set `border: initial`. If you want to restore an element to its default width, set `width: initial`.

You may be in the habit of using the value `auto` to do this sort of reset. In fact, you can use `width: auto` to achieve the same result. This is because the default value of `width` is `auto`.

It's important to note, however, that `auto` isn't the default value for all properties. It's not even valid for many properties; for example, `border-width: auto` and `padding: auto` are invalid and therefore have no effect. You could take the time to dig up the initial value for these, but it's often easier to use `initial`.

Note

Declaring `display: initial` is equivalent to `display: inline`. It won't evaluate to `display: block` regardless of what type of element you apply it to. That's because `initial` resets to the initial value for the property, not the element; `inline` is the default value for the display property. In this case, consider using the `revert` keyword instead.

1.3.3 The unset keyword

The inherit and initial keywords are useful for clearing values you've set on properties that are either inherited or non-inherited. The unset keyword is a

combination of the two. When applied to an inherited property, it sets the value to `inherit`, and when applied to a non-inherited property, it sets the value to `initial`.

You can of course do this with `inherit` and `initial`, respectively, but using `unset` instead makes it a little simpler and helps you avoid using the wrong keyword by mistake. In this footer, you can use unset to fix both the font color and background color of links.

Listing 1.17 The unset value

```
.footer a {
  color: unset;  #A
  background-color: unset;  #B
 text-decoration: underline;
}
```

This sets the color to inherit, allowing it to inherit the same gray color as the rest of the footer. But the background color is set to transparent, the initial value for the background-color property.

You will notice I still had to declare `text-decoration: underline` to restore link underlines. In this case, `unset` would not work, because the initial value for text-decoration is `none`, not `underline`. Remember, this is the initial value for the CSS property itself, which is the same regardless whether it's applied to a link or any other element. To "undo" the text decoration style we've set, we need one more keyword.

1.3.4 The revert keyword

The `initial` and `unset` keywords essentially override all styles, from both author and user-agent stylesheets. Sometimes what you want is to override your previously-set author styles, but leave the user-agent styles intact. This is what the `revert` keyword does.

In our footer, if you set all three properties to revert, it would revert all the way back to a blue link with an underline (the browser's default styles). But since we like the design we have with the gray link, you can use the keyword

more selectively, applying it only to text-decoration. Update your page to match listing 1.18 to see this.

Listing 1.18 The revert value

```
.footer a {
  color: unset;
  background-color: unset;
  text-decoration: revert;  #A
}
```

These keywords may seem like overkill in this example, where the properties are probably familiar to you; you likely know the values you hope for here and could declare them explicitly. But when working with newer CSS features or something like flexbox properties that have multiple keyword values, you may not know off the top of your head what the default values are. When this occurs, these keywords come in handy. There is also the added benefit of subtly indicating to future authors of the stylesheet that your goal is to effectively undo styles applied elsewhere in the stylesheet.

Warning

These keywords are normal cascaded values. That means it is still possible to override them with other values when another selector with higher specificity targets the same element.

1.4 Shorthand properties

Shorthand properties are properties that let you set the values of several other properties at one time. For example, `font` is a shorthand property that lets you set several font properties. This declaration specifies `font-style`, `font-weight`, `font-size`, `line-height`, and `font-family`:

```
font: italic bold 18px/1.2 "Helvetica", "Arial", sans-serif;
```

Similarly,

- `background` is a shorthand property for multiple background properties: `background-color`, `background-image`, `background-size`,

`background-repeat`, `background-position`, `background-origin`, `background-chip`, and `background-attachment`.

- `border` is a shorthand for `border-width`, `border-style`, and `border-color`, which are each in turn shorthand properties as well.
- `border-width` is shorthand for the top, right, bottom, and left border widths.

Shorthand properties are useful for keeping your code succinct and clear, but a few quirks about them aren't readily apparent.

1.4.1 Beware shorthands silently overriding other styles

Most shorthand properties let you omit certain values and only specify the bits you're concerned with. It's important to know, however, that doing this still sets the omitted values; they'll be set implicitly to their initial value. This can silently override styles you specify elsewhere. If, for example, you were to use the shorthand `font` property for the page title without specifying a `font-weight`, a font weight of `normal` would still be set (figure 1.11).

Figure 1.11 Using the shorthand `font: 32px sans-serif` sets font-weight and other omitted values to their initial value.

Add the code from this listing to your stylesheet for an example of how this works.

Listing 1.19 Shorthand property specifying all associated values

```
h1 {
  font-weight: bold;
}

.title {
  font: 32px Helvetica, Arial, sans-serif;
}
```

At first glance, it may seem that `<h1 class="title">` would result in a bold heading, but it doesn't. These styles are equivalent to this code:

Listing 1.20 Expanded equivalent to the shorthand in listing 1.19

```
h1 {
  font-weight: bold;
}

.title {
  font-style: normal;             #A
  font-variant: normal;            #A
  font-weight: normal;        #A
  font-stretch: normal;            #A
  line-height: normal;        #A
  font-size: 32px;
  font-family: Helvetica, Arial, sans-serif;
}
```

This means that applying these styles to `<h1>` results in a normal font weight, not bold. It can also override other font styles that would otherwise be inherited from an ancestor element. Of all the shorthand properties, `font` is the most egregious for causing problems, because it sets such a wide array of properties. For this reason, I avoid using it except to set general styles on the `<body>` element. You can still encounter this problem with other shorthand properties, so be aware of this possibility.

1.4.2 Remember the order of shorthand values

Shorthand properties try to be lenient when it comes to the order of the values you specify. You can set `border: 1px solid black` or `border: black 1px solid` and either will work. That's because it's clear to the browser which value specifies the width, which specifies the color, and which specifies the border style.

But there are many properties where the values can be more ambiguous. In these cases, the order of the values is significant. It's important to understand this order for the shorthand properties you use.

Top, right, bottom, left

Shorthand property order particularly trips up developers when it comes to properties like `margin` and `padding`, or some of the border properties that specify values for each of the four sides of an element. For these properties, the values are in clockwise order, beginning at the top.

Remembering this order can keep you out of trouble. In fact, the word *TRouBLe* is a mnemonic you can use to remember the order: top, right, bottom, left.

You can use this mnemonic to set padding on the four sides of an element. The links shown in figure 1.12 have a top padding of 10 px, right padding of 15 px, bottom padding of 0, and left padding of 5 px. This looks uneven, but it illustrates the principle.

Figure 1.12 Declaring `padding: 10px 15px 0 5px` applies different paddings to each side

This listing shows the CSS for these links.

Listing 1.21 Specifying padding on each side of an element

```
.nav a {
  color: white;
  background-color: #13a4a4;
  padding: 10px 15px 0 5px;    #A
  border-radius: 2px;
  text-decoration: none;
}
```

Properties whose values follow this pattern also support truncated notations. If the declaration ends before one of the four sides is given a value, that side takes its value from the opposite side. Specify three values, and the left and right side will both use the second one. Specify two values, and the top and bottom will use the first one. If you specify only one value, it will apply to all four sides. Thus, the following declarations are all equivalent:

```
padding: 1em 2em;
padding: 1em 2em 1em;
padding: 1em 2em 1em 2em;
```

These are equivalent to one another as well:

```
margin: 1em;
margin: 1em 1em;
margin: 1em 1em 1em;
margin: 1em 1em 1em 1em;
```

For many developers, the most problematic of these is when three values are given. Remember, this specifies the top, right, and bottom. Because no left value is given, it will take the same value as the right; the second value will be applied to both the left and right sides. Thus, `padding: 10px 15px 0` applies 15 px padding to both the left and right sides, whereas the top padding is 10 px and the bottom padding is 0.

Most often, however, you'll need two values. On smaller elements in particular, it's often better to have more padding on the sides than on the top and bottom. This approach looks good on buttons or, in your page, navigational links (figure 1.13).

Figure 1.13 Many elements look better with more horizontal padding.

Update your stylesheet to match this listing. It uses the property shorthand to apply the vertical padding first, then the horizontal.

Listing 1.22 Specifying two padding values

```
.nav a {
  color: white;
  background-color: #13a4a4;
  padding: 5px 15px;   #A
  border-radius: 2px;
```

```
  text-decoration: none;
}
```

Because so many common properties follow this pattern, it's worth committing this order to memory.

Horizontal, vertical

The TRouBLe mnemonic only applies to shorthand properties that apply individually to all four sides of the box. Other properties only support up to two values. These include properties like `background-position`, `box-shadow`, and `text-shadow` (which aren't shorthand properties, strictly speaking). Compared to the four-value properties like `padding`, the order of these values is reversed. Whereas `padding: 1em 2em` specifies the vertical top/bottom values first, followed by the horizontal right/left values, `background-position: 25% 75%` specifies the horizontal value first, followed by the vertical value.

Although it seems counter-intuitive that these are opposite, the reason for this is straightforward: the two values represent a Cartesian grid. Cartesian grid measurements are typically given in the order x, y (horizontal and then vertical). If, for example, you wanted to apply a shadow like the one shown in figure 1.14, you'd specify the x (horizontal) value first.

Figure 1.14 Box shadow positioned at `10px 2px`

The styles for this element are given here.

Listing 1.23 Box-shadow specifies x value then y value

```
.nav .featured {
  background-color: orange;
```

```
  box-shadow: 10px 2px #6f9090;     #A
}
```

The first (larger) value applies to the horizontal offset, whereas the second (smaller) value applies to the vertical.

If you're working with a property that specifies two measurements from a corner, think "Cartesian grid." If you're working with one that specifies measurements for each side all the way around an element, think "clock."

1.5 Progressive enhancement

CSS is constantly evolving, and an important part of working in an evolving language is keeping tabs on which features are still relatively new to the language. In particular, it is good to know when a feature is only supported by a subset of common browsers.

The language is intentionally designed to be both forwards- and backwards-compatible, and it provides you with the tools to keep your code working in both old and new browsers at the same time. With a little forethought, you can use cutting-edge features in your CSS even when you know they won't work for all of your users.

To do this, you can add the CSS needed to provide an acceptable (but less full-featured) experience to older browsers, then you can layer on CSS using newer features knowing those particular features will only work for users in the newest browsers. The benefit of this is that it means your code is future-compatible, and as more of your users receive browser upgrades these new features will begin working for them as well. This approach is called *progressive enhancement*.

Tip

To see which browser versions support a given feature, check https://caniuse.com or the relevant documentation page on MDN at https://developer.mozilla.org/en-US/docs/Web.

1.5.1 Leveraging the cascade for progressive enhancement

The simplest way to use progressive enhancement is built into the cascade itself. If a browser encounters a declaration it does not understand, it simply ignores it. Consider the following code:

```
aside {
  background-color: #333333;  #A
  background-color: #333333aa;  #B
}
```

Because the second declaration appears after the first, the cascade uses it to determine the cascaded value of background-color for these elements.

The second declaration uses a relatively new 8-digit hex color code format (the 7th and 8th digits specify an alpha value, indicating partial transparency). This syntax is supported in most browsers, but not some older browsers including Internet Explorer. So if a user happens to be using IE, their browser simply ignores that rule, and the first declaration remains the cascaded value. That user doesn’t get the full-featured experience with a bit of transparency, but they still see a perfectly usable result.The page doesn’t “break” or throw an error for them; the browser simply continues parsing the CSS and discards the unrecognized declaration.

This may seem odd from a debugging perspective, because the language never throws errors. But it’s an essential part of how the language works, and this is intentional for this very purpose of progressive enhancement.

1.5.2 Progressively enhancing selectors

This approach is not limited to new properties or value syntaxes. It can also be applied with new selector syntaxes. Browsers that understand the selector will use it, and those that don’t will ignore the entire ruleset.

However, there is an important nuance to be aware of: when a ruleset has multiple selectors, the browser will ignore the entire ruleset if any of the selectors are unsupported or invalid. Consider what that means for the following CSS:

```
input.invalid,
input:user-invalid {
```

```
  border: 1px solid red;
}
```

At the time of writing, the `:user-invalid` pseudo-class is a new addition to CSS, and is only supported in Firefox (see Appendix A for more details on this selector). Chrome may understand `input.invalid`, but because it doesn't know what `:user-invalid` means, it will never apply these styles, even when the first selector matches.

When you want to use a new selector like this, the best approach is to separate the two selectors into their own rulesets:

```
input.invalid {
  border: 1px solid red;
}
input:user-invalid {
  border: 1px solid red;
}
```

This requires unfortunate duplication, but it is the best option. This is important to keep in mind when using new pseudo-classes, pseudo-elements, or attribute selectors (See Appendix A).

In simple cases like the example above, this only requires a few lines of duplicated code, but occasionally this might require repeating a large block of CSS. Some developers who prefer to avoid the duplication will opt not to use the new feature until browsers reach a comfortable level of support for the selector in question.

1.5.3 Feature queries using @supports()

Relying on the techniques above is sufficient when the impact of using a new feature is small, and only affects one or two CSS rules, but occasionally you will need to specify multiple different declarations for browsers that support a feature compared with those that don't. In this case, you can use a *feature query* to provide a larger set of styles depending on whether or not the browser supports a given feature.

A feature query looks like this:

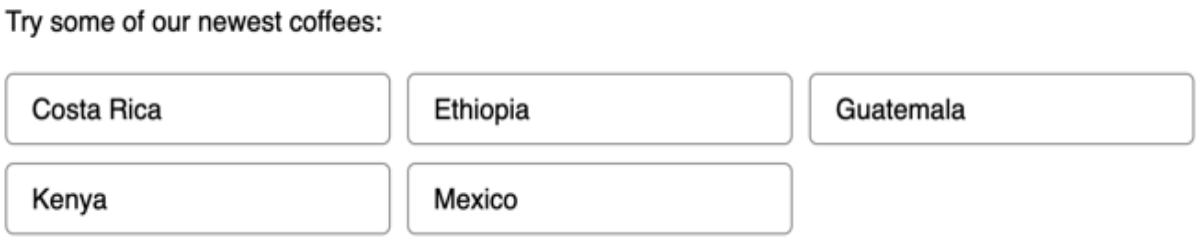
```
@supports (display: grid) { … }
```

The `@supports` rule is followed by a declaration in parentheses. If the browser understands the declaration (in this case, it supports grid), it applies any rulesets that appear between the braces. If it doesn't understand this, it will ignore them.

This means you can provide one set of styles using older layout technologies like floats. These will not necessarily be ideal styles (you'll have to make some compromises), but it will get the job done. Then, using a feature query, apply the full-featured layout using a grid.

Grid is now broadly supported in all modern browsers, but this was not always the case. I can use it to illustrate the behavior of `@supports`. Let's add a small grid of links to our page and lay them out in a grid, as shown in figure 1.15.

Figure 1.15 A grid of links you will define inside a `@supports` block



First, add the HTML shown in listing 1.24 to the `<main>` between your page header and footer.

Listing 1.24 Adding a series of links to the page.

```
<p>Try some of our newest coffees:</p>
<div class="coffees">
  <a href="/coffees/costa-rica">Costa Rica</a>
  <a href="/coffees/ethiopia">Ethiopia</a>
  <a href="/coffees/guatamala">Guatemala</a>
  <a href="/coffees/kenya">Kenya</a>
  <a href="/coffees/mexico">Mexico</a>
</div>
```

Then you can add some styles to lay these links out in a grid. First, you will provide a fallback behavior for old browsers, followed by a feature query that gives full-featured grid layout. Add these styles to your stylesheet:

Listing 1.25 Using a feature query for progressive enhancement

```
.coffees {
  margin: 20px 0;
}

.coffees > a {
  display: inline-block;   #A
  min-width: 300px;        #A
  padding: 10px 15px;
  margin-right: 10px;
  margin-bottom: 10px;
  color: black;
  background-color: transparent;
  border: 1px solid gray;
  border-radius: 5px;
}

@supports (display: grid) {   #B
  .coffees {
    display: grid;                          #C
    grid-template-columns: 1fr 1fr 1fr;  #C
    gap: 10px;                              #C
  }

  .coffees > a {
    margin: unset;      #D
    min-width: unset;   #D
  }
}
```

The fallback and other basic styles such as colors are outside of the feature query block, so they'll apply in all browsers. If you open the page in a browser that doesn't support grid, you'll see the fallback layout, which is similar to the grid layout. All styles relating to the grid-based layout are within the feature query block, so they'll only apply if the browser supports grid.

We will take a closer look at grid layout in Chapter 5, so don't worry too much about these specific declarations if they're unfamiliar.

You can imagine how the styles might apply if the `@supports` block is ignored, or you can even momentarily comment it out in your stylesheet to test the fallback behavior in a modern browser, and adjust it as necessary.

The `@supports` rule can be used to query for all sorts of CSS features. Use `@supports (mix-blend-mode: overlay)` to query for blend mode support (see chapter 11) or `@supports (color: color-mix(in oklab, red, white))` to query for color-mix support (see chapter 13).

Warning

Some very old browsers, including IE, don't support the `@supports` rule. They will ignore any rules within the feature query block, regardless of the actual feature support. This is usually okay, as you'll want the older browser to render the fallback layout.

Feature queries may be constructed in a few other ways as well:

- `@supports not(<declaration>)`—Only apply rules in the feature query block if the queried declaration isn't supported
- `@supports (<declaration>) or (<declaration>)`—Apply rules if either queried declaration is supported
- `@supports (<declaration>) and (<declaration>)`—Apply rules only if both queried declarations are supported
- `@supports selector(<selector>)`—Apply rules only if the given selector is understood by the browser (for example, `@supports selector(:user-invalid)`)

Using progressive enhancement, you can define the user experience for a full spectrum of users across multiple browsers, including future browsers. A browser that understands the new code will use it, and one that doesn't will not. Jen Simmons half-jokingly calls this "Quantum CSS." You can take a feature of CSS and "use it and not use it at the same time. It works and it doesn't work at the same time."

This feature of the language is known as *resilience*. CSS (and similarly, HTML) are designed to be fault-tolerant. As new features continue to roll out in CSS in the future, you can use this to your advantage.

Enabling experimental features

The W3C develops the CSS specifications in conjunction with browser development. This often means one or more browsers will begin to develop support for a feature before the specification is finalized. To discourage use of unstable CSS in production websites, these experimental features only work for developers who intentionally turn them on in their browser settings. This allows for early experimentation and feedback while a specification is still being finalized. It's important to know how to access experimental features should you want to learn those in the future.

In Chrome and Opera, this is done by enabling a flag. In Chrome, type `chrome://flags` into your address bar and press Enter. If you use Opera, go to `opera://flags` instead. Then scroll down until you find Experimental Web Platform Features (or use the browser's search feature) and click Enable.

If you prefer Firefox, you'll need to download and install either Firefox Developer Edition (https://www.mozilla.org/en-US/firefox/developer/) or Firefox Nightly (https://nightly.mozilla.org/). If you use Safari, you can install the Safari Technology Preview or the Webkit Nightly Builds edition.

1.6 Summary

- The browser follows the rules of the cascade to determine which styles are applied to which elements.
- A selector's specificity is determined by the number of ids, classes, and tag names in the selector. Declarations with higher specificity selectors will override those with lower specificity.
- When no cascaded value is present for certain properties, elements inherit a value from their parent element. These properties include those for text, lists, and table borders.
- Providing a value of `unset` undoes other styles otherwise provided for that property, including user-agent styles. Using `revert` undoes your author styles, but leaves user-agent styles intact.
- Shorthand properties provide a concise way to set values for multiple related properties at once.

- For shorthand properties such margin and padding, specify the values in clockwise order, beginning with the top.
- Progressive enhancement allows us to use cutting-edge CSS without breaking the page in older browsers.

2 Working with relative units

This chapter covers

- The versatility of relative units
- How to use ems and rems, without letting them drive you mad
- Using viewport-relative units
- An introduction to CSS variables

When it comes to specifying *length* values, CSS provides a wide array of options to choose from. One of the most familiar, and probably easiest to work with, is pixels. These are a type of *absolute* unit; that is, 5 px always means the same thing. Some other units, such as em and rem, are not absolute, but *relative*. The value of relative units changes, based on external factors; for example, the meaning of 2 em changes depending on which element (and sometimes even which property) you're using it on. Naturally, this makes relative units more difficult to work with.

Definition

Length is the formal name for a CSS value that denotes a distance measurement. It's a number followed by a unit, such as `5px`. Percentages are similar to lengths, but strictly speaking, they're not considered lengths.

When working with lengths, developers, even experienced CSS developers, often dislike working with relative units—the way the value of a relative unit can change makes it seem unpredictable and less clear-cut than an absolute unit such as the pixel. In this chapter, I'll remove the mystery surrounding relative units. First, I'll explain the unique value they bring to CSS, then I'll help you make sense of them. I'll explain how they work, and I'll show you how to tame their seemingly unpredictable nature. You can make relative values work for you; wielded correctly, they'll make your code simpler, more versatile, and easier to work with.

2.1 The power of relative units

CSS brings a late-binding of styles to the web page: The content and its styles aren't pulled together until after the authoring of both is complete. This adds a level of complexity to the design process that doesn't exist in other types of graphic design, but it also provides more power: one stylesheet can be applied to hundreds, even thousands, of pages. Furthermore, the final rendering of the page can be altered directly by the user, who can, for example, change the default font size or resize the browser window.

In early computer application development (as well as in traditional publishing), developers (or publishers) knew the exact constraints of their medium. A particular program window might be 400-px wide by 300-px tall, or a page could be 4 in. wide by 6½ in. tall. Consequently, when developers set about laying out the application's buttons and text, they knew exactly how big they could make those elements and exactly how much space that would leave them to work with for other items on the screen. On the web, this is not the case.

2.1.1 The rise of responsive design

In the web environment, the user can set their browser window to any number of sizes, and the CSS has to apply to it. Furthermore, users can resize the page after it's opened, and the CSS needs to adjust to new constraints. This means that styles can't be applied when you create your page; the browser must calculate those when the page is rendered onscreen.

This adds a layer of abstraction to CSS. You can't style an element according to an ideal context; we need to specify rules that'll work in any context where that element could be placed. With today's web, your page will need to render on a 4-inch phone screen as well as on a 30-inch monitor.

Pixels, points, and picas

CSS supports several absolute length units, the most common of which, and the most basic, is the pixel (px). Less common absolute units are mm (millimeter), cm (centimeter), Q (quarter-millimeter), in. (inch), pt (point—typographic term for 1/72nd of an inch), and pc (pica—typographic term for 12 points). Any of these units can be translated directly to another if you

want to work out the math: 1 in. = 25.4 mm = 101.6 Q = 2.54 cm = 6 pc = 72 pt = 96 px. Therefore, 16 px is the same as 12 pt (16 / 96 × 72). Designers are often more familiar with the use of points, where developers are more accustomed to pixels, so you may have to do some translation between the two when communicating with a designer.

Pixel is a slightly misleading name—a CSS pixel does not strictly equate to a monitor's pixel. This is notably the case on high-resolution ("retina") displays. Although the CSS measurements can be scaled a bit, depending on the browser, the operating system, and the hardware, 96 px is usually in the ballpark of 1 physical inch onscreen, though this can vary on certain devices or with a user's resolution settings.

With such a varied array of user devices and screen sizes, we have to constantly be aware of *responsive design*. If you give an element a width of 800 px, how will that look in a smaller window? How will a horizontal menu look if it doesn't all fit on one line? As you write your CSS, you need to be able to think simultaneously in specifics, as well as in generalities. When you have multiple ways to solve a particular problem, you'll need to favor the solution that works more generally under multiple and different circumstances.

Definition

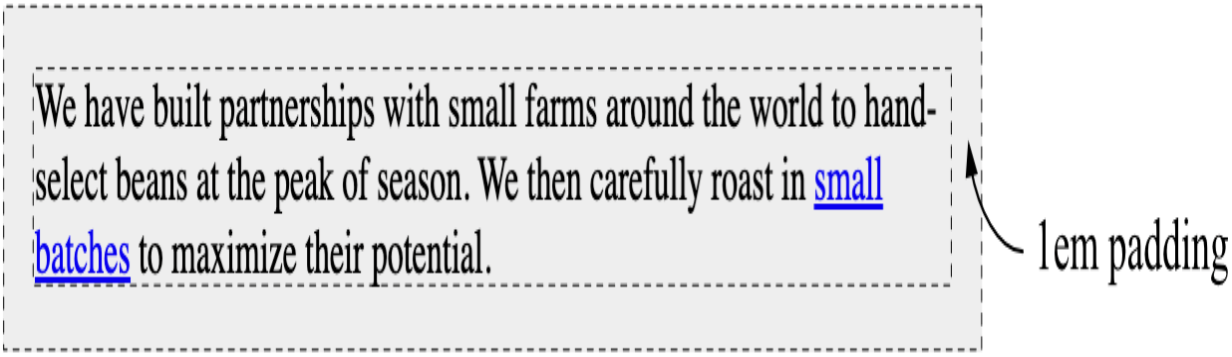
Responsive design is a term coined by Ethan Marcotte, which refers to styles that "respond" differently, based on the size of the browser window. This entails intentional consideration for mobile, tablet, or desktop screens of any size. We'll take a good look at responsive design in chapter 8, but in this chapter, I'll lay some important groundwork before we get there.

Relative units are one of the tools CSS provides to work at this level of abstraction. Instead of setting a font size at 14 px, you can set it to scale proportionally to the size of the window. Or, you can set the size of everything on the page relative to the base font size, and then resize the entire page with a single line of code. Let's take a look at what CSS provides to make this sort of approach possible.

2.2 Ems and rems

Ems, the most common relative length unit, are a measure used in typography, referring to a specified font size. In CSS, 1 em means the font size of the current element; its exact value varies depending on the element you're applying it to. Figure 2.1 shows a `div` with 1 em of padding.

Figure 2.1 1em of padding is equal to the font size (dashed lines added to illustrate padding)



The code to produce this is shown in the next listing. The ruleset specifies a font size of 16 px, which becomes the element's local definition for 1 em. Then the code uses ems to specify the padding of the element. Add this to a new stylesheet, and put some text in a `<div class="padded">` to see it in your browser.

Listing 2.1 Applying ems to padding

```
.padded {
  font-size: 16px;
  padding: 1em;       #A
}
```

This padding has a specified value of `1em`. This is multiplied by the font size, producing a rendered padding of 16 px. This is important: Values declared using relative units are evaluated by the browser to an absolute value, called the *computed value*.

In this example, editing the padding to 1.5em would produce a computed value of 24px. If another selector targets the same element and overrides it

with a different font size, it’ll change the local meaning of em, and the computed padding will change to reflect that.

Using ems can be convenient when setting properties like `padding`, `height`, `width`, or `border-radius` because these scale evenly with the element if it inherits different font sizes or if the user changes the font settings.

Figure 2.2 shows two differently sized boxes. The font size, padding, and border radius in each is not the same.

Figure 2.2 Relatively sized padding and border radius change with the font size

You can define the styles for these boxes by specifying the padding and border radius using ems. By giving each a padding and border radius of 1 em, you can specify a different font size for each element, and the other properties will scale along with the font.

In your HTML, create two boxes as shown next. Add the `box-small` and `box-large` classes to each, respectively, as size modifiers:

```
<span class="box box-small">Small</span>
<span class="box box-large">Large</span>
```

Now, add the styles in listing 2.2 to your stylesheet. This defines a box using ems. It also defines small and large modifiers, each scaling the elements with a different font size.

Listing 2.2 Applying ems applied to different elements

```
.box {
  padding: 1em;
```

```
  border-radius: 1em;
  background-color: lightgray;
}

.box-small {
  font-size: 12px;     #A
}

.box-large {
  font-size: 18px;     #A
}
```

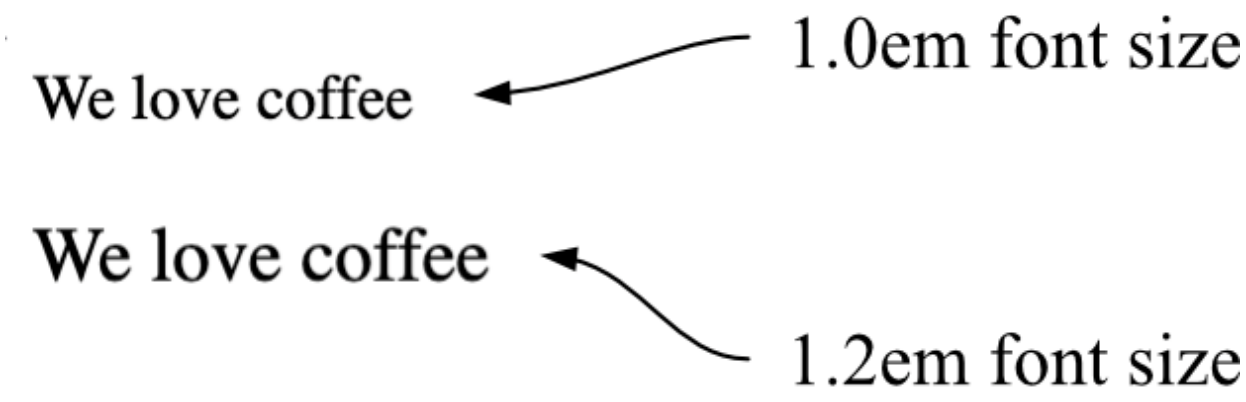
This is a powerful feature of ems. You can define the size of an element and then scale the entire thing up or down with a single declaration that changes the font size. You'll build another example of this in a bit, but first, let's talk about ems and font sizes.

2.2.1 Using ems to define font-size

When it comes to the `font-size` property, ems behave a little differently. As I said, ems are defined by the current element's font size. But, if you declare `font size: 1.2em`, what does that mean? A font size can't equal 1.2 times itself. Instead, `font-size` ems are derived from the inherited font size.

For a basic example, figure 2.3 shows two bits of text, each at a different font size. You'll define these using ems in listing 2.3.

Figure 2.3 Two different font sizes using ems



Change your page to match the following listing. The first line of text is inside the `<body>` tag, so it'll render at the body's font size. The second part, the slogan, inherits that font size.

```
<body>
  We love coffee
  <p class="slogan">We love coffee</p>   #A
</body>
```

The CSS in the next listing specifies the body's font size. I've used pixels here for clarity. Next, you'll use ems to scale up the size of the slogan.

Listing 2.3 Applying ems to font-size

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

.slogan {                #A
  font-size: 1.2em;      #A
}                        #A
```

The slogan's specified font size is 1.2 em. To determine the calculated pixel value, you'll need to refer to the inherited font size of 16 px: 16 times 1.2 equals 19.2, so the calculated font size is 19.2 px.

Tip

If you know the pixel-based font size you'd like, but want to specify the declaration in ems, here's a simple formula: divide the desired pixel size by the parent (inherited) pixel size. For example, if you want a 10 px font and your element is inheriting a 12 px font, 10 / 12 = 0.8333 em. If you want a 16 px font and the parent font is 12 px, 16 / 12 = 1.3333 em. We'll do this calculation several times throughout this chapter.

It's helpful to know that, for most browsers, the default font size is 16 px. Technically, it's the keyword value `medium` that calculates to 16 px.

Ems for font size together with ems for other properties

You've now defined ems for font-size (based on an inherited font size). And, you've defined ems for other properties like `padding` and `border-radius` (based on the current element's font size). What makes ems tricky is when you use them for both font size and any other properties on the same element. When you do this, the browser must calculate the font size first, and then it uses that value to calculate the other values. Both properties can have the same declared value, but they'll have different computed values.

In the previous example, we calculated the font size to be 19.2 px (16 px inherited font size times 1.2 em). Figure 2.4 shows the same slogan element, but with an added padding of 1.2 em and a gray background to make the padding size more apparent. This padding is a bit larger than the font size, even though both have the same declared value.

Figure 2.4 A font size in ems differs from padding in ems

What's happening here is the paragraph inherits a font size of 16 px from the body, producing a calculated font size of 19.2 px. This means that 19.2 px is now the local value for an em, and that value is used to calculate the padding. The CSS for this is shown next. Update your stylesheet to see this in your test page.

Listing 2.4 Applying ems to font-size and padding

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

.slogan {
  font-size: 1.2em;   #A
  padding: 1.2em;     #B
  background-color: #ccc;
}
#A Evaluates to 19.2 px
#B Evaluates to 23.04 px
```

In this example, `padding` has a specified value of 1.2 em. This multiplied by 19.2 px (the current element's font size) produces a calculated value of 23.04

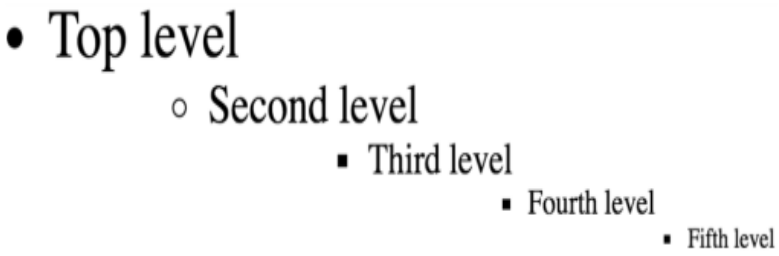
px. Even though `font-size` and `padding` have the same specified value, their calculated values are different.

The shrinking font problem

Ems can produce unexpected results when you use them to specify the font sizes of multiple nested elements. To know the exact value for each element, you'll need to know its inherited font size, which, if defined on the parent element in ems, requires you to know the parent element's inherited size, and so on up the tree.

This becomes quickly apparent when you use ems for the font size of lists and then nest lists several levels deep. Almost every web developer at some point in their career loads their page to find something resembling figure 2.5. The text is shrinking! This is exactly the sort of problem that leaves developers dreading the use of ems.

Figure 2.5 Font size of 0.8em causing shrinking text when applied to nested lists



Shrinking text occurs when you nest lists several levels deep and apply an em-based font size to each level. Listing 2.5 provides an example of this by setting the font size of unordered lists to 0.8 em. The selector targets every `<ul>` on the page; so when these lists inherit their font size from other lists, the ems compound.

Listing 2.5 Applying ems to a list

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

ul {
```

```
  font-size: 0.8em;
}
```

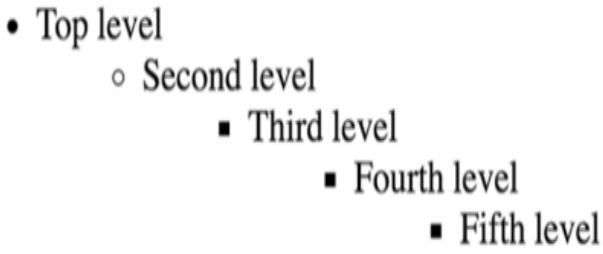
If you were to apply these styles to a page with multiple nested lists, you would see the issue. When applied to a page with markup such as listing 2.6, each `<ul>` inherits its font size from the parent list, and the em value reduces it.

Listing 2.6 Nested lists

```
<ul>
  <li>Top level
    <ul>                        #A
      <li>Second level          #A
        <ul>                      #B
          <li>Third level         #B
            <ul>                      #C
              <li>Fourth level        #C
                <ul>
                  <li>Fifth level</li>
                </ul>
              </li>
            </ul>
          </li>
        </ul>
      </li>
    </ul>
  </li>
</ul>
```

Each list has a font size 0.8 times that of its parent. This means the first list has a font size of 12.8 px, but the next one down is 10.24 px (12.8 px × 0.8), and the third level is 8.192 px, and so on. Similarly, if you specified a size larger than 1 em, the text would continually grow instead. What you want is to specify the font at the top level, then maintain the same font size all the way down, as in figure 2.6.

Figure 2.6 Using a font size of 1em for nested lists keeps the text size consistent



One way you can accomplish this is with the code in listing 2.7. This sets the font size of the first list to 0.8 em as before (listing 2.5). The second selector in the listing then targets all unordered lists within an unordered list—all of them except the top level. The nested lists now have a font size equal to their parents, as shown in figure 2.6.

Listing 2.7 Correcting the shrinking text

```
ul {
  font-size: 0.8em;
}

ul ul {                #A
  font-size: 1em;  #A
}                      #A
```

This fixes the problem, though it’s not ideal; you’re setting a value and then immediately overriding it with another rule. It would be nicer if you could avoid overriding rules by inching up the specificity of the selectors.

By now, it should be clear that ems can get away from you if you’re not careful. They’re nice for padding, margins, and element sizing, but when it comes to font size, they can get complicated. Thankfully, there is a better option—*rems*.

2.2.2 Using rems for font-size

When the browser parses an HTML document, it creates a representation in memory of all the elements on the page. This representation is called the DOM (Document Object Model). It’s a tree structure, where each element is

represented by a node. The `<html>` element is the top-level (or root) node. Beneath it are its child nodes, `<head>` and `<body>`. And beneath those are their children, then their children, and so on.

The root node is the ancestor of all other elements in the document. It has a special pseudo-class selector (`:root`) that you can use to target it. This is equivalent to using the type selector `html` with the specificity of a class rather than a tag.

Rem is short for “root em.” Instead of being relative to the current element, rems are relative to the root element. No matter where you apply it in the document, 1.2 rem has the same computed value: 1.2 times the font size of the root element. The following listing establishes the root font size and then uses rems to define the font size for unordered lists relative to that.

Listing 2.8 Specifying font size using rems

```
:root {               #A
  font-size: 1em;        #B
}

ul {
  font-size: 0.8rem;
}
```

In this example, the root font size is the browser’s default of 16 px (an em on the root element is relative to the browser’s default). Unordered lists have a specified font size of 0.8 rem, which calculates to 12.8 px. Because this is relative to the root, the font size remains constant, even if you nest lists.

Rems simplify a lot of the complexities involved with ems. In fact, they offer a good middle ground between pixels and ems by providing the benefits of relative units, but are easier to work with. Does this mean you should use rems everywhere and abandon the other options? No.

In CSS, again, the answer is often, “It depends.” Rems are but one tool in your tool bag. An important part of mastering CSS is learning when to use which tool. My default is to use rems for font sizes, pixels for borders, and ems or rems for most other measures, especially paddings, margins, and border radius.

This way, font sizes are predictable, but you'll still get the power of ems scaling your padding and margins, should other factors alter the font size of an element. Pixels make sense for borders, particularly when you want a nice fine line. These are my go-to units for the various properties—though my preferences change over time and between projects. They're tools, and in some circumstances, a different tool does the job better.

Tip

When in doubt, use rems for font size, pixels for borders, and either ems or rems for most other properties.

Accessibility: use relative units for font size

Some browsers provide two ways for the user to customize the size of text: zoom and a default font size. By pressing Ctrl-plus (+) or Ctrl-minus (–) — or Cmd-plus/minus on a Mac — the user can zoom the page up or down. This visually scales all fonts and images and generally makes everything on the page larger or smaller. In some browsers, this change is only applied to the current tab and is temporary, meaning it doesn't get carried over to new tabs.

Setting a default font size is a bit different. This is usually set in the browser's settings page, but changes at this level persist until the user changes the value again. The catch is that this setting does not resize fonts defined using pixels or other absolute units. Because a default font size is vital to some users, particularly those who are vision-impaired, you should always specify font sizes with relative units or percentages.

Working with pixels is initially easier to understand, but if you take the time to become familiar with rems, you will have a broader range of tools at your disposal, and you can produce more accessible pages at the same time.

2.3 Stop thinking in pixels

One pattern, or rather, antipattern, that has been common in the past is to reset the font size at the page's root to 0.625 em or 62.5%. This takes the

browser's default font size, 16 px, and scales it down to 10 px. This practice simplifies the math: If your designer tells you to make the font 14 px, you can easily divide by 10 in your head and type 1.4 rem, all while still using relative units. An example of this is shown in the following CSS:

```
html {
  font-size: 0.625em;
}
```

I don't recommend doing this.

Initially, this may be convenient, but there are two problems with this approach. First, it forces you to write a lot of duplicate styles. Ten pixels is too small for text, so you'll have to override it throughout the page. You'll find yourself setting paragraphs to 1.6 rem and asides to 1.6 rem and `nav` links to 1.6 rem and so on. This introduces more places for error, more points of contact in your code when it needs to change, and increases the size of your stylesheet.

The second problem is that when you do this, you're still thinking in pixels. You might type 1.6 rem into your code, but in your mind, you're still thinking "16 pixels." On the responsive Web, you should get comfortable with "fuzzy" values. It doesn't matter how many pixels 1.2 em evaluates to; all you need to know is that it's a bit bigger than the inherited font size. And, if it doesn't look how you want it on screen, change it. This takes some trial and error, but in reality, so does working with pixels. (In a later chapter, we'll look at additional concrete rules to refine this approach.)

When working with ems, it's easy to get bogged down obsessing over exactly how many pixels things will evaluate to, especially font sizes. You'll drive yourself mad dividing and multiplying em values as you go. Instead, I challenge you to get into the habit of using ems first. If you're accustomed to using pixels, using em values may take practice, but it's worth it.

This isn't to say you'll never have to work with pixels. If you're working with a designer, you'll probably need to talk in some concrete pixel numbers, and that's okay. At the beginning of a project, you'll need to establish a base font size (and often a few common sizes for headings and

footnotes). Absolute values are easier to use when discussing the size of things.

Converting to rems involves arithmetic, so keep a calculator handy. (I press Command-Space on my Mac, and type the equation into Spotlight.) Putting a root font size in place defines a rem. From that point on, working in pixels should be the exception, not the norm.

I'll continue to mention pixels throughout this chapter. This will help me iterate why the relative units behave the way they do, as well as help you get accustomed to the calculation of ems. After this chapter, I'll primarily discuss font sizes using relative units.

2.3.1 Setting a sane default font size

Let's say you want your default font size to be 14 px. Instead of setting a 10 px default then overriding it throughout the page, set that value at the root. The desired value divided by the inherited value—in this case, the browser's default—is 14/16, which equals 0.875. Using ems here allows you to adjust the default font size while still respecting any user font settings.

Add the following listing to the top of a new stylesheet, because you'll be building on it. This sets the default font at the root (`<html>`).

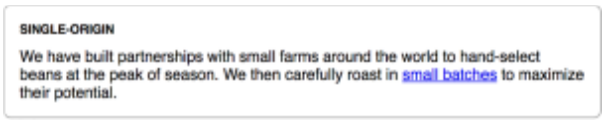
Listing 2.9 Setting the true default font size

```
:root {   #A
  font-size: 0.875em;   #B
}
```

Now your desired font size is applied to the whole page. You won't need to specify it elsewhere. You'll only need to change it in places where the design deviates from this, such as headings.

Let's create the panel shown in figure 2.7. You'll build this panel based on the 14 px font size, using relative measurements.

Figure 2.7 An example panel you will create using relative units and an inherited font size



The markup for this panel is shown here. Add this to your page.

Listing 2.10 Markup for a panel

```
<div class="panel">
  <h2>Single-origin</h2>
  <div class="panel-body">
    We have built partnerships with small farms around the world
to
    hand-select beans at the peak of season. We then carefully
roast
    in <a href="/batch-size">small batches</a> to maximize their
    potential.
  </div>
</div>
```

The next listing shows the styles. You'll use ems for the padding and border radius, rem for the font size of the heading, and px for the border. Add these to your stylesheet.

Listing 2.11 Panel with relative units

```
.panel {
  padding: 1em;             #A
  border-radius: 0.5em;     #A
  border: 1px solid #999;   #B
}

.panel > h2 {
  margin-top: 0;            #C
  font-size: 0.8rem;              #D
  font-weight: bold;              #D
  text-transform: uppercase;      #D
}
```

This code puts a thin border around the panel and styles the heading. I opted for a header that is smaller, but bold and all caps. (You can make this larger or a different typeface if your design calls for it.)

The > in the second selector is a *direct descendant combinator*; it indicates a direct parent-child relationship between two elements in the selector. In this case, the selector targets any `h2` that's a child element of a `.panel` element. (See appendix A for a complete reference of selectors and combinators.)

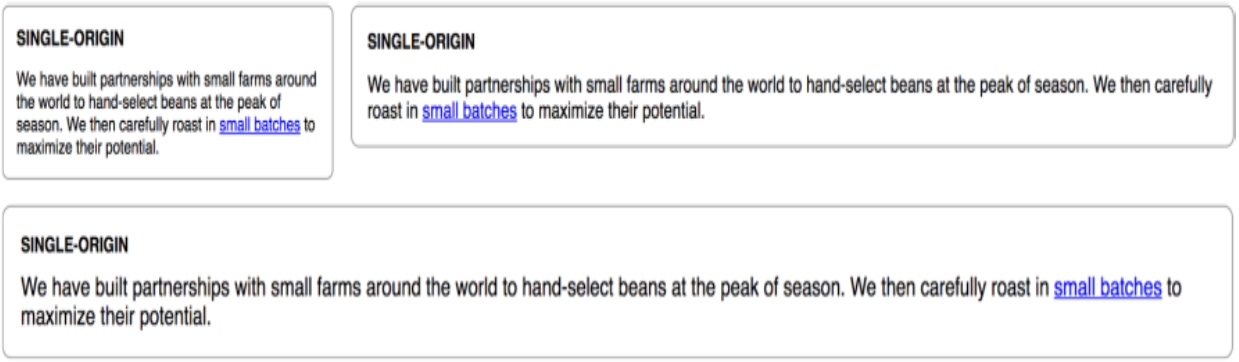
In listing 2.10, I added a `panel-body` class to the main body of the panel for clarity, but you'll notice you didn't need to use it in your CSS. Because this element already inherits the root font size, it already appears how you want it to look.

2.3.2 Making the panel responsive

Let's take this a bit further. You can use some media queries to change the base font size, depending on the screen size.

A media query uses an `@media` rule used to specify styles that will be applied only to certain screen sizes or media types (for example, print or screen). This is a key component of responsive design. This will make the panel render at different sizes based on the size of the user's screen (figure 2.8).

Figure 2.8 A responsive panel as seen on different screen sizes: 300 px (top left), 800 px (top right), and 1,440 px (bottom)



To see this result, edit this portion of your stylesheet to match this listing.

Listing 2.12 Responsive base font-size

```
:root {
  font-size: 0.85em;
}

@media (min-width: 800px) {
  :root {
    font-size: 1em;
  }
}

@media (min-width: 1200px) {
  :root {
    font-size: 1.15em;
  }
}
```

This first ruleset specifies a small default font size. This is the font size that we want to apply on smaller screens. Then you used media queries to override that value with incrementally larger font sizes on screens with a width of 800 px and 1,200 px or more.

By applying these font sizes at the root on your page, you've responsively redefined the meaning of em and rem throughout the entire page. This means that the panel is now responsive, even though you made no changes to it directly. On a small screen, such as a smartphone, the font will be rendered smaller (13.6 px); likewise, the padding and border radius will be smaller to match. And, on larger screens more than 800 px and 1,200 -px wide , the component scales up to a 16 px and 18.4 px font size, respectively. Resize your browser window to watch these changes take place.

If you are disciplined enough to style your entire page in relative units like this, the entire page will scale up and down based on the size of the user's browser window. This can be a huge part of your responsive strategy. These two media queries near the top of your stylesheet can eliminate the need for dozens of media queries throughout the rest of your CSS. But it doesn't work if you define your values in pixels.

Similarly, if you later decide the fonts on a site are too small or too large, you can change them globally by touching only one line of code. The change

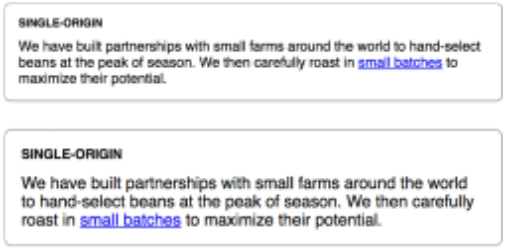
will ripple throughout the rest of your page, effortlessly.

2.3.3 Resizing a single component

You can also use ems to scale an individual component on the page. Sometimes you might need a larger version of the same part of your interface on certain parts of the page. Let's do this with our panel. You'll add a `large` class to the panel: `<div class="panel large">`.

Figure 2.9 shows both the normal and the large panel for comparison. The effect is similar to the responsive panels, but both sizes can be used simultaneously on the same page.

Figure 2.9 A panel defined in ems can be enlarged by increasing its font size



Let's make a small change to the way you defined the panel's font sizes. You'll still use relative units, but you'll adjust what they're relative to. First, add the declaration `font-size: 1rem` to the parent element of each panel. This means each panel will establish a predictable font size for itself, no matter where it's placed on the page.

Second, redefine the heading's font size using ems rather than rems to make it relative to the parent's font size you just established at 1 rem. The code for this is in listing 2.13. Update your stylesheet to match.

Listing 2.13 Creating a larger version of the panel

```
.panel {
  font-size: 1rem;      #A
  padding: 1em;
  border: 1px solid #999;
  border-radius: 0.5em;
}
```

```
.panel > h2 {
  margin-top: 0;
  font-size: 0.8em;   #B
  font-weight: bold;
  text-transform: uppercase;
}
```

This change has no effect on the appearance of the panel, but now it sets you up to make the larger version of the panel with a single declaration. All you have to do is override the parent element's 1-rem font size with another value. Because all the component's measurements are relative to this, overriding it will resize the entire panel. Add the CSS in the next listing to your stylesheet to define a larger version.

```
.panel.large {      #A
  font-size: 1.2rem;
}
```

Now, you can use `class="panel"` for a normal panel and `class="panel large"` for a larger one. Similarly, you could define a smaller version of the panel by setting a smaller font size. If the panel were a more complicated component, with multiple font sizes or paddings, it'd still take only this one declaration to resize it, as long as everything inside is defined using ems.

You don't need to follow this pattern exactly as I've shown. This dynamic nature of ems and rems is a powerful tool you can adjust to suit your needs. For instance, if you don't want your border radius to scale, define it using rems, and use ems only for the properties you want to make scalable. Taking the time to familiarize yourself with these relative units will provide you with a lot more options than simply defining everything in pixels.

CSS is a living standard

CSS is defined by a large number of W3C specifications. It began as a single specification with a version number, but this approach changed after version 2.1.

After that point, the specification was broken up into individual modules, each independently versioned. The specification for backgrounds and

borders is now separate from the one for the box model, and from the one for cascading and inheritance. This allows the W3C to make new revisions to one area of CSS without unnecessarily updating areas that are not changing. Many of these specifications remain at version 3 (now called level 3), but some, such as the selectors specification, are at level 4 or higher while others, such as a flexbox, are at level 1.

This means we're no longer working with one particular version of CSS. It's a living standard. Each browser is continually adding support for new features. Developers work with those changes and adapt to them.

You may occasionally hear the term "CSS3", but this does not strictly refer to a version 3.0 of one specification, but rather a series of specifications that were all published in a short timeframe in the early 2010s. There won't be a CSS4, except perhaps as a more generic marketing term. Although this book covers CSS3 features, I don't necessarily call them out as such because, as far as the web is concerned, it's all CSS.

Ems and rems are the most commonly used relative units, but they aren't the only relative unit types available. Now that you've seen the flexibility they provide, let's look at another import type of relative unit.

2.4 Viewport-relative units

You've learned that ems and rems are defined relative to `font-size`, but these aren't the only type of relative units. There are also *viewport-relative units* for defining lengths relative to the browser's viewport.

Definition

The viewport is the framed area in the browser window where the web page is visible. This excludes the browser's address bar, toolbars, and status bar, if present.

With some recent additions to the language, there is a grand total of 24 viewport-relative units in CSS. This probably sounds overwhelming, but these are all the permutations of a few relatively simple concepts mixed and

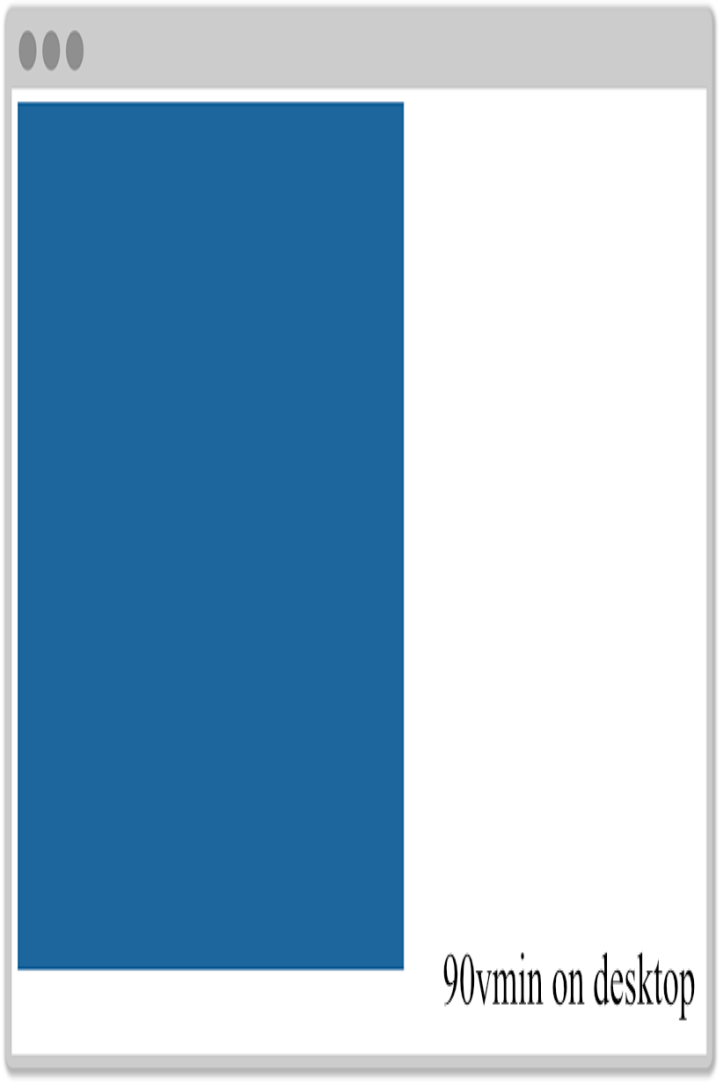
matched in various ways. We'll start with the basics. Here are the four basic units that were first added to the language:

- *vh*—1% of the viewport height
- *vw*—1% of the viewport width
- *vmin*—1% of the smaller dimension, height or width
- *vmax*—1% of the larger dimension, height or width

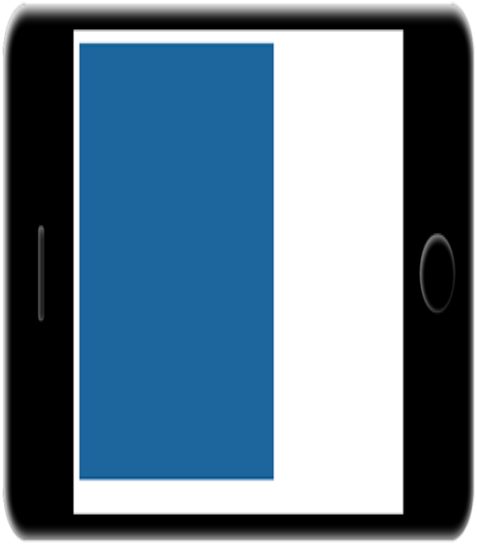
For example, 50 vw is equal to half the width of the viewport, and 25 vh equals 25% of the viewport's height. vmin is based on which of the two (height or width) is smaller. This is helpful for ensuring that an element will fit on the screen regardless of its orientation: If the screen is landscape, it'll be based on the height; if portrait, it's based on the width.

Figure 2.10 shows a square element as it appears in several viewports with different screen sizes. It's defined with both a height and a width of 90 vmin, which equals 90% of the smaller of the two dimensions—90% of the height on landscape screens, or 90% of the width on portrait.

Figure 2.10 An element with a height and width of 90 vmin will always display as a square a little smaller than the viewport, regardless of its size or orientation.


90vmin on desktop


90vmin on
mobile in portrait


90vmin on
mobile in landscape

Listing 2.18 shows the styles for this element. It produces a large square that always fits in the viewport no matter how the browser is sized. You can add a `<div class= "square">` to your page to see this.

Listing 2.14 Square element sized using vmin

```
.square {
  width: 90vmin;
  height: 90vmin;
  background-color: #369;
}
```

The viewport-relative lengths are great for things like making a large hero image fill the screen. Your image can be inside a long container, but setting the image height to 100 vh, makes it exactly the height of the viewport—at least, that was the theory.

After developers started using these units to fill the viewport with hero images, we discovered a problem: on mobile devices the viewport size can change dynamically. In an attempt to maximize available screen size, mobile browsers have a feature where some UX controls—the address bar at the top of the screen and the navigation buttons at the bottom—slide out of view as you scroll down the page. And they slide back into view if you scroll back up.

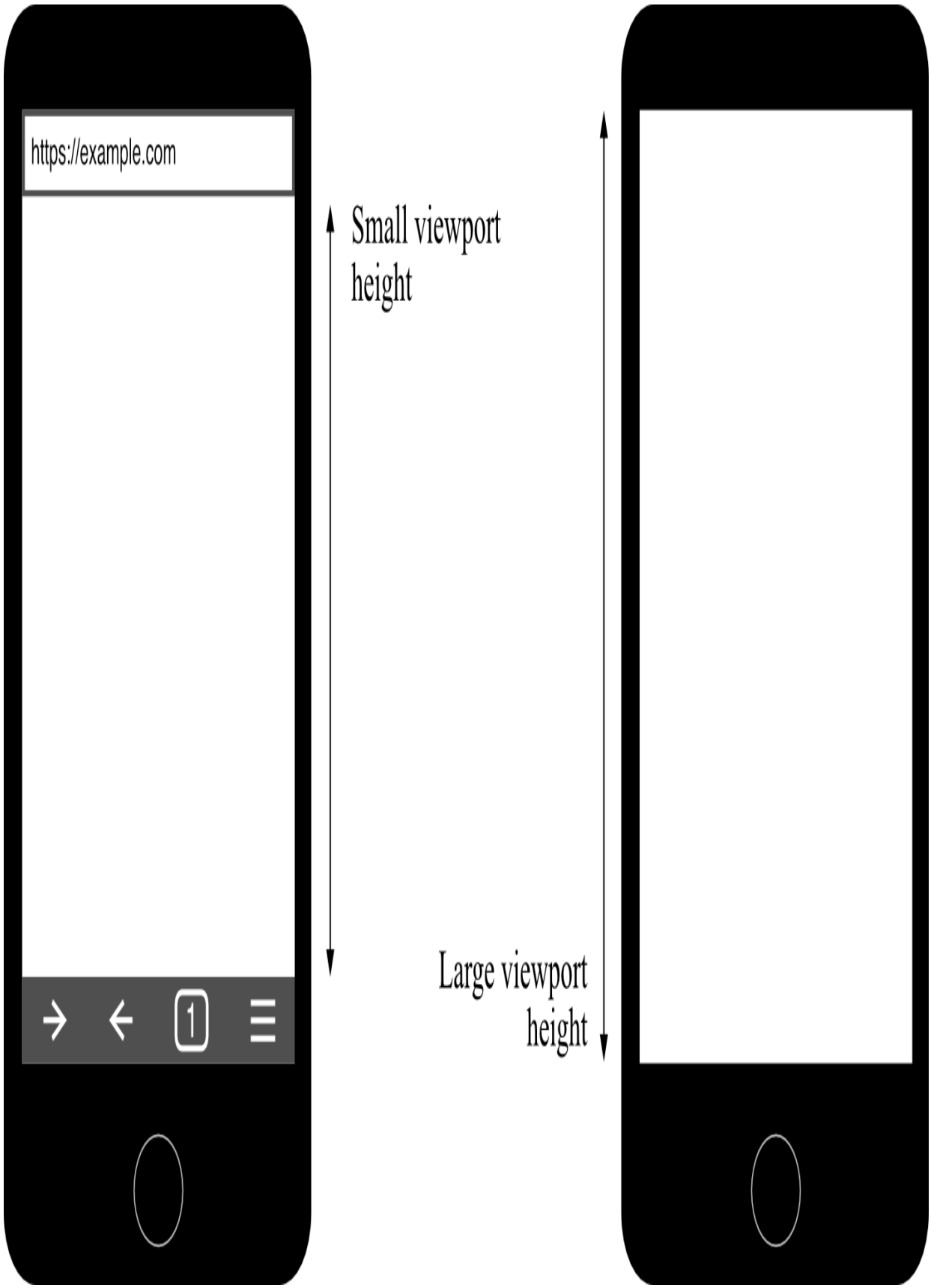
These dynamic changes cause a resize of the viewport, which in turn causes elements on the page using vh units to change size, and content beneath them to jump around on the screen. You can imagine a scenario where the user has scrolled past multiple 100 vh boxes, but then scrolls up slightly. This can cause all the content in view to snap upward by hundreds of pixels, making for a very jarring reading experience. It also has performance implications, because so much of the page layout has to be recalculated by the browser (often called *layout thrashing*).

In the end, most mobile browsers stopped this behavior by re-interpreting vh units based on the largest possible viewport size and ignoring the impact the address bar has on viewport size. But sometimes as a developer, this is not the behavior you want, so several more units were added to the CSS specification.

2.4.1 Selecting from the newer viewport units

To address the problem of layout thrashing, CSS introduced the concept of large and small viewports. The *large viewport* is the biggest possible viewport, when all the browser's UX elements are hidden. The *small viewport* is the smallest possible viewport, when all the UX elements are shown (figure 2.11).

Figure 2.11 Comparison of the large and small viewports on a mobile device


https://example.com
1
Small viewport height
Large viewport height

Large viewport units are available for width, height, min, and max, just like the regular viewport units. Prepend the letter “l” to use large viewport units: `lvw`, `lvh`, `lvmin`, `lvmax`. Similarly, prepend the letter “s” to use small viewport units: `svw`, `svh`, `svmin`, `svmax`.

With these new units, you have the ability to choose which behavior is more important to you. Is it important for a full-height masthead to fill the entire screen, even if that means it might extend a bit past the bottom? If so, use `height: 100lvh`. Or is it more important that the entire masthead remains visible, even if there may be a little space beneath it at the bottom of the screen? Then use `height: 100svh`.

At this time, I have been unable to find examples of any browser differentiating between the widths of the small and large viewports, so for all practical purposes, vw, svw, and lvw all behave the same. However, it’s important to keep in mind that they are conceptually different, and it is entirely possible that browsers will make a distinction between them in the future. For example, the Edge browser on desktop has a dynamic sidebar that opens and closes; this doesn’t impact svw and lvw at this time, but that behavior could change down the road.

Here are a few other caveats to keep in mind:

- Viewport units do not take scrollbars into account. This unfortunately means an element of width 100svw will introduce horizontal scrolling when a vertical scrollbar is present.
- The CSS specification does not dictate whether an on-screen keyboard should shrink the small viewport size. It is currently taken into consideration in some Android browsers, but not on iOS; these could potentially change in the future.
- The original viewport units generally behave like large viewport units in most browsers, but this behavior is not guaranteed.

This leaves us with one use case that isn’t addressed: there may be instances where we want the original behavior of viewport units, where they dynamically shift if when the browser’s UX elements appear or hide. For this behavior, there is a third type of viewport units: the *dynamic viewport*. For these, prepend the letter “d” to the viewport units: `dvw`, `dvh`, `dvmin`,

`dvmax`. These behave like small viewport units while the viewport is small, but like large viewport units while the viewport is large.

You should use the dynamic viewport units sparingly. As a user on a mobile device scrolls up and down, it can cause layout thrashing if element heights change dynamically.

Table 2.1 lists of all the available viewport units. For completeness, this includes an additional set of unit types: inline and block. These are called “logical properties”, which behave like width and height, respectively, but are transposed for vertically-written languages such as Japanese. I’ll cover logical properties in more depth in chapter 3.

Table 2.1 All viewport unit types

	Unspecified viewport (original units)	Large viewport	Small viewport	Dynamic viewport

width/height	`vw`
	
	`vh`
		`lvw`
		
		`lvh`
			`svw`
			
			`svh`
				`dvw`
				
				`dvh`

min/max	`vmin`
	
	`vmax`
		`lvmin`
		
		`lvmax`
			`svmin`
			
			`svmax`
				`dvmin`
				
				`dvmax`

inline/block	`vi`
	
	`vb`
		`lvi`
		
		`lvb`
			`svi`
			
			`svb`
				`dvi`
				
				`dvb`

For many uses beyond mastheads or other similar full-screen or half-screen elements, there may not be a strong reason to favor units of one viewport

size over another. With so many viewport units available, this can lead to a bit of decision paralysis. My go-to when this happens is to use small viewport units.

2.4.2 Using viewport units for font size

One application for viewport-relative units that may not be immediately obvious is font size. In fact, I find this use more practical than applying vh and vw to element heights or widths.

Consider what would happen if you applied `font-size: 2svw` to an element. On a desktop monitor at 1,200 px, this evaluates to 24 px (2% of 1,200). On a tablet with a screen width of 768 px, it evaluates to about 15 px (2% of 768). And, the nice thing is, the element scales smoothly between the two sizes. This means there're no sudden breakpoint changes; it transitions incrementally as the viewport size changes.

Unfortunately, 24 px is a bit too large on a big screen. And worse, it scales all the way down to around 7.5 px on an iPhone SE. What would be nice is this scaling effect, but with the extremes a little less severe. You can achieve this with either of CSS's `calc()` or `clamp()` functions.

Getting responsive with the calc() function

The `calc()` function lets you do basic arithmetic with two or more values. This is particularly useful for combining values that are measured in different units. This function supports addition (+), subtraction (-), multiplication (*) and division (/). The addition and subtraction operators must be surrounded by whitespace, so I suggest getting in the habit of always adding a space before and after each operator; for example, `calc(1em + 10px)`.

You'll use `calc()` in the next listing to combine ems with svw units. Remove the previous base font size (and the related media queries) from your stylesheet. Add the following in its place:

```
:root {
  font-size: calc(0.5em + 1svw);
```

```
}
```

Now, open the page and slowly resize your browser to be wider or narrower. You'll see the font scale smoothly as you do. The `0.5em` here operates as a sort of minimum font size, and the `1svw` adds a responsive scalar. This gives you a base font size that scales from 11.75 px on an iPhone SE up to 20 px in a 1,200-px browser window.

Warning

When using viewport units for font size, always make sure part of the calculation includes ems or rems for accessibility. This ensures the user's font settings are taken into account in the final rendered size.

You can adjust these values to your liking, but it can be a little difficult to find something that works well without being too small in a small viewport or too large in a big one.

Improving things with the clamp() function

This responsive font size is useful, but a newer function, `clamp()`, provides a little better control. Clamp takes three arguments: a minimum value, a preferred value as an expression, and a maximum value. Update your page once more, using the following instead of the previous approach.

```
:root {
  font-size: clamp(0.9rem, 0.6rem + 0.5svw, 1.5rem);
}
```

This specifies a font size of `0.6rem + 0.5svw`, but the clamp function ensures the final value is never smaller than 0.9rem and never larger than 1.5rem. This way, very large or very small viewports do not end up with text sizes outside of a reasonable range.

Now that your font size is responsive, any other sizes on your page defined using ems or rems will scale responsively as well. You've accomplished a large piece of your responsive strategy without a single media query. Instead of three or four hard-coded breakpoints, everything on your page will scale fluidly according to the viewport. There is more to responsive design than

this—and we'll take a much deeper look in later chapters—but this will get you off to a nice start.

Tip

Two other related functions that may be useful from time to time are `min()` and `max()`. Min resolves to the smallest of the given values (e.g. `width: min(200px, 20svw);`). Max resolves to the largest of the given values (e.g. `min-height: max(200px, 20svw);`).

2.5 Unitless numbers and line-height

Some properties allow for unitless values (that is, a number with no specified unit). Properties that support this include `line-height`, `z-index`, and `font-weight` (700 is equivalent to bold; 400 is equivalent to normal, and so on). You can also use the unitless value `0` anywhere a length unit (such as px, em, or rem) is required because, in these cases, the unit does not matter —0 px equals 0% equals 0 em.

Warning

A unitless 0 can be used only for length values and percentages, such as in paddings, borders, and widths. It can't be used for angular values, such as degrees or time-based values like seconds.

The `line-height` property is unusual in that it accepts both units and unitless values. You should typically use unitless numbers because they're inherited differently. Let's put text into the page and see how this behaves. Add the code in the following listing to your page.

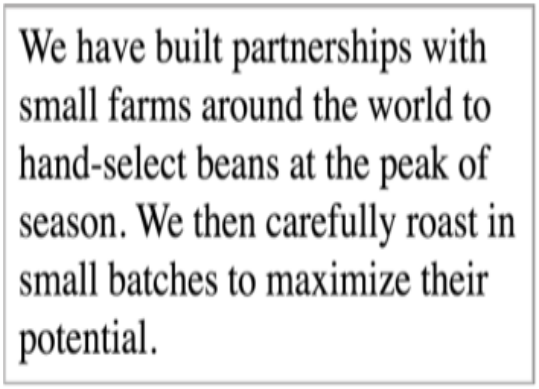
Listing 2.15 Inherited line-height markup

```
<body>
  <p class="about-us">
    We have built partnerships with small farms around the world
to
    hand-select beans at the peak of season. We then carefully
roast in
    small batches to maximize their potential.
```

```
  </p>
</body>
```

You'll specify a line height for the body element and allow it to be inherited by the rest of the document. This will work as expected, no matter what you do to the font sizes in the page (figure 2.12).

Figure 2.12 Unitless line height is recalculated for each descendant element, generally producing well-spaced lines of text.



Add listing 2.16 to your stylesheet for these styles. The paragraph inherits a line height of 1.2. Because the font size is 32 px (2 em × 16 px, the browser's default), the line height is calculated locally to 38.4 px (32 px × 1.2). This will leave an appropriate amount of space between lines of text.

Listing 2.16 Line height with a unitless number

```
body {
  line-height: 1.2;       #A
}

.about-us {
  font-size: 2em;
}
```

If instead you were to specify the line height using a unit, you can encounter unexpected results, like that shown in figure 2.13. The lines of text overlap

one another. Listing 2.17 shows the CSS that generated the overlap.

Figure 2.13 Using units in a line-height can produce undesirable spacing when inherited by child elements

Listing 2.17 Line height with units results in unexpected output

```
body {
  line-height: 1.2em;  #A
}

.about-us {
  font-size: 2em;     #B
}
```

These results are due to a peculiar quirk of inheritance: when an element has a value defined using a length (px, em, rem, and so forth), its computed value is inherited by child elements. When units such as ems are specified for a line height, their value is calculated, and that calculated value is passed down to any inheriting children. With the `line-height` property, this can cause unexpected results if the child element has a different font size, like the overlapping text.

When you use a unitless number, that declared value is inherited, meaning its computed value is recalculated for each inheriting child element. This will almost always be the result you want. Using a unitless number lets you set the line height on the body and then forget about it for the rest of the page, unless there are particular places where you want to make an exception.

2.6 Custom properties (aka CSS variables)

Custom properties enable a higher level of dynamic, context-based styling. These function in many ways like variables; you can declare a variable and assign it a value; then you can reference this value throughout your stylesheet. You can use this technique to reduce repetition in your stylesheet, as well as some other beneficial applications as you'll see shortly.

Note

If you use a CSS preprocessor that supports its own variables, such as Sass or Less, you may be tempted to disregard CSS variables. Don't. CSS variables are different in nature and are far more versatile than anything a preprocessor can accomplish. I tend to refer to them as "custom properties" rather than variables to emphasize this distinction.

To define a custom property, you declare it much like any other CSS property. The next listing is an example of a variable declaration. Start a fresh page and stylesheet, and add this CSS.

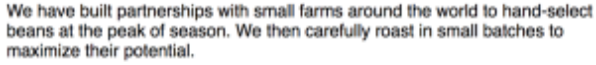
```
:root {
  --main-font: Helvetica, Arial, sans-serif;
}
```

This listing defines a variable named `--main-font` and sets its value to a set of common sans-serif fonts. The name must begin with two hyphens (`--`) to distinguish it from other CSS properties, followed by whatever name you'd like to use.

Variables must be declared inside a declaration block. I've used the `:root` selector here, which sets the variable for the whole page—I'll explain this shortly.

By itself, this variable declaration doesn't do anything until we use it. Let's apply it to a paragraph to produce a result like that in figure 2.14.

Figure 2.14 Simple paragraph using a variable's sans-serif font



A function called `var()` allows the use of variables. You'll use this function to reference the `--main-font` variable just defined. Add the ruleset shown in the following listing to put the variable to use.

Listing 2.18 Using a custom property

```
:root {
  --main-font: Helvetica, Arial, sans-serif;
}

p {                                  #A
  font-family: var(--main-font);     #A
}                                    #A
```

Custom properties let you define a value in one place, as a "single source of truth," and reuse that value throughout the stylesheet. This is particularly useful for recurring values like colors. The next listing adds a `brand-color` custom property. You can use this variable dozens of times throughout your stylesheet, but if you want to change it, you only have to edit it in one place.

Listing 2.19 Using custom properties for colors

```
:root {
  --main-font: Helvetica, Arial, sans-serif;
  --brand-color: #369;    #A
}
p {
  font-family: var(--main-font);
  color: var(--brand-color);
}
```

The `var()` function accepts an optional second parameter, which specifies a fallback value. If the variable specified in the first parameter is not defined, then the second value is used instead.

Listing 2.20 Providing fallback values

```
:root {
  --main-font: Helvetica, Arial, sans-serif;
  --brand-color: #369;
}

p {
```

```
  font-family: var(--main-font, sans-serif);   #A
color: var(--secondary-color, blue);     #B
}
#A Specifies a fallback value of sans-serif
#B The secondary-color variable is not defined, so the fallback
value blue is used.
```

This listing specifies fallback values in two different declarations. In the first, `--main-font` is defined as `Helvetica`, `Arial`, `sans-serif`, so this value is used. In the second, `--secondary-color` is an undefined variable, so the fallback value `blue` is used.

Note

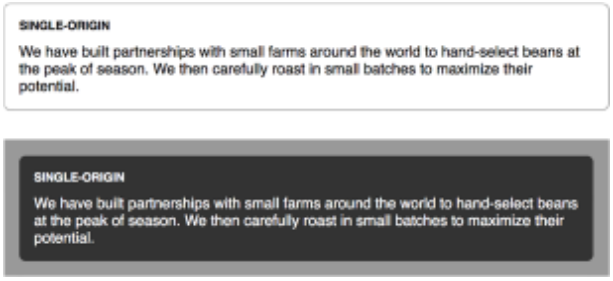
If a `var()` function evaluates to an invalid value, the property will be set to its initial value. For example, if the variable in `padding: var(--brand-color)` evaluates to a color, it would be an invalid padding value. In that case, the padding would be set to 0 instead.

2.6.1 Changing custom properties dynamically

In the examples so far, custom properties are merely a nice convenience; they can save you from a lot of repetition in your code. But what makes them particularly interesting is that the declarations of custom properties cascade and inherit: You can define the same variable inside multiple selectors, and the variable will have a different value for various parts of the page.

You can define a variable as black, for example, and then redefine it as white inside a particular container. Then, any styles based on that variable will dynamically resolve to black if they are outside the container and to white if inside. Let’s use this to achieve a result like that shown in figure 2.15.

Figure 2.15 Custom properties produce different colored panels based on local variable values.



This panel is similar to the one you created earlier, but with added support for a dark version. The HTML for this is shown in listing 2.21. It has two instances of the panel: one inside the body and one inside a dark section. Update your HTML to match this.

Listing 2.21 Two panels in different contexts on the page

```
<body>
  <div class="panel">              #A
    <h2>Single-origin</h2>
    <div class="body">
      We have built partnerships with small farms
      around the world to hand-select beans at the
      peak of season. We then careful roast in
      small batches to maximize their potential.
    </div>
  </div>

  <aside class="dark">              #B
    <div class="panel">             #B
      <h2>Single-origin</h2>
      <div class="body">
        We have built partnerships with small farms
        around the world to hand-select beans at the
        peak of season. We then careful roast in
        small batches to maximize their potential.
      </div>
    </div>
  </aside>
</body>
```

Next, redefine the panel to use variables for text and background color. Add the next listing to your stylesheet. This sets the background color to white and the text to black. I'll explain how this works before you add styles for the dark variant.

Listing 2.22 Using variables to define the panel colors

```
:root {
  --main-bg: #fff;      #A
  --main-color: #000;   #A
}

.panel {
  font-size: 1rem;
  padding: 1em;
  border: 1px solid #999;
  border-radius: 0.5em;
  background-color: var(--main-bg);  #B
  color: var(--main-color);            #B
}

.panel > h2 {
  margin-top: 0;
  font-size: 0.8em;
  font-weight: bold;
  text-transform: uppercase;
}
```

Again, you've defined the variables inside a ruleset with the `:root` selector. This is significant because it means these values are set for everything in the root element (the entire page). When a descendant element of the root uses the variables, these are the values they'll resolve to.

You have two panels, but they still look the same. Now, let's define the variables again, but this time with a different selector. The next listing provides styles for the dark container. It sets a dark gray background on the container, as well as a little padding and margin. It also redefines both variables. Add this to your stylesheet.

Listing 2.23 Styling the dark container

```
.dark {
  margin-top: 2em;          #A
  padding: 1em;
  background-color: #999;     #B
  --main-bg: #333;           #C
  --main-color: #fff;        #C
}
```

Reload the page, and the second panel has a dark background and white text. This is because when the panel uses these custom properties, they resolve to the values defined on the dark container, rather than on the root. Notice you didn't have to restyle the panel, or apply any additional classes.

In this example, you've defined custom properties twice: first on the root (where `--main-color` is black), and then on the dark container (where `--main-color` is white). The custom properties behave as a sort of scoped variable because the values are inherited by descendant elements. Inside the dark container, `--main-color` is white; elsewhere on the page, it's black.

Custom properties are an extremely versatile tool with countless applications. We will continue to use them in various ways throughout the rest of the book.

2.7 Summary

- Relative units can be used to specify sizes that adapt to the font size or viewport size.
- Ems define a length in terms of an element's font size, except when specifying font size itself, in which case ems are in terms of the element's inherited font size.
- Rems define a length in terms of the font size specified on the root `<html>` element.
- By scaling the root font size in a responsive design, elements on the page defined using ems and rems will scale at the same time.
- Viewport-relative units define a length in terms of the viewport's width or height.
- A line height defined with a unitless number will inherit more predictably to child elements.
- Custom properties work similar to variables, but can be manipulated dynamically via the cascade and inheritance.

3 Document flow and the box model

This chapter covers

- General tips for building a page layout
- Practical advice for element sizing
- Introduction to logical properties
- Negative margins and margin collapsing
- Consistent spacing of components on the page

When it comes to laying out elements on the page, you'll find a lot of things going on. On a complex site, you may have grids, absolutely positioned elements, and other elements of various sizes. You have a lot of things to keep track of, and learning everything involved with layout can be overwhelming.

We'll spend several chapters taking a close look at several layout techniques. Before we get to those, it's important to have a solid grasp on the fundamentals of how the browser sizes and places elements. The more advanced topics of layout are built atop concepts like document flow and the box model; these are the basic rules that determine the location and size of elements on the page.

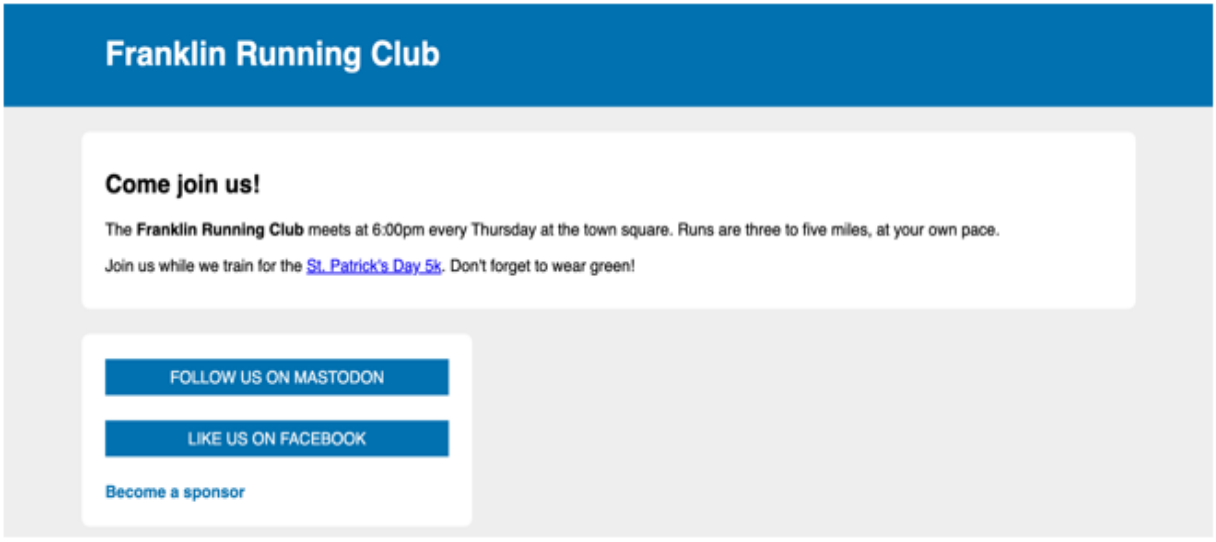
In this chapter, you'll build a basic one-column page layout. You may be familiar with this as a classic beginner exercise for CSS, but I'll guide you through it in a way that highlights several often-overlooked nuances of layout. We'll look at some of the edge cases of the box model, and I'll give you practical advice for sizing and aligning elements.

3.1 Normal document flow

In this chapter, you'll build a simple page with a header at the top and content beneath it. The width content will be restricted to avoid very long lines of text. This sort of layout is often called a one-column layout, because there are no blocks of text side-by-side anywhere.

By the end of the chapter, your page will look like the one shown in figure 3.1. I've intentionally made the page design a bit "blocky," so you can readily see the size and position of the elements.

Figure 3.1 Example one-column page layout you'll build by the end of the chapter



Start a new page and an empty stylesheet. Add the markup shown in listing 3.1 to your page. Your page will have a header and a primary container that comprises the bulk of the page. Inside the container are a `<main>` with content and an `<aside>` with some social media links.

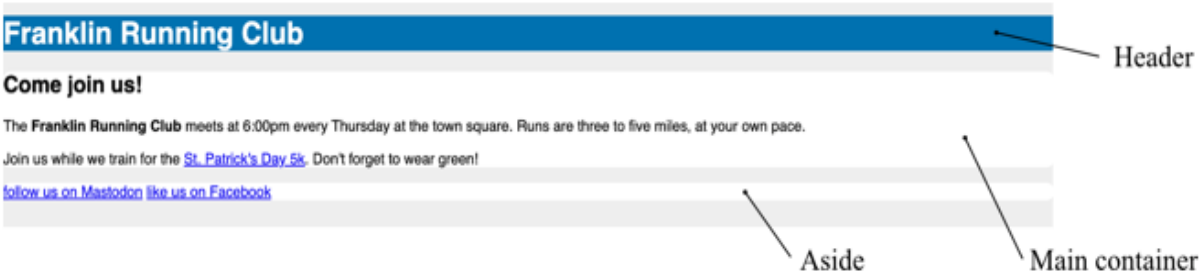
Listing 3.1 HTML for page with a one-column layout

```
<!doctype html>
<html lang="en-US">
<head>
  <link href="styles.css" rel="stylesheet" type="text/css" />
</head>
<body>
  <header class="page-header">
    <h1>Franklin Running Club</h1>
  </header>
  <div class="container">
    <main class="main">
      <h2>Come join us!</h2>
      <p>
        The <b>Franklin Running Club</b> meets at 6:00pm every
```

```
Thursday at the
        town square. Runs are three to five miles, at your own
pace.
      </p>
      <p>
        Join us while we train for the
        <a href="/st-patricks">St. Patrick's Day 5k</a>. Don't
forget to wear
        green!
      </p>
    </main>
    <aside class="social-links">
      <a href="/mastodon" class="button-link">follow us on
Mastodon</a>
      <a href="/facebook" class="button-link">like us on
Facebook</a>
    </aside>
  </div>
</body>
</html>
```

Let’s begin with some of the obvious styles. You’ll set the font for the page, then background colors for the page and each of the main containers. This will help you see the position and size of each as you go. After you do this, your page will look like the one shown in figure 3.2.

Figure 3.2 Three main containers with background colors



The styles for this are shown in listing 3.2. Add this code to your stylesheet.

Listing 3.2 Applying font and colors

```
:root {
  --brand-color: #0072b0;   #A
}
```

```
body {
  margin: unset;   #B
  background-color: #eee;
  font-family: Helvetica, Arial, sans-serif;
}

.page-header {
  color: #fff;
  background-color: var(--brand-color);
}

.main {
  background-color: #fff;
  border-radius: 0.5em;
}

.social-links {
  background-color: #fff;
  border-radius: 0.5em;
}
```

Before you begin adjusting the size or layout of these elements, it's worth taking note how the page's default layout behaves. There are two basic types of element: *inline* and *block*, each with a different behavior.

Inline elements in our page include the `<b>` around "Franklin Running Club" and the `<a>` around the links. These flow along with the text of the page, from left to right. Inline elements will line wrap if they reach the edge of their container.

Block elements (usually called "block-level" elements) appear on their own individual lines. They automatically fill the width of their container. They have a line break above and below them, even if styles are added to reduce their width. Elements such as `<p>`, `<div>`, and `<header>` are set to `display: block` by the user-agent stylesheet, giving them this behavior.

This behavior is called *normal document flow*.

Definition

Normal document flow refers to the default layout behavior of elements on the page. Inline elements flow along with the text of the page, from left to

right, line wrapping when they reach the edge of their container. Block-level elements fall on individual lines, with a line break above and below.

The important thing to note is that height and width are fundamentally different. Normal document flow is designed to work with a constrained width and an unlimited height. Contents fill the width of their container and then line wrap as necessary.

This means that the width of a parent element determines the width of its children, but for height, the opposite is true: the heights of children elements determine the height of the parent. As a result, we use different approaches for manipulating each. We'll look at a number of these approaches throughout the rest of the chapter.

3.1.1 Centering content horizontally

When building the layout of a page, you should do so from the outside in. That is, start with the higher-level DOM elements first. Get them sized and placed in relation to one another before focusing on their child elements. This gets a "rough cut" of the page in place and makes it easier to then focus in on the smaller details.

Tip

To begin laying out a page, it is best to do so from the outside in. Get the larger container elements where you want them before moving on to the smaller ones inside them.

On your page, you'll want to constrain the width of the page's main column. Because block-level elements fill the width of their container by default, you generally don't need to do anything like `width: 100%` or `width: 100svw` for full-width elements, but you will often want to reduce their width from this default.

Your page after constraining the width of the content is shown in figure 3.3. Notice the light gray margins on both sides and how both the header and the main container are equal widths within, aligning their content on the left side.

Figure 3.3 Page with constrained width

This layout is common for centering content on a page. You can achieve it by placing your content inside two nested containers and then set margins on the inner container to center it within the outer one (figure 3.4). Web developer Brad Westfall calls this the *double-container pattern*.

Figure 3.4 The double-container pattern horizontally centers a block of content

In your page, `<body>` serves as the outer container. By default, this is already 100% of the page width, so you won't have to apply any new styles to it. Inside that, you've wrapped the main contents of the page in a `<div class="container">`, which serves as the inner container. To that you'll apply a width and margins to center the contents.

You can also use the same pattern a second time in the header to keep it aligned with the content beneath. The `<header>` will serve as the outer container and the `<h1>` will be the inner container. Because this needs to be the same width, you can use a custom property to control the size of both in one place. Update your stylesheet to include the code shown in listing 3.3.

Listing 3.3 Styles for the double container

```
:root {
  --brand-color: #0072b0;
  --column-width: 1080px;
}
```

```
.page-header h1 {
  max-width: var(--column-width);  #A
  margin: 0 auto;           #B
}

.container {
  max-width: var(--column-width);  #A
  margin: 0 auto;           #B
}
```

By setting a left and right margin of `auto`, the margins will automatically expand as much as necessary to fill the remaining width available in the outer container. This is often the simplest way to center a block of content horizontally. It's important to note, however, that under normal document flow, this technique does not work for top and bottom margins.

Using `max-width` instead of width allows the element to shrink below 1080 px if the screen's viewport is narrower than that. That is to say, in smaller viewports, the inner container will fill the screen, but on larger ones, it'll expand to 1080 px. This is important to avoid horizontal scrolling on devices with smaller screens. We'll take a little closer look at `max-width` and related properties later in the chapter.

3.1.2 Using logical properties

In the past, the default behavior of document flow has made it difficult when working with websites that need to be translated into certain languages. Normal document flow goes from left to right, top to bottom. This is because most languages, including English, are written this way. But in order for the "world-wide" web to accommodate the whole world, it needs to also work for languages that are read in other ways. This includes right-to-left languages, such as Arabic and Hebrew, and vertically written languages, such as Japanese and traditional Chinese. For this reason, the W3C has done a lot of work to introduce the concept of *logical properties* to CSS.

Definition

Logical properties provide a way to work with elements in terms of their block and inline directions—which can change for different writing modes

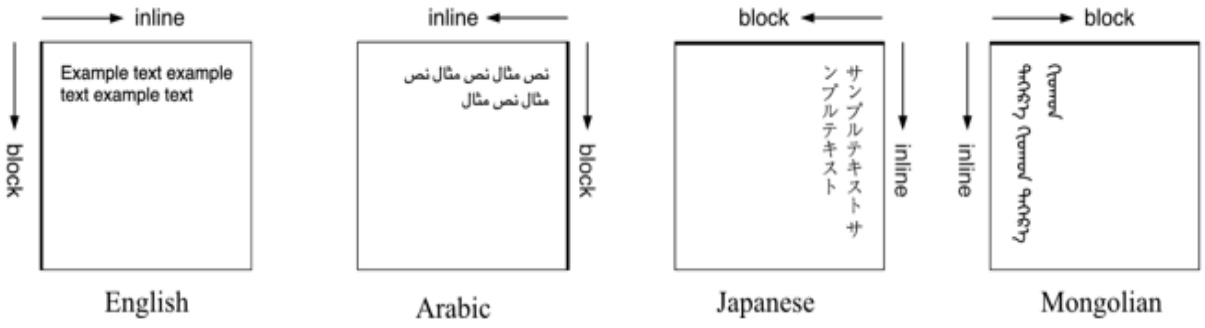
—rather than explicitly referring to top, right, bottom, and left or to width and height.

When using logical properties, we swap out the concepts of horizontal and vertical for *inline base direction* (the inline flow of text) and *block flow direction* (the direction in which boxes such as paragraphs stack). Instead of setting `width`, we can set `inline-size`; the two do exactly the same thing in a horizontal writing mode, but `inline-size` adapts to specify the height when used with vertical writing modes. And instead of `height`, we can use `block-size`, which adapts to specify a width in a vertical writing mode.

Logical properties also replace top, right, bottom, and left with *start* and *end*. Thus, `padding-left` and `padding-right` become `padding-inline-start` and `padding-inline-end`, respectively. Or `border-top` and `border-bottom` become `border-block-start` and `border-block-end`. These re-orient their meanings according to the writing mode.

Figure 3.5 shows the block and inline directions for various writing modes. Arrows point from the “start” to the “end” in each dimension. I’ve made one border for each example a little thicker using `border-inline-start` to highlight how it behaves.

Figure 3.5 Inline and block directions in various writing modes



Adapting to use logical properties is primarily a matter of becoming familiar with these new names. You don’t need to change how you’re laying out the page, just some of the terminology that you’ve become accustomed to. You shouldn’t have to concern yourself with multiple writing modes or languages until it actually comes to enacting a translation of your page content; if your

layout works in a familiar left-to-right language using logical properties, it will automatically change to work if the writing mode changes.

Do I need logical properties if I'm working in only one language?

Depending where you live and work, you may infrequently, if ever, need to develop sites that support other writing modes. Even then, I still think it's important to understand the principles of logical properties for three reasons:

First, they are a useful primer for concepts that are integral to flexbox and grid, which we will be diving into in the next two chapters. Second, there are a few extra logical properties that don't actually have a classic counterpart, and these can be more convenient to use at times. And third, they are becoming more widely used in the industry, and it's important to understand the code you encounter.

In all practicality, if you are working on a site that has no need for alternate writing modes, it is entirely up to you whether you use logical properties, classic properties, or some combination of the two. Even when working in these languages, sometimes you will need to use some of the classic properties if the style you're adding needs to work the same regardless of the writing direction.

If you've been inspecting elements with your browser's DevTools, you might have noticed its user-agent stylesheet is already using logical properties for default margins, the inline-start padding on lists, and other similar properties.

Both classic properties and their equivalent logical properties can override each other in the cascade. This way, if you set `padding-left` on a list, it will override the user-agent styles applying `padding-inline-start`. But you can also use `margin-block-start` with a higher specificity selector to override a `margin-top` applied with lower specificity. This means you can use both approaches interchangeably in your stylesheet and still have a predictable outcome.

Nearly every CSS property that deals with vertical or horizontal values has a logical counterpart. This way, you can define layouts that adapt based on the

needs of the language. Here are a number of examples of classic properties and values and the corresponding logical properties and values:

- `width / inline-size`
- `height / block-size`
- `margin-top / margin-block-start`
- `margin-bottom / margin-block-end`
- `margin-left / margin-inline-start`
- `margin-right / margin-inline-end`
- `text-align: left / text-align: start`
- `text-align: right / text-align: end`
- `border-top-left-radius / border-start-start-radius`
- `border-top-right-radius / border-start-end-radius`
- `border-bottom-left-radius / border-end-start-radius`
- `border-bottom-right-radius / border-end-end-radius`

This is only a partial list, but it serves to illustrate the pattern. By using block/inline and start/end, all the patterns you're already familiar with when using classic properties can be applied in the same way with logical properties. For a comprehensive list of logical properties, see https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Logical_Properties.

3.1.3 Adopting useful shorthand logical properties

Some logical properties happen to provide shorthand approaches to common patterns. For example, `margin-inline` allows you to set the start (left) and end (right) margin at once without setting the other two margins. You can do something like `margin-inline: 2rem` to set both start and end (left and right) margins to 2 rem, or `margin-inline: 2rem 4em` to set the start (left) margin to 2 rem and the end (right) margin to 4 em. The same approach is true for `margin-block` to set the block start and end (top and bottom) margins, as well as the other similar properties `padding-inline`, `padding-block`, `border-inline`, and `border-block`. There are no classic properties that behave quite like this.

You can use this on your page for a slightly cleaner approach to the double container pattern you've built. Update your stylesheet to this, as shown in

listing 3.4. This also includes changes replacing `max-width` with the logical property equivalent, `max-inline-size`.

Listing 3.4 Using logical properties for the double container pattern

```
.page-header h1 {
  max-inline-size: var(--column-width);  #A
  margin-inline: auto;           #B
}

.container {
  max-inline-size: var(--column-width);  #A
  margin-inline: auto;           #B
}
```

These changes don’t visually change anything on your page; it’s just an authoring convenience. I have found these shorthand properties useful, because I frequently want to specify only top and bottom or only left and right values for margins or other properties.

3.2 The box model

The next thing to address on the page you’re building is some padding in the main-container and the social-links box. Currently, the text in these areas is right up against the edges of the white background, so adding a little space there will make it look less crowded and more readable. Update your stylesheet to include the changes in listing 3.5.

Listing 3.5 Adding padding to the containers

```
.main {
  padding: 1em 1.5rem;    #A
  background-color: #fff;
  border-radius: 0.5em;
}

.social-links {
  padding: 1em 1.5rem;    #A
  background-color: #fff;
  border-radius: 0.5em;
}
```

Now the content inside two white containers has been made slightly narrower, leaving adequate space to breathe. However, by doing this, the left side of the text is no longer horizontally aligned with the text above in the page header (figure 3.6).

Figure 3.6 After adding padding, the text no longer all aligns on the left side

It might seem like we could fix this by adding a similar padding to the `<h1>` in the page-header. Add this as shown in listing 3.6, but you'll notice it doesn't seem to make a difference.

Listing 3.6 Adding padding to the page header

```
.page-header h1 {
  max-inline-size: var(--column-width);
  margin-inline: auto;
  padding-inline: 1.5rem;   #A
}
```

If you happen to be on a smaller screen (less than about 1100 px wide), this may look like it worked, but on wider screens, it will have made no visual difference at all. Even with a padding added, the content didn't get narrower as it did with the main content box.

This is because of the default behavior of the *box model*. According to the box model, each element on the page is made up of four overlapping rectangles. The *content area* is the innermost rectangle where the contents of the element reside. The *padding area* contains the content area plus any padding. Likewise, the *border area* is the padding area plus any border and

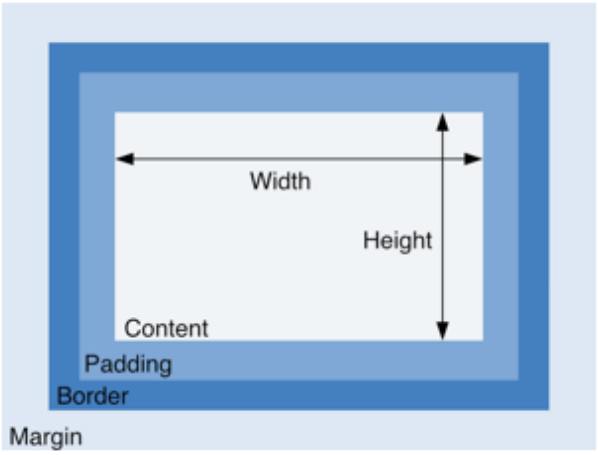
the *margin area* is the outermost rectangle, containing the border area plus any margins.

Definition

The box model refers to the parts of an element (content, padding, border, and margin) and the size they contribute to their element. These produce rectangular boxes that are laid out by the browser on the page.

Specifying a height or width of an element sets the size of its content area; any padding, border, and margins are added outside that (figure 3.7).

Figure 3.7 The default box model



This behavior means that an element with a 300-px width, a 10-px padding, and a 1-px border has a rendered width of 322 px (width plus left and right padding plus left and right border). This gets even more confusing when the units aren't all the same.

On your page, this means adding padding to the `<h1>`, which had a width of 1080 px, added to its effective width. The padding is outside that 1080px, and the main content stayed 1080px wide.

Note

Top and bottom margins and paddings behave a little unusually on inline elements. They will still increase the height of the element, but they will not increase the height that the inline element contributes to its container; that height is derived from the inline element's line-height. Using `display: inline-block` will change this behavior if you need.

Outline: the other type of border

Similar to a border, you can also add an `outline` to an element. This behaves much like a border, but does not add to the element's size and is not part of the box model. It is placed outside the border, overlapping the margin. It will not change the size or position of the element, nor will it affect the page layout in any way.

Like `border`, `outline` is a shorthand property for `outline-color`, `outline-style`, and `outline-width`. For example, `outline: orange solid 2px` will place a 2px-wide orange outline around an element. Unlike a border, you cannot specify a different outline for each side of the element; all four sides will always have the same outline style. Historically, an outline always had squared-off corners, but some browsers have recently changed their outline behavior to match the curve of any `border-radius` on the element.

You can manipulate the placement of an outline with the `outline-offset` property. A positive value (for example, `outline-offset: 3px`) will expand the position of the outline outward in all directions, adding space between the element's border area and the outline. A negative value will bring the outline inward, causing it to overlap with the element's border area.Understanding the box model is an essential part of working with CSS. The way padding and border can increase an element's size can catch you off guard if you aren't ready for it. Knowing why this happens is an important first step in adapting your styles to accommodate it.

3.2.1 Avoid magic numbers

Sometimes when you encounter problems like this, the temptation can be to fiddle with the values until it works. This is especially the case when you happen to use percent to define sizes.

Imagine if, instead of 1080 px width, your layout used 70%. A naive fix might be to reduce that percentage for the `<h1>`. Perhaps a width of 66% seems to work—but this is unreliable. The 66% is known as a *magic number.* Instead of using a desired value, you found it by making haphazard changes to the styles until you got the result you want.

For programming in general, magic numbers aren't desirable. It's often hard to explain why a magic number works. If you don't understand where the number comes from, you can't foresee how it will behave under different circumstances. Maybe it works with a viewport 1400-px wide, but on larger or smaller screens, the text no longer aligns. Although there's a place for trial and error in CSS, typically that's for choices that are stylistic in nature and not for forcing things to fit into position.

One alternative to this magic number is to let the browser do the math. In your case, the `<h1>` is 3 em too wide (due to the left and right padding), so you can use the `calc()` function to reduce the width by exactly that much. A width of `calc(var(--columns-width) - 3em)` gives you exactly what you need. But there's still a better way.

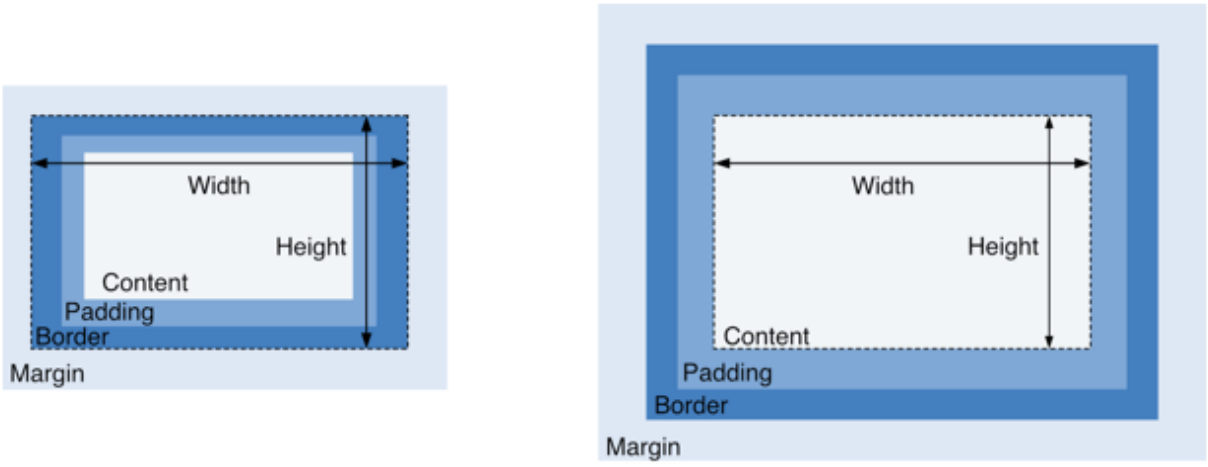
3.2.2 Adjust the box model

The default box model tends to cause problems with sizing and alignment of elements on the page. Instead, you'll want your specified widths to include the padding and borders. CSS allows you to adjust the box model behavior with its `box-sizing` property.

By default, `box-sizing` is set to the value of `content-box`. This means that any height or width you specify sets the size of only the content box. You can assign a value of `border-box` to the box sizing instead. That way, the `width`, `height`, `inline-size`, and `block-size` properties set the combined size of the content, padding, and border, which is exactly what you want in this example.

Figure 3.8 shows the box model with box sizing set to `border-box`. With this model, padding doesn't make an element wider; it makes the inner content narrower. It also does the same for height.

Figure 3.8 Border-box sizing changes the box model so width and height are more predictable



Border box sizing

Content box sizing

If you update the `<h1>` to use border box sizing, its content will align with the content below (figure 3.9).

Figure 3.9 Left edge of content aligns with border box sizing

To adjust the box model for your heading, update your stylesheet to match this listing.

Listing 3.7 Heading with a corrected box model

```
.page-header h1 {
  box-sizing: border-box;    #A
  max-inline-size: var(--column-width);
```

```
  margin-inline: auto;
  padding-inline: 1.5rem;
}
```

Using `box-sizing: border-box`, the padding is now counted within the 1080 px width. The heading text now aligns with the main content beneath it.

3.2.3 Use universal border-box sizing

You have made box sizing more intuitive for this one element, but you'll surely run into other elements with the same problem. It would be nice to fix it once, universally for all elements, so you won’t have to think about this adjustment again. You can do this with the universal selector (`*`), which targets all elements on the page as the following listing shows. I’ve added selectors to target every pseudo-element on the page as well. Put this code at the top of your stylesheet.

Listing 3.8 Universal border-box fix

```
*,
::before,
::after {
  box-sizing: border-box;    #A
}
```

After applying this to the page, `height` and `width` will always specify the actual height and width of an element. Padding won’t change them.

Now, every element on your site will have a more predictable box model. I recommend that you add listing 3.8 to your CSS every time you start a new site; it’ll save you a lot of trouble in the long run. It can be a little problematic in an existing stylesheet, however, especially if you’ve already written lots of styles based on the default content-box model. If you do add this to an existing project, be sure to give it a thorough review for any resulting bugs.

Note

Adding this snippet near the beginning of your stylesheet has become common practice. From this point on, every example in this book will assume that this `border-box` fix is at the beginning of your stylesheet.

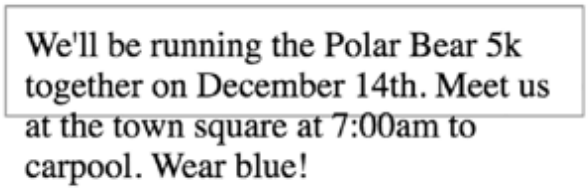
3.3 Element height

Working with the height (block-size) of elements can be tricky. Normal document flow is designed to work with a constrained width and an unlimited height. The height of a container is organically determined by its content, not the container itself. Typically, it's best to avoid setting explicit height on elements—particularly elements with a lot of content—though there is often a temptation to do so.

3.3.1 Controlling overflow behavior

When you explicitly set an element's height, you run the risk of its contents *overflowing* the container. This happens when the content doesn't fit the specified constraint and renders outside the parent element. Figure 3.10 shows this behavior. Document flow doesn't account for overflow, and any content after the container will render over the top of the overflowing content.

Figure 3.10 Content overflowing its container



You can control the exact behavior of the overflowing content with the `overflow` property, which supports five values:

- `visible` (default value)—All content is visible, even when it overflows the edges of the content area.

- `hidden`—Content that overflows the container's padding area is clipped and won't be visible. The user will not be able to scroll to this content normally, though the element can still be scrolled programmatically via JavaScript.
- `clip`—Behaves the same as `hidden`, but programmatic scrolling is also disabled.
- `scroll`—Scrollbars are added to the container so the user can scroll to see the remaining content. On some operating systems, both horizontal and vertical scrollbars are added, even if all the content is visible. In this case, the scrollbars will be disabled (grayed).
- `auto`—Scrollbars are added to the container only if the contents overflow.

Figure 3.11 illustrates four containers with various overflow settings.

Figure 3.11 Overflow from left to right: `visible`, `hidden`, `scroll`, and `auto`

Typically, I prefer to use `auto` rather than `scroll` because, in most cases, I don't want the scrollbars to appear except when necessary.

Horizontal overflow

It's possible for content to overflow horizontally, not just vertically. This can occur when a long URL appears in a narrow container, or when viewing a table with a lot of columns on a small mobile device. The same rules apply here as with vertical overflow.

You can control only horizontal overflow using the `overflow-x` property, or vertical overflow with `overflow-y`. These properties support the same values as the `overflow` property. Explicitly setting both `x` and `y` to different values, however, tends to have unpredictable results.

Be judicious with the use of scrollbars. Browsers insert a scrollbar for scrolling the page, and adding nested scrollable areas inside your page can

be frustrating to users. If a user is using a mouse scroll wheel to scroll down the page, and their cursor reaches a smaller scrollable area, their scroll wheel will stop scrolling the page and will scroll the smaller box instead.

3.3.2 Using alternatives to percentage-based heights

Specifying height using a percentage can be problematic. Percentage refers to the size of an element's containing block; the height of that container, however, is typically determined by the height of its children. This produces a circular definition that the browser can't resolve, so it will ignore the declaration. For percentage-based heights to work, the parent must have an explicitly defined height.

One place where percentage-based heights can be useful is when used for absolutely positioned elements, when you want to size them in relation to their container. We will take a look at positioning in a later chapter.

There are two use cases in particular where developers often try to use percentage heights that lead them into trouble.

The first is when trying to make a container fill the screen. This can produce a scenario where you need to put `height: 100%` on an element, and its container, and so on up the DOM tree until you've applied it to both `<body>` and `<html>`. A better approach is to use the viewport-relative units, which you reviewed in chapter 2, since a height of 100 svh/lvh is generally what you're looking for in this case.

The second reason is to create columns of equal height. Page designs often call for two or more side-by-side elements to have the same height. It was historically a tricky thing to size one element in CSS in relation to a sibling element. There were some very complicated tricks to make it work, but this can now be done much more easily with more modern techniques. We'll see how in the following chapters on flexbox and grid.

3.3.3 Using min-height and max-height

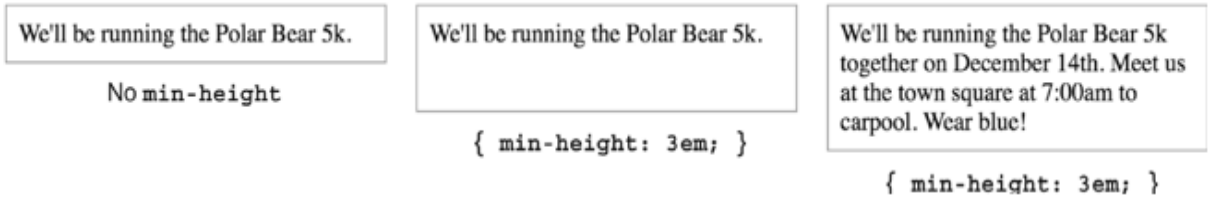
Two properties that can be immensely helpful are `min-height` and `max-height` (and their logical counterparts `min-block-size` and `max-block-`

size). Instead of explicitly defining a height, you can use these properties to specify a minimum or maximum value, allowing the element to size naturally within those bounds.

Suppose you want to place your hero image behind a larger paragraph of text, and you're concerned about it overflowing the container. Instead of setting an explicit height, you can specify a minimum height with `min-height`. This means the element will be at least as high as you specify, and if the content doesn't fit, the browser will allow the element to grow naturally to prevent overflow.

Figure 3.12 shows three elements. The element on the left has no `min-height`, so its height is determined naturally, while each of the other two has a `min-height` of 3 em. The element in the middle would have a natural height shorter than that, but the `min-height` value has brought it to a height of 3 em. The element on the right has enough content that it has exceeded 3 em, and the container has grown naturally to contain the content.

Figure 3.12 Three elements: one with no specified height, and two elements with a 3 em `min-height`



Most of the time I feel like I want to set the height on a container, I use `min-height` instead. It often saves me the headache of sorting out overflow problems later on.

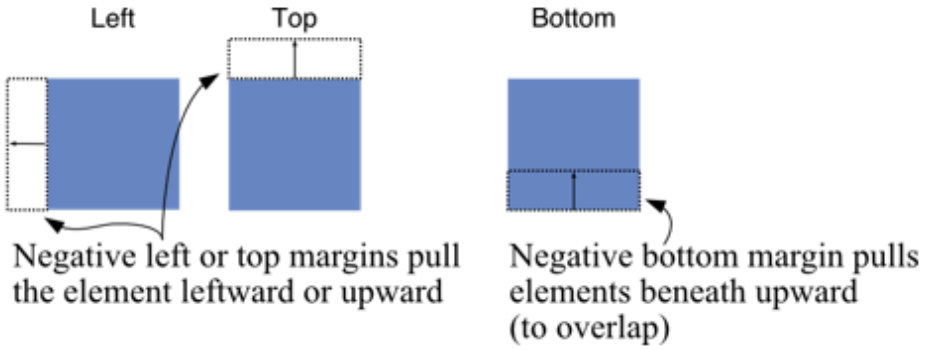
The corresponding `max-height` property allows an element to size naturally, up to a point. If that size is reached, the element doesn't become any taller, and the contents will overflow. Similar properties `min-width` and `max-width` (and `min-inline-size` and `max-inline-size`) constrain an element's width.

3.4 Negative margins

Unlike padding and border width, you can assign a negative value to margins. This has some peculiar uses, such as allowing elements to overlap or stretch wider than their containers.

The exact behavior of a negative margin depends on which side of the element you apply it to. You can see this illustrated in figure 3.13. If applied to the left or top, the negative margin moves the element leftward or upward, respectively. This can cause the element to overlap another element preceding it in the document flow. If applied to the bottom side, a negative margin doesn't shift the element; instead, it pulls up any elements beneath it; this is not unlike giving the elements directly beneath it a negative top margin.

Figure 3.13 Behavior of negative margins



Although negative margins can be used to cause an overlap between multiple elements, it can be tricky to keep track of anything complicated. Often, it's a better idea to accomplish this using the `position` property instead; we'll look at positioning in detail in chapter 6.

Warning

Using negative margins to overlap elements can render interactive elements unclickable if they're moved behind other elements.

Negative right margins behave differently depending whether the element is inline or block-level. On inline elements, they behave similar to negative

bottom margins, pulling any successive content to the left so it overlaps the element. In practice, I have never had to use this.

On a block-level element, a negative right margin pulls the edge of the element right, making the element wider. This possibly brings the edge of the element outside its container.

If you join this with a negative left margin, both sides of the element will be extended outside the container, making it wider than its container. This is a sort of an alternate application of the double container pattern, illustrated in figure 3.14.

Figure 3.14 Negative left and right margins can extend a block-level element outside its container.

The code for this sort of approach is shown in listing 3.9. This can be used to "bleed" an image wider than the column of text it appears with.

Listing 3.9 Using negative margins to bleed an image outside its container

```
.container {
  max-width: 1080px;
  margin-inline: auto;
}

.expanded-child {
  margin-inline: -2em;   #A
}
```

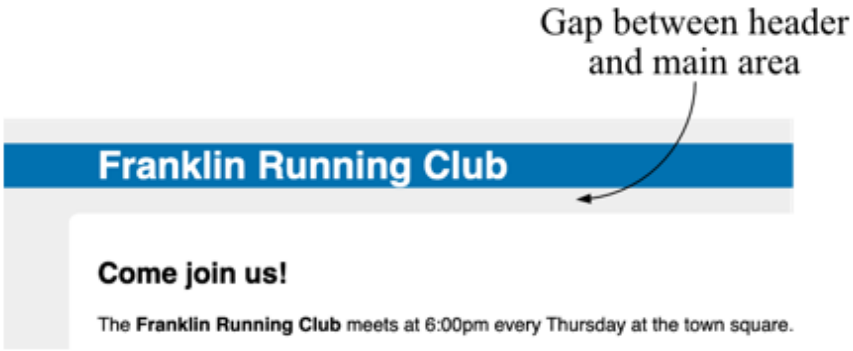
Be aware that this can cause content to extend outside the browser viewport. Take care that things still work as expected on small screens when using this

technique.

3.5 Collapsed margins

Let’s continue building your page. Notice something strange going on with the margins? You haven’t applied any margin to the header or the container, yet there’s a gap between them (figure 3.15). Why is that gap there?

Figure 3.15 Gap caused by the margins collapsing



When top and/or bottom margins are adjoining, they overlap, combining to form a single margin. This is referred to as *collapsing*. The space below the header in figure 3.16 is the result of collapsed margins. Let’s look at how this works.

3.5.1 Collapsing between text

The main reason for collapsed margins has to do with the spacing of blocks of text. Paragraphs (`<p>`), by default, have a 1 em top margin and a 1 em bottom margin. This is applied by the user agent stylesheet. But when you stack two paragraphs, one after the other, their margins don’t add up to a gap of 2 em. Instead they collapse, overlapping to produce only 1 em of space between the two paragraphs.

You can see this sort of collapsed margin inside the `<main>` section on your page. The title (Come join us!) in an `<h2>` has a bottom margin of 0.83 em, which collapses with the top margin of the following paragraph. The

margins of each are illustrated in figure 3.16. Note how the margins of each element occupy the same space on the page.

Figure 3.16 Outlined margins of the heading (left) and paragraph (right)

The size of the collapsed margin is equal to the largest of the joined margins. In this case, the heading has a bottom margin of 19.92 px (24-px font size × 0.83 em), and the paragraph has a top margin of 16 px (16-px font size × 1-em margin). The larger of these, 19.92 px, is the amount of space rendered between the two elements.

3.5.2 Collapsing multiple margins

Elements don’t have to be adjacent siblings for their margins to collapse. Even if you wrap the paragraph inside an extra `div`, as in the next listing, the visual result will be the same. In the absence of any other CSS interfering, all the adjacent top and bottom margins will collapse.

Listing 3.10 Paragraph wrapped in a `div`, with the same result

```
<main class="main">
  <h2>Come join us!</h2>
  <div>                                                #A
    <p>                                                #A
      The Franklin Running club meets at 6:00pm        #A
      every Thursday at the town square. Runs
      are three to five miles, at your own pace.
    </p>
  </div>
</main>
```

In this case, there are three different margins collapsing together: the bottom margin of the `<h2>`, the top margin of the `<div>`, and the top margin of the

<p>. The computed values of these are 19.92 px, 0 px, and 16 px, respectively, so the space between the elements is still 19.92 px, the largest of the three. In fact, you can nest the paragraph inside several divs, and it will still render the same—all the margins collapse together.

In short, any adjacent top and bottom margins will collapse together. If you add an empty, unstyled div (one with no height, border, or padding) to the page, its own top and bottom margins will collapse.

Note

Margin collapsing occurs with only top and bottom margins. Left and right margins don’t collapse. (In a horizontal writing mode, this is reversed, and left/right margins will collapse instead.)

Collapsed margins act as a sort of “personal space bubble.” If two people standing at a bus stop are each comfortable with 3 feet of personal space between, they’ll happily stand 3 feet apart. They don’t need to stand 6 feet apart to both be satisfied.

This behavior typically means you can style margins on various elements without much concern for what might appear above or below them. If you apply a bottom margin of 1.5 em to the headings, you can expect the same spacing following the headings, whether the next element is a <p> with a top margin of 1 em or a with no top margin. The collapsed margin between the elements appears larger only if the following element requires more space.

3.5.3 Collapsing outside a container

The way three consecutive margins collapse might catch you off guard. An element's margin collapsing outside its container typically produces an undesirable effect if the container has a background.

This is what’s causing the gap below the header in your page. The page title is an <h1>, with a 0.67 em (21.44 px) bottom margin applied by the user agent styles. That title is inside a <header> with no margins. The bottom margins of both elements are adjacent, so they collapse, resulting in a 21.44-

px bottom margin on the header. The same thing happens with the top margins of the two elements as well.

This is a little strange. In this case, you want the blue background of the `<header>` to be taller, so there's some space around the heading. Margins don't always collapse exactly to the spot where you want. Fortunately, there are a number of ways to prevent this. In fact, you've already fixed it for the main section of the page; notice that the margin above "Come join us!" doesn't collapse upward outside of its container. That's because you've applied a padding to the container, and margins won't collapse if there's any padding between them.

If you add top and bottom padding to the header, the margins inside it won't collapse to the outside. This means both the margins of the `<h1>` and the padding of the `<header>` would both contribute space around the text, which might be a bit much. So it's probably best to remove the margin altogether and allow the padding to define the space you need.

Figure 3.17 Adding padding to the header prevents margin collapsing

Update your stylesheet to match listing 3.14. You'll notice this now means there's no longer any space between the header and the main content. We'll come back to address that shortly.

Listing 3.11 Applying padding to the header

```
.page-header h1 {
  max-inline-size: var(--column-width);
  margin: 0 auto;     #A
  padding: 1em 1.5rem;   #B
}
```

Any time you see unexpected spaces above or below containers, or you see text pressed up against the top or bottom of its container, margin collapsing is most likely the cause.

Here are ways to prevent margins from collapsing:

- Applying `overflow: auto` (or any value other than `visible`) to the container prevents margins inside the container from collapsing with those outside the container. This is often the least intrusive solution.
- Adding a border or padding between two margins stops them from collapsing.
- Margins won't collapse to the outside of a container that is an inline-block, that is floated (chapter 12), or that has an absolute or fixed position (chapter 6).
- When using a flexbox or grid layout, margins won't collapse between elements that are part of the flex layout (chapters 4 and 5).
- Elements with a `table-cell` display don't have a margin, so they won't collapse. This also applies to `table-row` and most other table display types. Exceptions are `table`, `table-inline`, and `table-caption`.

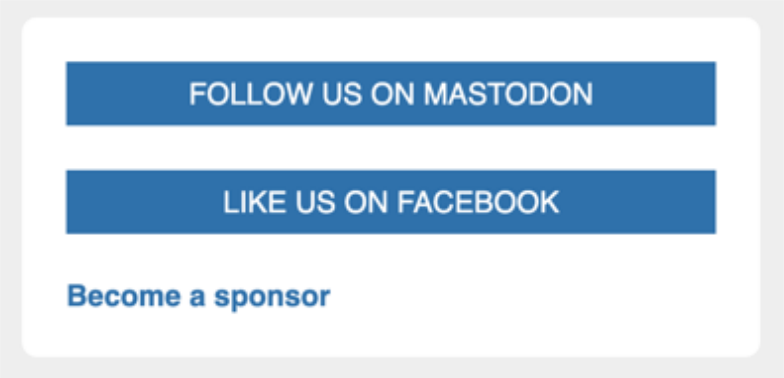
Many of these change the layout behavior of the element, though, so you probably won't want to apply them unless they produce the layout you're looking for.

3.6 Spacing elements within a container

The interplay between the padding of a container and the margins of its content can be tricky to work with. Let's add some styles to the social links at the end of your page and work through problems that might arise.

You'll style the two links to social media pages and another, less important link. Your goal is for this section to look like figure 3.18.

Figure 3.18 The social links with properly spaced contents



Let’s start with the two social links. They already have a `button-link` class that will be a good target for your CSS selector.

You’ll apply styles for the links’ general appearance. You’ll make them block elements so they’ll fill the width of the container, and each will appear on its own line. Add this CSS to your stylesheet:

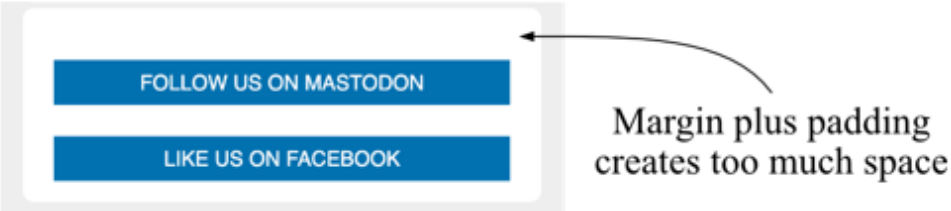
Listing 3.12 Setting size, fonts, and colors for the sidebar buttons

```
.social-links {
  max-inline-size: 25em;   #A
  padding: 1em 1.5rem;
  background-color: #fff;
  border-radius: 0.5em;
}

.button-link {
  display: block;     #B
  padding: 0.5em;
  color: #fff;
  background-color: var(--brand-color);
  text-align: center;
  text-decoration: none;
  text-transform: uppercase;
}
```

Now the links are styled correctly, but you still need to figure out the spacing between them. Without margins, they’ll stack directly atop one another, as they do now. You have options: you could give them separate top and bottom margins or both, where margin collapsing would occur between the two buttons.

No matter which approach you choose, however, you'll still encounter a problem: the margin needs to work in conjunction with the container's padding. If you add `margin-top: 1.5em` to both links, you'll get the result shown in figure 3.19.

Figure 3.19 The top margin adds spacing to the container's padding.



Now you'll have extra space at the top of the container. The first link's top margin plus the container's top padding produce spacing that's uneven with the other three sides of the container.

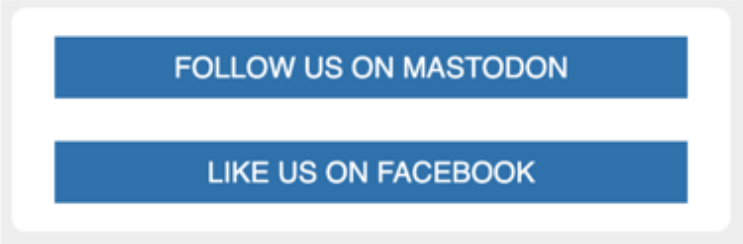
You can fix this in a number of ways. Listing 3.13 shows one of the simpler fixes. It uses the adjacent sibling combinator (+) to target only `button-links` that immediately follow other `button-links` as siblings under the same parent element. Now the margin only appears between two buttons.

Listing 3.13 Using an adjacent sibling combinator to apply a margin between buttons

```
.button-link {
  display: block;
  padding: 0.5em;
  color: #fff;
  background-color: var(--brand-color);
  text-align: center;
  text-decoration: none;
  text-transform: uppercase;
}
.button-link + .button-link {   #A
  margin-block-start: 1.5em;    #A
}                               #A
```

This appears to work (figure 3.20). The first button no longer has a top margin, so the spacing is even.

Figure 3.20 Even spacing is applied around the buttons.



Using the + combinator like this is a useful pattern that you can use in a variety of scenarios to space a series of elements within a container. The same approach also works to apply a left margin between a series of inline or inline-block elements.

3.6.1 Considering changing content

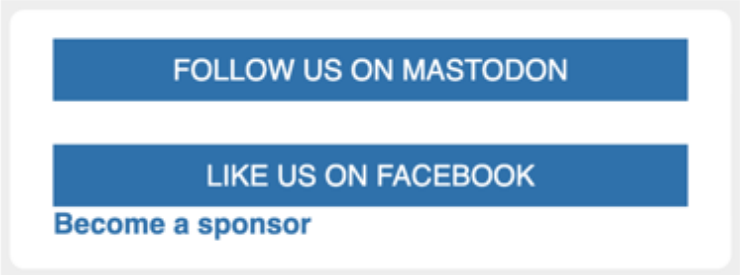
You're on the right track, but the spacing problem arises again as soon as you add more content to the sidebar. Add the third link to your page, as shown in the following listing. This one has the class `sponsor-link` so you can apply different styles to the link.

Listing 3.14 Adding a different type of link to the sidebar

```
<aside class="sidebar">
  <a href="/mastodon" class="button-link">
    Follow us on Mastodon
  </a>
  <a href="/facebook" class="button-link">
    Like us on Facebook
  </a>
  <a href="/sponsors" class="sponsor-link">    #A
    Become a sponsor                          #A
  </a>                                        #A
</aside>
```

You'll style this one, but again, you'll have to address the question of the spacing between it and the other buttons. Figure 3.21 shows how the link will look *before* you fix the margin.

Figure 3.21 Spacing is off between the second button and the bottom link.



The styles for this are shown in the next listing. Add these to your stylesheet. You're probably tempted to add a top margin to this link as well; hold off on that for now. I'll show you an interesting alternative next.

Listing 3.15 Adding styles for the sponsor link

```
.sponsor-link {
  display: block;
  color: var(--brand-color);
  font-weight: bold;
  text-decoration: none;
}
```

You could add a top margin, and it would look right. But consider this: HTML has a nasty habit of changing. At some point, whether next month or next year, something in this sidebar will need to be moved or replaced. Maybe the sponsorship link will need to be moved to the top of the sidebar. Or, maybe you'll need to add a widget to sign up for an email newsletter.

Every time things change, you'll have to revisit the question of these margins. You'll need to make sure that there's space between each item, but no extraneous space at the top (or bottom) of the container.

3.6.2 Creating a more general solution

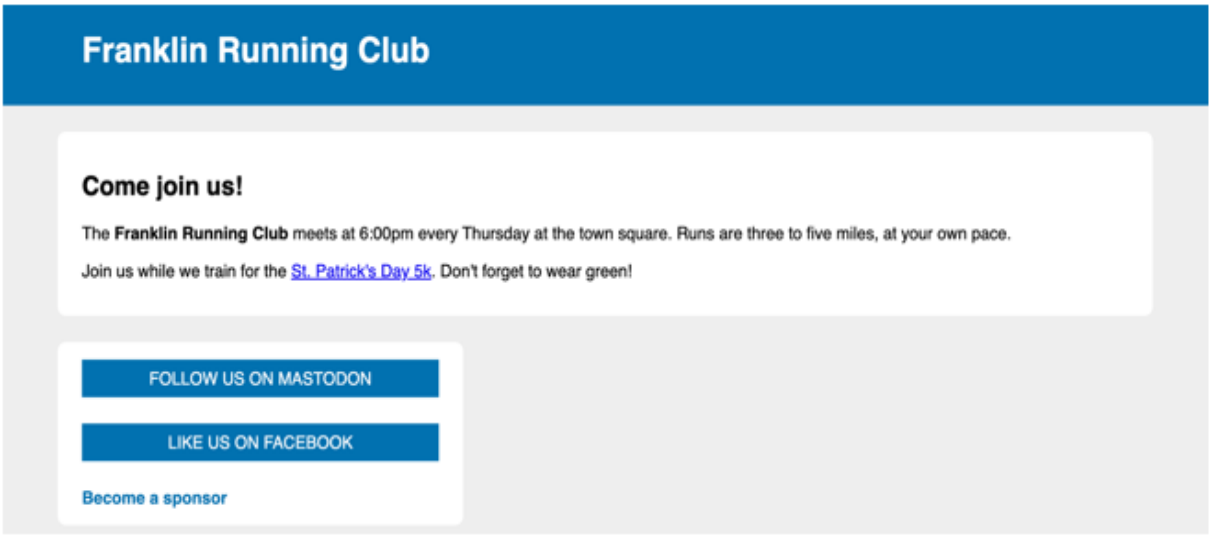
Web designer Heydon Pickering once said margins are "like applying glue to one side of an object before you've determined whether you actually want to stick it to something or what that something might be." Instead of fixing margins for the current page contents, let's fix it in a way that works no matter how the page gets restructured. You'll do this with something Pickering calls a *lobotomized owl selector*. It looks like this: `* + *`. He wrote

an article explaining this selector at https://alistapart.com/article/axiomatic-css-and-lobotomized-owls/.

That's a universal selector (`*`) that targets all elements, followed by an adjacent sibling combinator (+), followed by another universal selector. It earns its name because it resembles the vacant stare of an owl. The lobotomized owl is not unlike the selector you used earlier: `.social-button + .social-button`. Except, instead of targeting buttons that immediately follow other buttons, it targets any element that immediately follows any other element. That is, it selects all elements that aren't the first child of their parent. (Alternatively, you could use the selector `:not(:first-child)`, which functions essentially the same.)

I'll show you how to use the lobotomized owl to add top margins to elements throughout your page. You'll combine it with a new class name, "stack," that you can add to any container where you want to vertically space all child elements. The result is shown in figure 3.22.

Figure 3.22 All elements are neatly spaced



Add listing 3.16 to your stylesheet. This uses another combinator, the child combinator (>) to limit the lobotomized owl to only direct descendants of the stack. This little bit of code can do a lot of work on your page.

Listing 3.16 Create a "stack" to vertically space elements in a container

```
.stack > * + * {           #A
  margin-block-start: 1.5em;
}
```

Then, you can add the `stack` class to any containers where you want to space out child elements. First add it to the `<body>` (to add space between the `<header>` and `<div class="container">`). Then, add it to `<div class="container">`, to add space between the main content and the social links container. Finally, add it to the social links container, to space out all three links within.

Note

You might be worried about the performance implications of the universal selector (`*`). In IE6, it was incredibly slow, so developers avoided using it. Today, this is no longer a concern because modern browsers handle it well. Furthermore, using it with a reusable class like this reduces the number of selectors in your stylesheet, because it fixes multiple elements.

With this spacing added, your page is complete. The full stylesheet is given in listing 3.17.

Listing 3.17 Final stylesheet

```
*,
*::before,
*::after {
  box-sizing: border-box;
}

:root {
  --brand-color: #0072b0;
  --column-width: 1080px;
}

body {
  margin: unset;
  background-color: #eee;
  font-family: Helvetica, Arial, sans-serif;
}

.page-header {
  color: #fff;
```

```
  background-color: var(--brand-color);
}

.page-header h1 {
  max-inline-size: var(--column-width);
  margin: 0 auto;
  padding: 1em 1.5rem;
}

.container {
  max-inline-size: var(--column-width);
  margin-inline: auto;
}

.main {
  padding: 1em 1.5rem;
  background-color: #fff;
  border-radius: 0.5em;
}

.social-links {
  max-inline-size: 25em;
  padding: 1em 1.5rem;
  background-color: #fff;
  border-radius: 0.5em;
}

.button-link {
  display: block;
  padding: 0.5em;
  color: #fff;
  background-color: var(--brand-color);
  text-align: center;
  text-decoration: none;
  text-transform: uppercase;
}

.sponsor-link {
  display: block;
  color: var(--brand-color);
  font-weight: bold;
  text-decoration: none;
}

.stack > * + * {
  margin-block-start: 1.5em;
}
```

3.7 Summary

- In normal document flow, inline elements are placed side-by-side, line wrapping as necessary. Block-level elements are each placed on their own line, filling the width of their container by default.
- Logical properties refer to the sides or sizes of an element in terms of the document flow, rather than their explicit cardinal directions.
- The box model describes the way margin, border, padding, and content define the size of an element. Using `box-sizing: border-box` makes the behavior of this sizing more intuitive.
- The height of elements is determined organically based on their contents. Explicitly setting a height can lead to overflow issues.
- Margins can collapse with other margins outside a container, leading to undesirable spacing between elements. Appling `overflow: auto` is one way to prevent this behavior.
- The “lobotomized owl” selector is a useful tool for adding space between block-level elements.

4 Flexbox

This chapter covers

- Laying out elements of a page using flexbox
- Flex containers and flex items
- Main axis and cross axis
- Element sizing in flexbox
- Element alignment in flexbox

In chapter 3, we looked at normal document flow—the way the browser lays out elements on the page by default. In the next few chapters, we'll explore the other forms of layout available to you, particularly flexbox, grid, and positioning. With these tools, you will have far more options available; you will be able to structure pages almost any way imaginable.

Flexbox—formally Flexible Box Layout—is a method for laying out elements on the page. It's primarily used for arranging elements in a row or column. It also provides a simple solution to the historically troublesome problems of vertical centering and equal height columns.

If flexbox has one weakness, it's the overwhelming number of options it provides. It introduces more than a dozen new properties to CSS, including some shorthand properties. These can be a lot to take in at once. When I first started learning flexbox, it felt a bit like drinking from a fire hose, and I had a hard time committing all the new properties to memory. I'm going to ease into the topic by covering the most essential of these properties first.

I'll show you the basic principles of the flexbox layout that you'll need to understand, followed by practical examples. You don't need to learn all the new properties in order to use flexbox. I've found that only a few of them do most of the heavy lifting. The rest of the properties provide options for aligning and spacing elements. Near the end of the chapter, I'll explain those and provide a reference guide that you can return to when you need it.

4.1 Flexbox principles

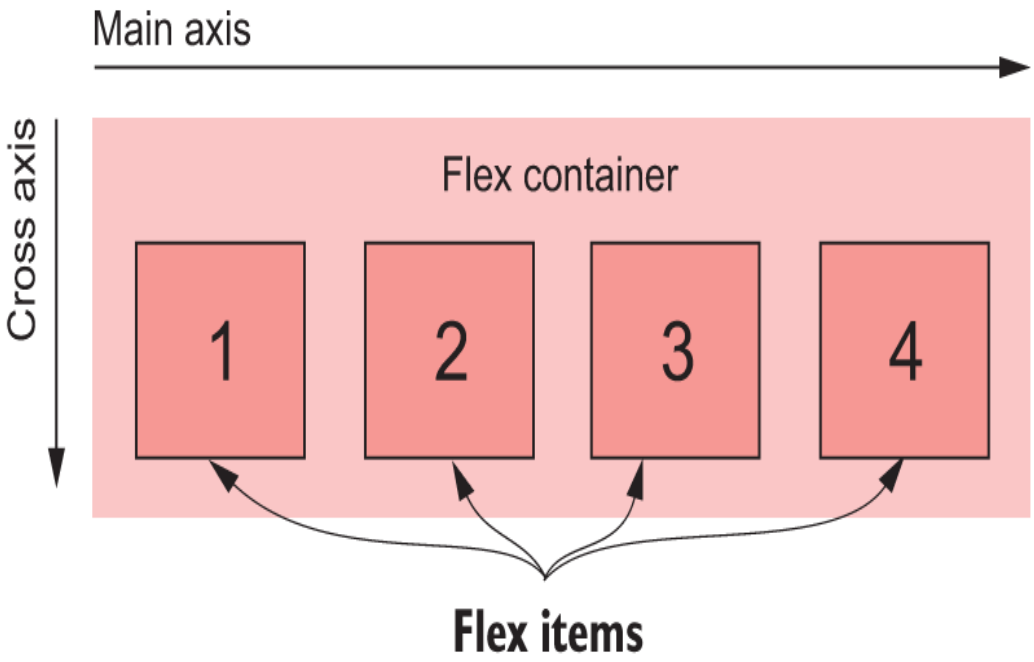
Flexbox begins with the familiar `display` property. Applying `display: flex` to an element turns it into a *flex container*, and its direct children turn into *flex items*. By default, flex items align side by side, left to right, all in one row. The flex container fills the available width like a block element, but the flex items may not necessarily fill the width of their flex container. The flex items are all the same height, determined naturally by their contents.

Tip

You can also use `display: inline-flex`. This creates a flex container that behaves more like an inline-block element rather than a block. It flows inline with other inline elements, but it won't automatically grow to 100% width. Flex items within it generally behave the same as with `display: flex`. Practically speaking, you won't need to use this very often.

Flexbox is unlike previous display values (inline, inline-block, and so on), which affect only the elements they are applied to. Instead, a flex container asserts control over the layout of the elements within. A flex container and its items are illustrated in figure 4.1.

Figure 4.1 A flexbox container and its elements

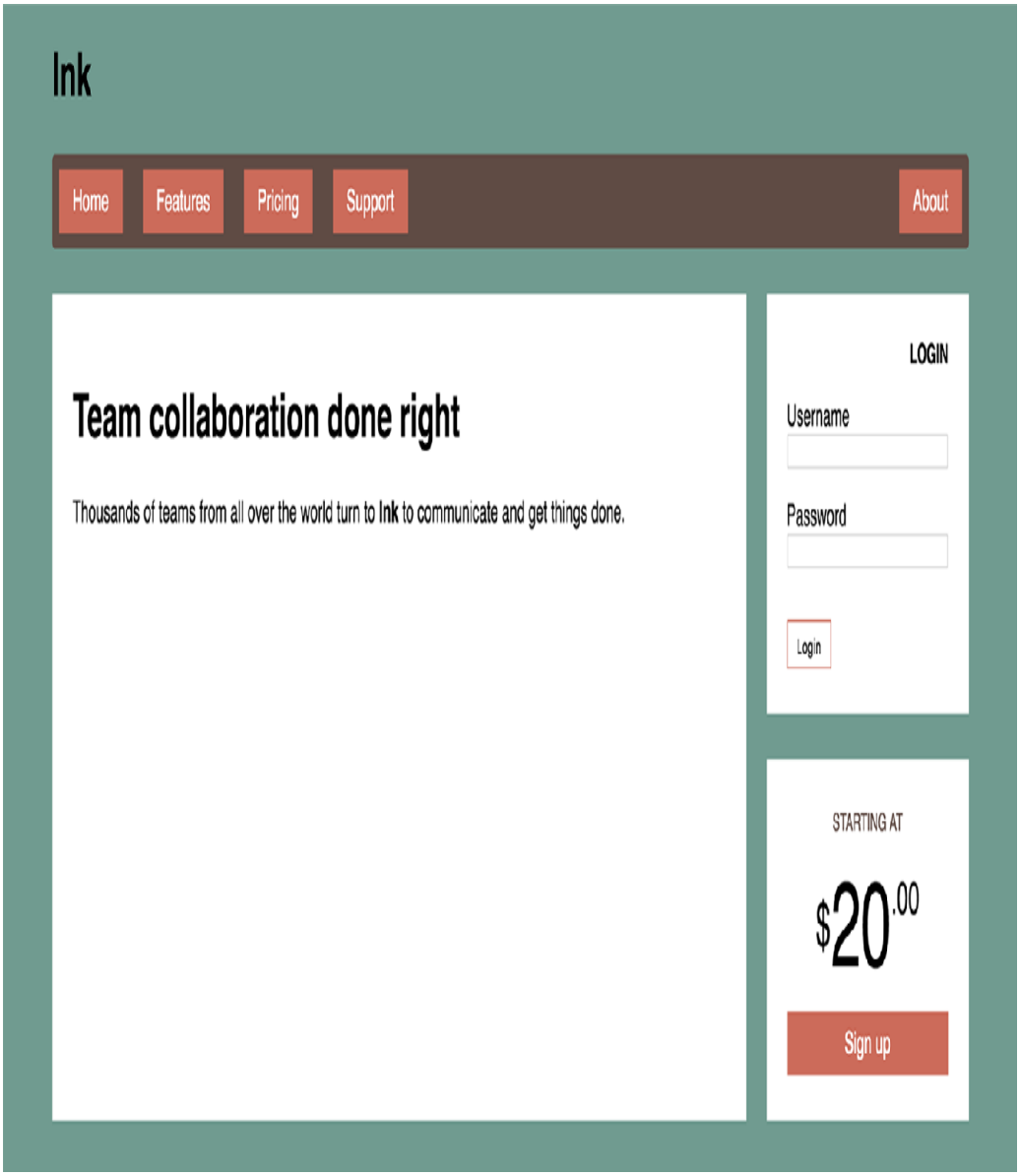


The items are placed along a line called the *main axis*, which goes from the *main start* (left) to the *main end* (right). Perpendicular to the main axis is the *cross axis*. This goes from the *cross start* (top) to the *cross end* (bottom).

Conceptually, the main axis and cross axis are similar to inline and block directions. As you'll see shortly, the flexbox properties make consistent use of the "start" and "end" terminology. The direction of these axes can be changed, for instances where you want elements to flow in another direction. I'll show you how to do this later in the chapter.

These concepts (flex container, flex items, and the two axes) cover a lot of what you need to know about flexbox. Applying `display: flex` gets you pretty far before you'll need to pick up any of those new properties. To try it out, you'll build the page shown in figure 4.2.

Figure 4.2 Finished page with a flexbox layout



I've structured this page to illustrate a number of ways to use flexbox. We'll use flexbox for the navigation menu across the top, to lay out the three white boxes, and for the stylistic "$20.00" on the bottom right.

Start a new page and link it to a new stylesheet, as shown in listing 4.1. Add this markup to your page.

Listing 4.1 Markup for the page

```
<!doctype html>
<html lang="en-US">
<head>
  <title>Flexbox example page</title>
  <link href="styles.css" rel="stylesheet"
```

```
    type="text/css" />
</head>
<body>
  <div class="container">
    <header>
      <h1>Ink</h1>
    </header>
    <nav>
      <ul class="site-nav">                              #A
        <li><a href="/">Home</a></li>                    #A
        <li><a href="/features">Features</a></li>        #A
        <li><a href="/pricing">Pricing</a></li>          #A
        <li><a href="/support">Support</a></li>          #A
        <li class="nav-right">                           #A
          <a href="/about">About</a>                     #A
        </li>                                            #A
      </ul>                                              #A
    </nav>

    <main class="flex">
      <div class="column-main tile">                     #B
        <h1>Team collaboration done right</h1>           #B
        <p>Thousands of teams from all over the          #B
          world turn to <b>Ink</b> to communicate        #B
          and get things done.</p>                       #B
      </div>                                             #B

      <div class="column-sidebar">                #C
        <div class="tile">
          <form class="login-form">
            <h3>Login</h3>
            <p>
              <label for="username">Username</label>
              <input id="username" type="text"
                name="username"/>
            </p>
            <p>
              <label for="password">Password</label>
              <input id="password" type="password"
                name="password"/>
            </p>
            <button type="submit">Login</button>
          </form>
        </div>
        <div class="tile centered stack">
          <small>Starting at</small>
          <div class="cost">
            <span class="cost-currency">$</span>
            <span class="cost-dollars">20</span>
            <span class="cost-cents">.00</span>
          </div>
          <a class="cta-button" href="/pricing">
            Sign up
          </a>
        </div>
      </div>
    </main>
  </div>
```

```
</body>
</html>
```

This HTML includes a link to styles.css. Create a file with that name and enter the CSS shown in listing 4.2. Most of these styles should look familiar after the last chapter.

Listing 4.2 Base styles for the page

```
*,                              #A
::before,                       #A
::after {                       #A
  box-sizing: border-box;       #A
}                               #A

body {                                          #B
  margin: unset;                                #B
  background-color: #709b90;                    #B
  font-family: Helvetica, Arial, sans-serif;    #B
}                                               #B

.stack * + * {                #C
  margin-block-start: 1.5em; #C
}                             #C

.container {                 #D
  max-inline-size: 1080px;   #D
  margin-inline: auto;       #D
}                            #D
```

Now that the page is set up, let's start laying out some things with flexbox. You'll start with the navigational menu at the top.

4.1.1 Building a basic flexbox menu

For this example, you'll want the navigational menu to look like figure 4.3. Most of the menu items will align to the left, but you'll move one over to the right side.

Figure 4.3 Navigation menu with items laid out using flexbox

To build this menu, you should consider which element needs to be the flex container; keep in mind that its child elements will become the flex items. In the case of our page menu, the flex container should be the unordered list (`<ul>`). Its children, the list items (`<li>`), will be the flex items. Here's what this looks like:

```
<ul class="site-nav">
  <li><a href="/">Home</a></li>
  <li><a href="/features">Features</a></li>
  <li><a href="/pricing">Pricing</a></li>
  <li><a href="/support">Support</a></li>
  <li class="nav-right"><a href="/about">About</a></li>
</ul>
```

We'll take a few passes at this as I walk you through building this menu step by step. First, you'll apply `display: flex` to the list. You'll also need to override the default list styles from the user-agent stylesheet and apply the colors. Figure 4.4 shows the result for these steps.

Figure 4.4 Menu with flexbox and colors applied

In the markup, you've given the `<ul>` a `site-nav` class, which you can use to target it in the styles. Add these declarations to your stylesheet.

Listing 4.3 Applying flexbox and colors to the menu

```
.site-nav {
  display: flex;          #A
  padding: unset;             #B
  list-style-type: none;      #B
  background-color: #5f4b44;
}

.site-nav > li > a {
  background-color: #cc6b5a;
  color: white;
  text-decoration: none;      #C
}
```

Note that you're working with three levels of elements here: the `site-nav` list (the flex container), the list items (the flex items), and the anchor tags (the links) within them. I've used direct descendant combinators (>) to ensure you only target direct child elements. This is probably not strictly necessary; it's unlikely future changes will add a nested list inside the navigational menu, but it doesn't hurt to play it safe. If you're not familiar with this combinator, see appendix A for more information.

4.1.2 Adding padding and spacing

Our menu looks rather scrawny at this point. Let's flesh it out a bit with some padding. You'll add padding to both the container and to the menu links. After this step, your menu

will look like figure 4.5.

Figure 4.5 Menu with padding and link styles added

If you're not too familiar with building this sort of menu (whether with flexbox or any other layout method), it's important to note how to do this. In the examples, you'll apply the menu item padding to the internal `<a>` elements, not the `<li>` elements. You'll need the entire area that looks like a menu link to behave like a link when the user clicks it. Because the link behavior comes from clicking the `<a>` element, you don't want to turn the `<li>` into a big nice-looking button, but only have a small clickable target area (the `<a>`) inside it.

Update your styles to match those in this listing. This will fill out the padding of the menu.

Listing 4.4 Adding padding to the menu and its links

```
.site-nav {
  display: flex;
  padding: 0.5rem;          #A
  list-style-type: none;
  background-color: #5f4b44;
}

.site-nav > li > a {
  display: block;            #B
  padding: 0.5em 1em;          #C
  background-color: #cc6b5a;
  color: white;
  text-decoration: none;
}
#A Adds padding to menu, outside of the links
#B Makes links block level so they add to the parent elements' height
#C Adds padding inside the links
```

You'll notice you made the links a display block. If they were to remain inline, the height they'd contribute to their parent would be derived from their line height—not their padding and content, which is the behavior you want here.

Next, you'll need to add space between the menu items. You can do this with margins, but flexbox has a special property just for this called `gap`. For example, applying `gap: 1rem` will put 1-rem of space between each of the flex items.

Additionally, flexbox allows you to use `margin: auto` to fill all available space between flex items. You can also use this to push the final menu item all the way to the right side. After applying these changes, the menu will be complete (figure 4.6).

Figure 4.6 Gap and margins apply spacing between flex items

See the styles for this in listing 4.5, where you'll apply a gap. This is stored in a custom property for reuse later. These changes also apply an `auto` margin to the left side of the last button, which pushes the last button all the way to the right. Add this listing to your stylesheet.

Listing 4.5 Using margins to space the items

```
:root {
  --gap-size: 1.5rem;
}

.site-nav {
  display: flex;
  gap: var(--gap-size);     #A
  padding: 0.5rem;
  list-style-type: none;
  background-color: #5f4b44;
}

.site-nav > .nav-right {
  margin-inline-start: auto;    #B
}
#A Adds a 1.5 rem gap between each of the flex items
#B Auto margins inside a flexbox will fill the available space.
```

You applied the auto margin to only one element (About). If you were to apply it to the Support menu item instead, it would shift both links to the right.

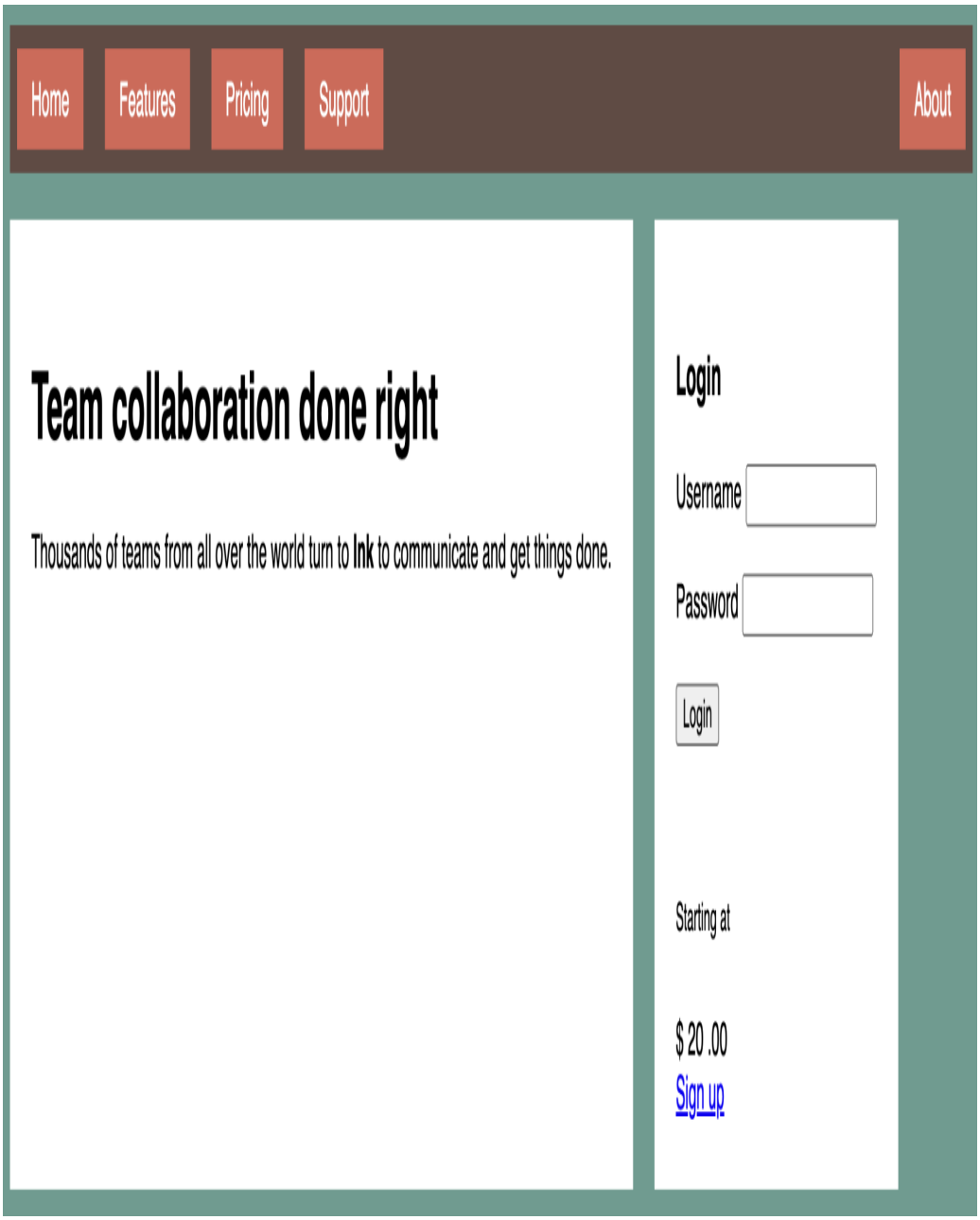
There are more options for spacing flex items, including the ability to spread them out evenly across the entire flex container. We'll take a look at these options in a bit, but the styles you've used here are appropriate for this nav menu.

4.2 Flex item sizes

When it comes to sizing flexbox elements, you can use the familiar `width` and `height` properties, but flexbox provides more options for sizing than these properties alone can accomplish. Let's look at one of the more useful flexbox properties, `flex`.

The `flex` property controls the size of the flex items along the main axis (normally the width). In listing 4.6, you'll apply a flex layout to the main area of the page, then you'll use the `flex` property to control the size of the columns. Initially, your main area will look like figure 4.7.

Figure 4.7 Main area with a flex layout applied to the white tiles



Add the styles in listing 4.6 to your stylesheet. This provides a white background to the three tiles via the `tile` class and a flex layout to the `<main>` element by targeting the `flex` class.

Listing 4.6 Applying flexbox to the main container

```
.tile {                       #A
  padding: 1.5em;             #A
  background-color: #fff;     #A
}                             #A

.flex {                     #B
  display: flex;            #B
  gap: var(--gap-size);     #B
}                           #B
```

Now your content is divided into two columns: on the left is the larger area for the primary content of the page (`column-main`), and on the right is a login form and a small pricing box (`column-sidebar`). You haven't done anything yet to specify the width of the two columns, so they size themselves naturally, based on their content. On my screen (figure 4.7), this means they don't quite fill the width of the available space, though this isn't necessarily the case with a smaller window size.

The `flex` property, which is applied to flex items, gives you a number of options. Let's apply the most basic use case first to get familiar with it. You'll use the `column-main` and `column-sidebar` classes to target the columns, using `flex` to apply widths of two-thirds and one-third. Add the following to your stylesheet.

Listing 4.7 Using the `flex` property to set column widths

```
.column-main {
  flex: 2;
}

.column-sidebar {
  flex: 1;
}
```

Now the two columns grow to fill the space, so together they are the same width as the `nav` bar, with the main column twice as wide as the sidebar. Flexbox was kind enough to take care of the math for you. Let's take a closer look at what's going on.

The `flex` property is shorthand for three different sizing properties: `flex-grow`, `flex-shrink`, and `flex-basis`. In this listing, you've only supplied `flex-grow`, leaving the other two properties to their default values (1 and 0% respectively). So `flex: 2` is equivalent to `flex: 2 1 0%`. These shorthand declarations are generally preferred, but you can also declare the three individually:

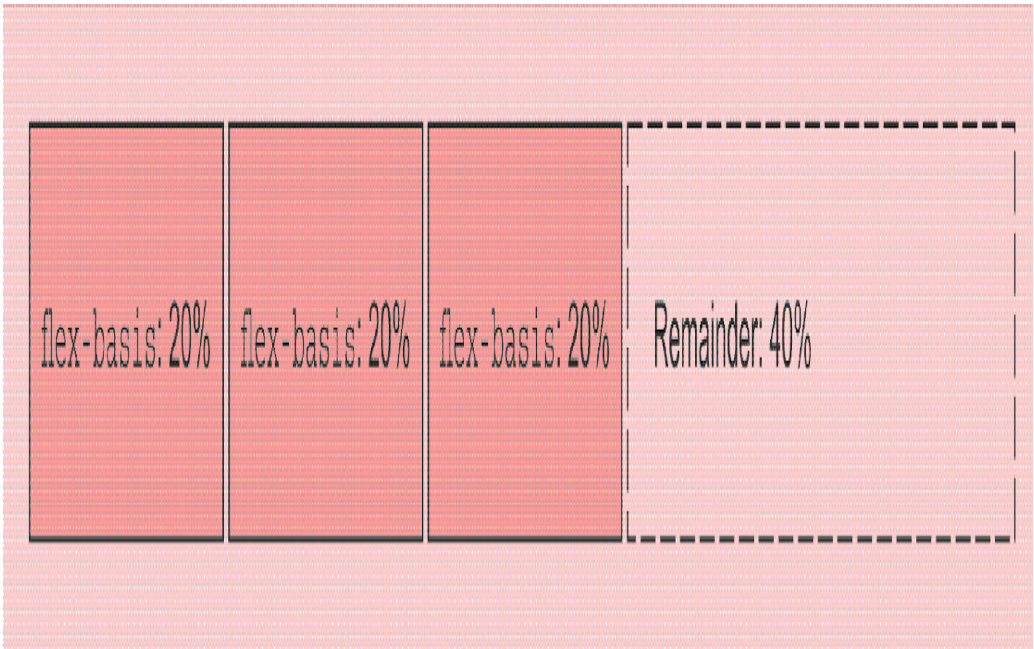
```
flex-grow: 2;
flex-shrink: 1;
flex-basis: 0%;
```

Let’s look at what these three properties mean, one at a time. We’ll start with `flex-basis`, as the other two are based on it.

4.2.1 Flex basis

The *flex basis* defines a sort of starting point for the size of an element—an “initial main size” (referring to its size along the main axis). The `flex-basis` property can be set to any value that would apply to `width`, including values in px, ems, or percentages. Its initial value is `auto`, which means the browser will look to see if the element has a `width` declared. If so, the browser uses that size; if not, it determines the element’s size naturally by the contents. This means that `width` will be ignored for elements that have any flex basis other than `auto`. Figure 4.8 illustrates this.

Figure 4.8 Three flex items with a flex basis of 20%, giving each an initial main size (width) of 20%



Once this initial main size is established for each flex item, they may need to grow or shrink in order to fit (or fill) the flex container along the main axis. That’s where `flex-grow` and `flex-shrink` come in.

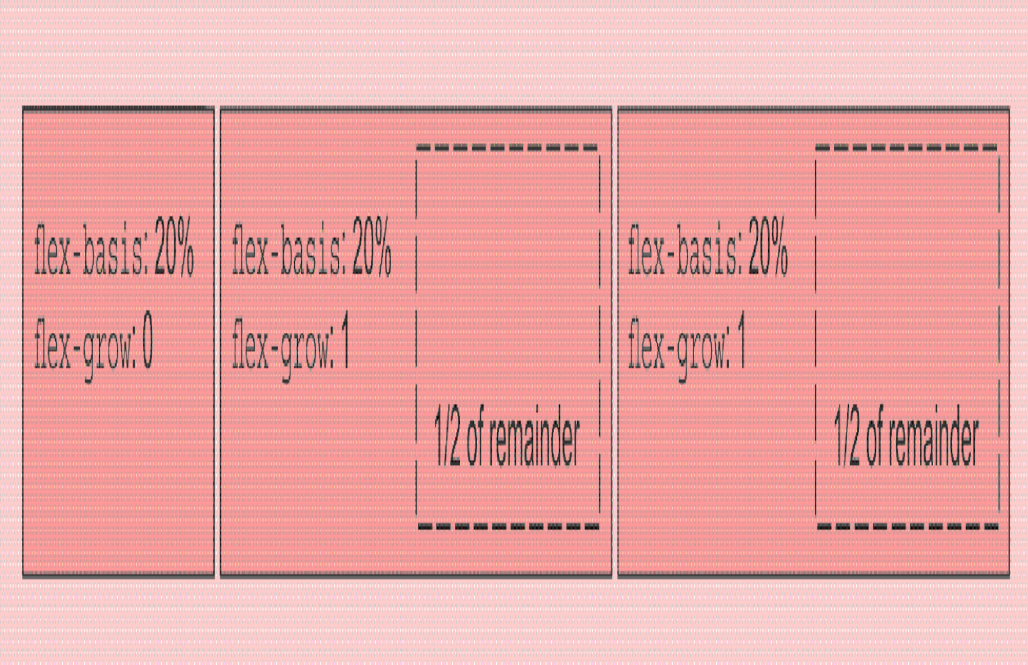
4.2.2 Flex grow

Once `flex-basis` is computed for each flex item, they (plus any gap or margins between them) will add up to some width. This width may not necessarily fill the width of the flex container, leaving a remainder (figure 4.8).

The remaining space (or remainder) will be consumed by the flex items based on their `flex-grow` values, which is always specified as a non-negative integer. If an item has a `flex-grow`

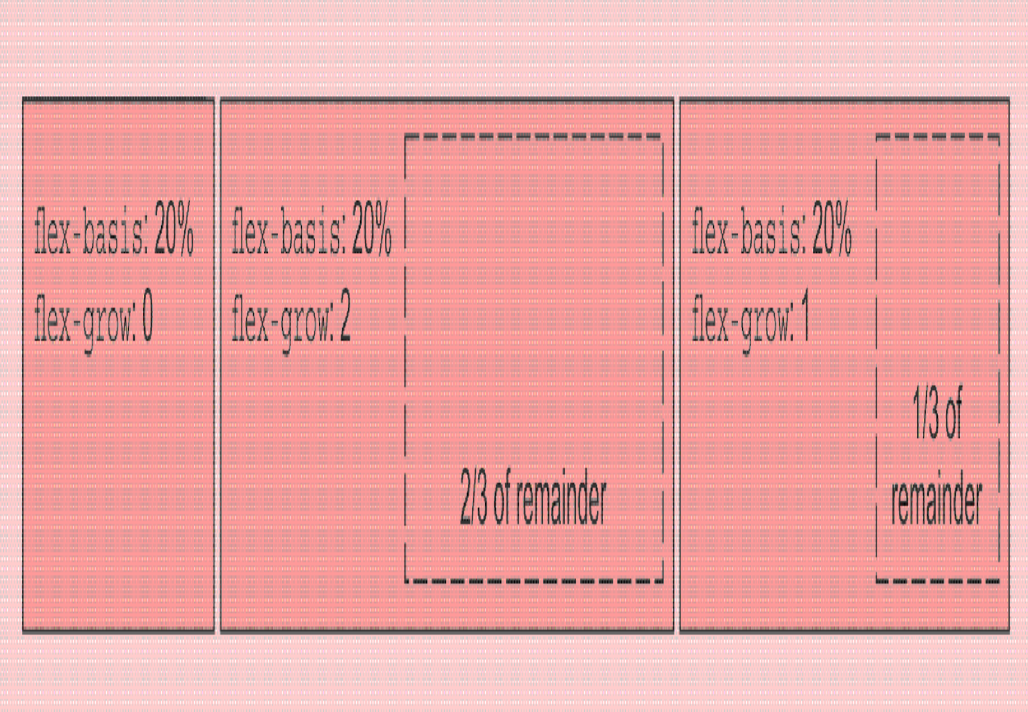
of `0`, it won't grow larger than its flex basis. If any items have a non-zero flex grow value, those items will grow until all of the remaining space is used up. This means the flex items will fill the width of the container (figure 4.9).

Figure 4.9 Remaining width partitioned evenly among items with equal `flex-grow` values



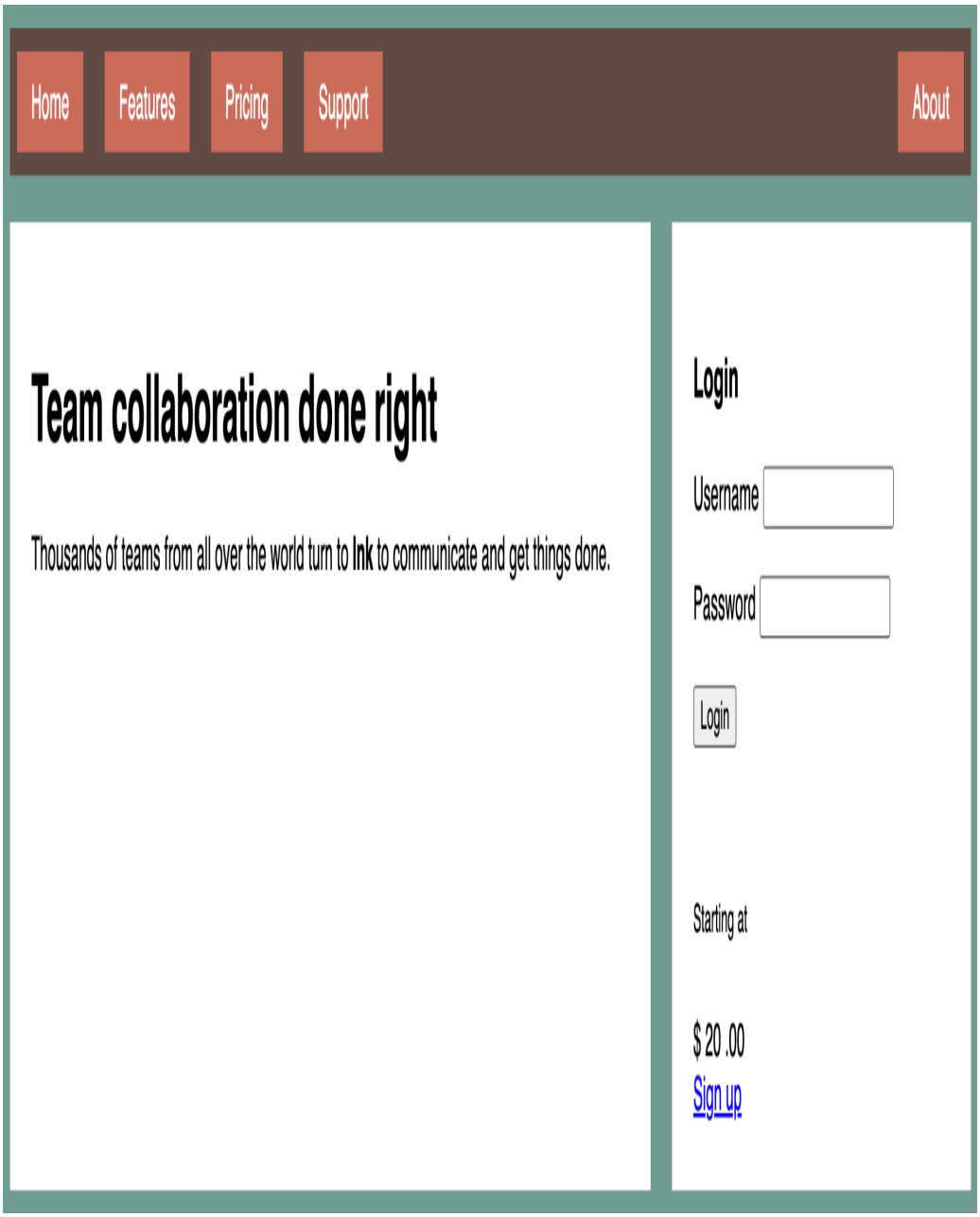
Declaring a higher flex grow value gives that element more "weight"; it'll take a larger portion of the remainder. An item with `flex-grow: 2` will grow twice as much as an item with `flex-grow: 1` (figure 4.10).

Figure 4.10 Items with a higher `flex-grow` value consume a higher proportion of the remaining available width.



This is what you did on your page. The shorthand declarations `flex: 2` and `flex: 1` set a flex basis of 0%, so 100% of the container's width is the remainder (minus the 1.5-em gap between the two columns). The remainder is then distributed to the two columns: two-thirds to the first column and the remaining third to the second (figure 4.11).

Figure 4.11 The two columns fill the flex container's width.



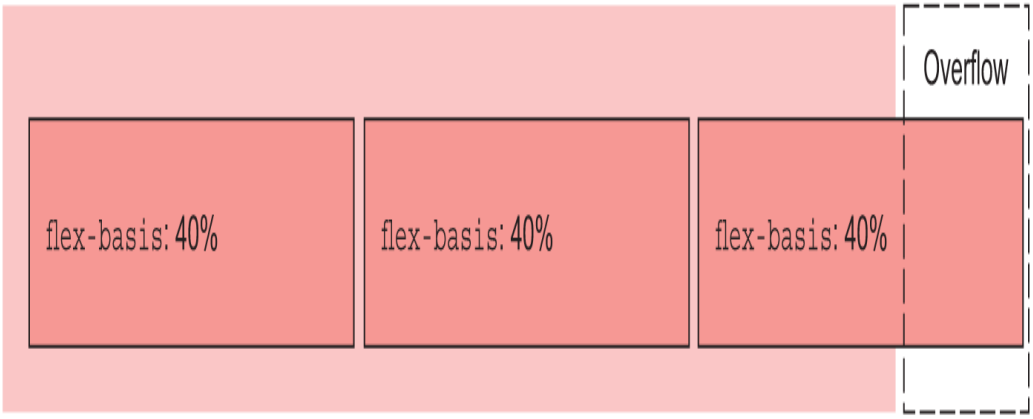
Favor the use of the shorthand `flex` property as you did here, instead of declaring only `flex-grow`. Unlike most shorthand properties, the values aren't set to their initial values when omitted. Instead, the shorthand assigns useful default values for any of the three that you omit: flex grow factor of 1, flex shrink factor of 1, and a flex basis of 0%. These are most commonly what you'll need.

4.2.3 Flex shrink

The `flex-shrink` property follows similar principles as `flex-grow`. After determining the initial main size of the flex items, they could exceed the size available in the flex container.

Without `flex-shrink`, this would result in an overflow (figure 4.12).

Figure 4.12 Flex items can have an initial size exceeding that of the flex container.



The `flex-shrink` value for each item indicates whether it should shrink to prevent overflow. If an item has a value of `flex-shrink: 0`, it will not shrink. Items with a value greater than 0 will shrink until there is no overflow. An item with a higher value will shrink more than an item with a lower value, proportional to the `flex-shrink` values.

As an alternate approach to your page, you could achieve similar column sizing by relying on `flex-shrink`. To do this, specify the flex basis for each column using the desired percent (66.67% and 33.33%). The width plus the 1.5-em gap would overflow by 1.5-em. Give both columns a `flex-shrink` of 1, and 0.75-em is subtracted from the width of each, allowing them to fit in the container. Listing 4.8 shows what this code would look like.

Listing 4.8 Using the `flex` property to set widths

```
.column-main {
  flex: 66.67%;   #A
}

.column-sidebar {
  flex: 33.33%;    #B
}
```

This is a different approach to get effectively the same result as before (listing 4.7). Either one suits your purposes for this page.

Note

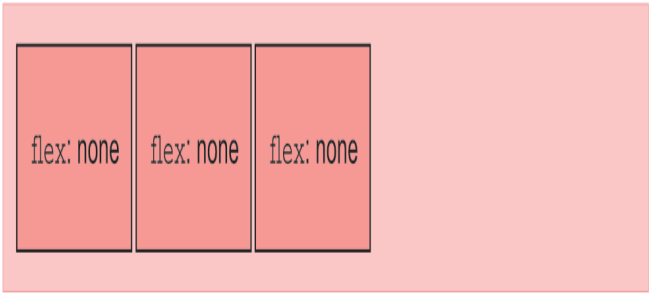
If you look at the nitty gritty details, there's a slight discrepancy between the results of listing 4.7 and listing 4.8. The reason for this is a little complicated, but in short, it's because the `column-main` has padding but the `column-sidebar` doesn't. The padding changes the way the initial main size of the flex item is determined when `flex-basis` is 0%. Therefore,

the `column-main` from listing 4.7 is slightly wider than that in listing 4.8. If you need your measurements to be precise, grid is a better approach (chapter 5).

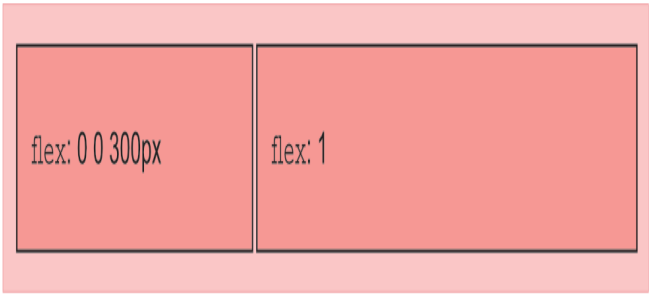
4.2.4 Some practical examples

You can make use of the `flex` property in countless ways. You can define proportional columns using `flex-grow` values or `flex-basis` percentages as you did on your page. You can define fixed width columns and "fluid" columns that scale with the viewport. You can build a "grid system", which is something libraries such as Bootstrap do, to provide a series of reusable classes defining columns of preset sizes. Figure 4.13 illustrates some layouts you can build with flexbox.

Figure 4.13 Some ways you can define item sizes using flex

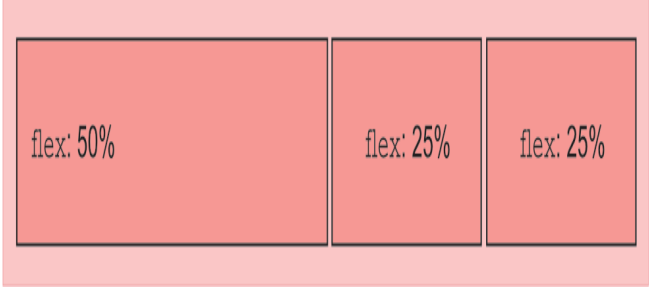


Items grow to their natural width. They don't necessarily fill the width of the container.



First item has a fixed width of 300 px. Optionally add a max-width to prevent its contents from forcing it to grow wider. Second item fills all remaining space.

The first and third item have a fixed width of 200 px each. The center item grows to fill all remaining space.



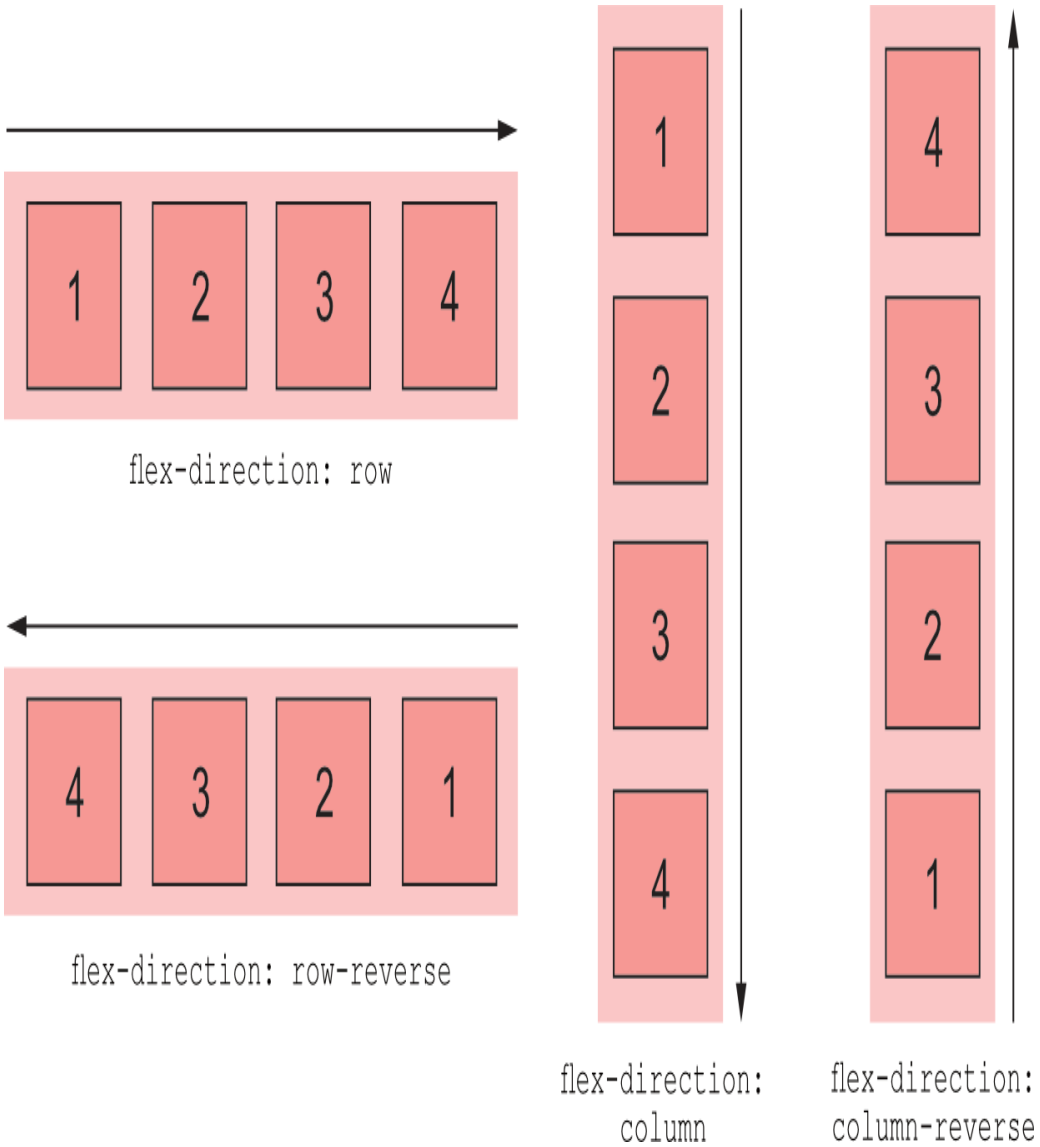
Items grow to the percentage widths specified.

The third example illustrates what was once called the "Holy Grail" layout. This is a layout that was once notoriously difficult in CSS. The two sidebars are a fixed width, whereas the center column is "fluid," meaning it will grow to fill the available space. Most notably, all three columns are equal height, as determined by their contents. As you might imagine, you can mix and match these layouts in any number of ways, with any different number of flex items.

4.3 Flex direction

Another important option in flexbox is the ability to shift the direction of the axes. The `flex-direction` property, applied to the flex container, controls this. Its initial value (`row`) causes the items to flow in the inline direction, as you've done. Specifying `flex-direction: column` causes the flex items to stack vertically (in the block direction) instead. Flexbox also supports `row-reverse` to flow items right to left, and `column-reverse` to flow items bottom to top (figure 4.14).

Figure 4.14 Changing the flex direction changes the main axis. The cross axis also changes to remain perpendicular to the main axis.



You'll use this in the right column of the page, where two tiles are stacked atop one another. This may seem unnecessary; after all, the two tiles in the right column are already stacked.

Normal block elements behave like this. But there's a problem with the page layout that isn't immediately obvious. It shows up if you add more content to the main tile. This is shown in figure 4.15.

Add a few more headings and paragraphs to the `column-main` in your code. You'll see that the main tile grows beyond the bottom of the tiles on the right. Flexbox is supposed to provide columns of equal height, so why isn't this working?

Figure 4.15 The main tile grows beyond the height of the tiles in the right column (dashed line indicates the size of `column-sidebar`).

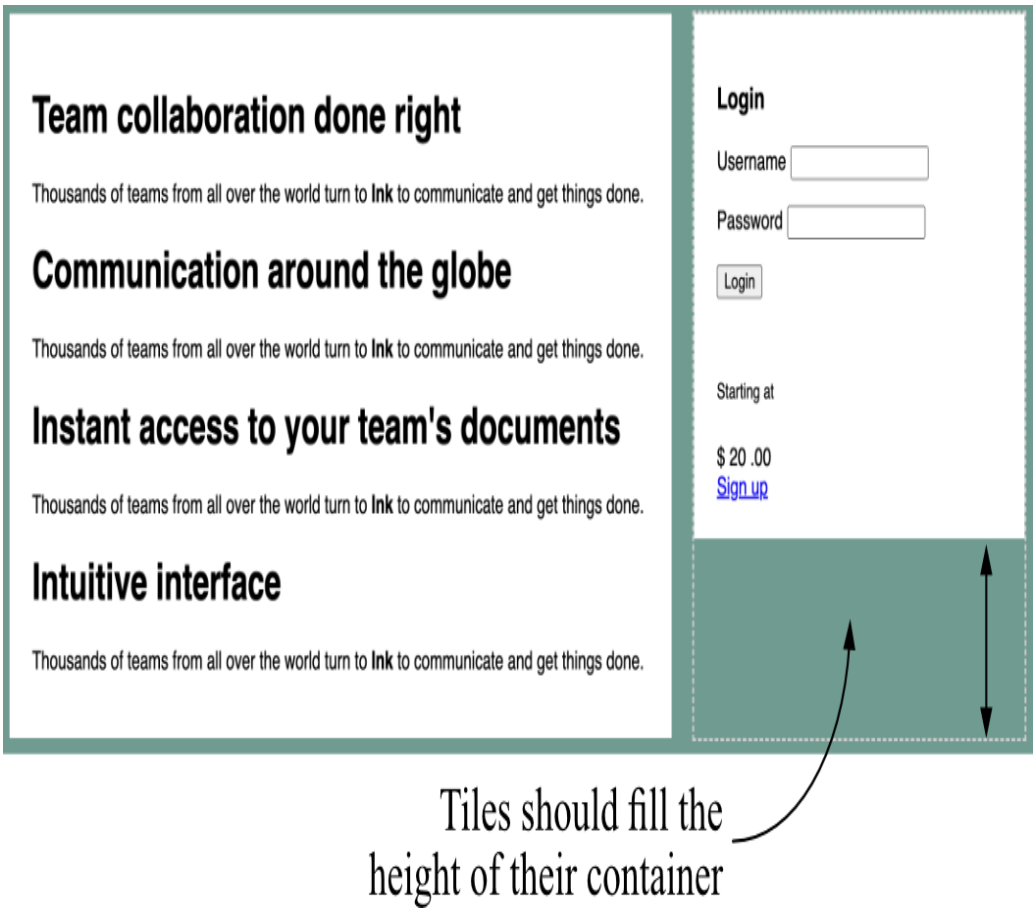


Figure 4.15 shows (by the dashed outline I've added) that the flex items are in fact equal height. The problem is that the tiles inside the right flex item don't grow to fill it.

The ideal layout would be the one shown in figure 4.16. The two tiles on the right grow to fill the column, even when the content on the left is longer. Before flexbox, this effect was impossible to achieve using CSS (though it was possible with a little help from JavaScript).

Figure 4.16 The ideal layout: Tiles in the right column align with the large tile on the left.

4.3.1 Changing the flex direction

What you need is for the two columns to grow if necessary to fill the container's height. To do this, turn the right column (the `column-sidebar`) into a flex container with a `flex-direction: column`. Then, apply a non-zero `flex-grow` value to both tiles within. The next listing shows the code for this. Update your stylesheet to match.

Listing 4.9 Creating a flex column on the right

```
.column-sidebar {           #A
  flex: 1;                  #A
  display: flex;            #A
  flex-direction: column;   #A
  gap: var(--gap-size);     #A
```

```
}

.column-sidebar > .tile {
  flex: 1;        #B
}
#A A flex item for the outer flexbox and a flex container for the new inner
one
#B Applies flex-grow to the items within
```

You now have nested flexboxes. The element `<div class="column-sidebar">` is a flex item for the outer flexbox, and it's the flex container for the inner flexbox. The overall structure of these elements looks like this (with text removed for brevity):

```
<main class="flex">
  <div class="column-main tile">
    ...
  </div>
  <div class="column-sidebar">
    <div class="tile”>...</div>
    <div class="tile”>...</div>
  </div>
</div>
```

The inner flexbox here has a flex direction of `column`, so the main axis is rotated. It flows from top to bottom (and the cross axis now flows from left to right). This means that for those flex items, `flex-basis`, `flex-grow`, and `flex-shrink` now apply to the element height rather than the width. By specifying `flex: 1`, the height of these items will stretch if necessary to fill the container. Now, regardless of which side is taller, the bottom of the large tile and the bottom of the second smaller tile align.

When working with a vertical flexbox (`column` or `column-reverse`), the same general concepts for rows apply, but there's one difference to keep in mind—recall that in CSS, working with height is fundamentally different than working with widths. A flex container will be 100% the available width, but the height is still determined naturally by its contents. This behavior does not change when you rotate the main axis.

The flex container's height is determined by its flex items. They fill it perfectly. In a vertical flexbox, `flex-grow` and `flex-shrink` applied to the items will have no effect unless something else forces the height of the flex container to a specific size. On your page, that "something" is the height derived from the outer flexbox.

4.3.2 Styling the login form

You've now applied the overall layout to the entire page. All that remains is styling the smaller elements in the two tiles on the right: the login form and the signup link. You don't need flexbox for the login form, but for the sake of completeness, I'll walk you through it briefly. Afterwards, the form should look like figure 4.17.

Figure 4.17 Login form

The `<form>` has the class `login-form`, so you'll use that to target it in your CSS. Add the code in this listing to your stylesheet. This will style the login form in three parts: the heading, the input fields, and the button.

Listing 4.10 Login form styles

```
.login-form h3 {                #A
  margin: 0;                    #A
  font-size: 0.9em;             #A
  font-weight: bold;            #A
  text-align: end;              #A
  text-transform: uppercase;    #A
}                               #A

.login-form input:not([type=checkbox]):not([type=radio]) {  #B
  display: block;
  inline-size: 100%;
}

.login-form button {            #C
  margin-block-start: 1em;      #C
  border: 1px solid #cc6b5a;    #C
```

```
  background-color: white;     #C
  padding: 0.5em 1em;          #C
  cursor: pointer;             #C
}                              #C
```

First is the heading, which uses font properties you should be familiar with. You used `text-align` to shift the text to the right and `text-transform` to make the text all uppercase. Notice it wasn't capitalized in the HTML. When capitalization is purely a styling decision, as this is, you would normally capitalize it according to standard grammatical rules in the HTML and use CSS to manipulate it. This way, you can change it in the future without having to retype portions of the HTML in proper caps.

The second ruleset styles the input boxes. The selector here is peculiar, mainly because the `<input>` element is peculiar. The input element is used for text inputs and passwords, as well as a number of other HTML5 inputs that look similar, like numbers, emails, and dates. It's also used for form input items that look entirely different; namely, radio buttons and checkboxes.

I've combined the `:not()` pseudo-class with the attribute selectors `[type=checkbox]` and `[type=radio]` (see appendix A for details). This targets all input elements except checkboxes and radio buttons. It's an exclusion-based approach, stating what you don't want to target. You could alternatively use an inclusion-based approach, using multiple attribute selectors to name every type of input you want to target, but this can get rather long.

Note

The form for this page only uses a text input and a password input, but it's important for you to consider other markup the CSS could be applied to in the future and try to account for that.

Inside the ruleset, you make the inputs display `block`, so they appear on their own line. You also had to specify an inline-size of 100%. Normally, display block elements automatically fill the available width, but `<input>` is a bit different; its width is determined by the `size` attribute, which indicates roughly the number of characters it should contain without scrolling. This attribute reverts to a default value if not specified. You can force a specific width with the CSS `width` or `inline-size` property.

The third ruleset styles the Login button. These styles are mostly straightforward; however, the `cursor` property may be unfamiliar. It controls the appearance of the mouse cursor when the cursor is over the element. The value `pointer` turns the cursor into a hand with a pointing finger, like the default cursor when pointing at links. This communicates to the user that they can click the element. It gives a final detail of polish to the button.

4.4 Alignment, spacing, and other details

You should now have a solid grasp of the most essential parts of flexbox. But as I mentioned earlier, there's a wide array of options that you'll occasionally need. These pertain mostly to the alignment or spacing of flex items within the flex container. You can also enable line wrapping or reorder individual flex items. The properties that control these are all illustrated on the following pages: table 4.1 lists all the properties that may be applied to a flex container, and table 4.2 lists all the properties for flex items.

In general, you'll begin a flexbox with the methods we've already covered:

- Identify a container and its items and use `display: flex` on the container
- If necessary, set a `gap` and/or `flex-direction` on the container
- Declare `flex` values for the flex items where necessary to control their size

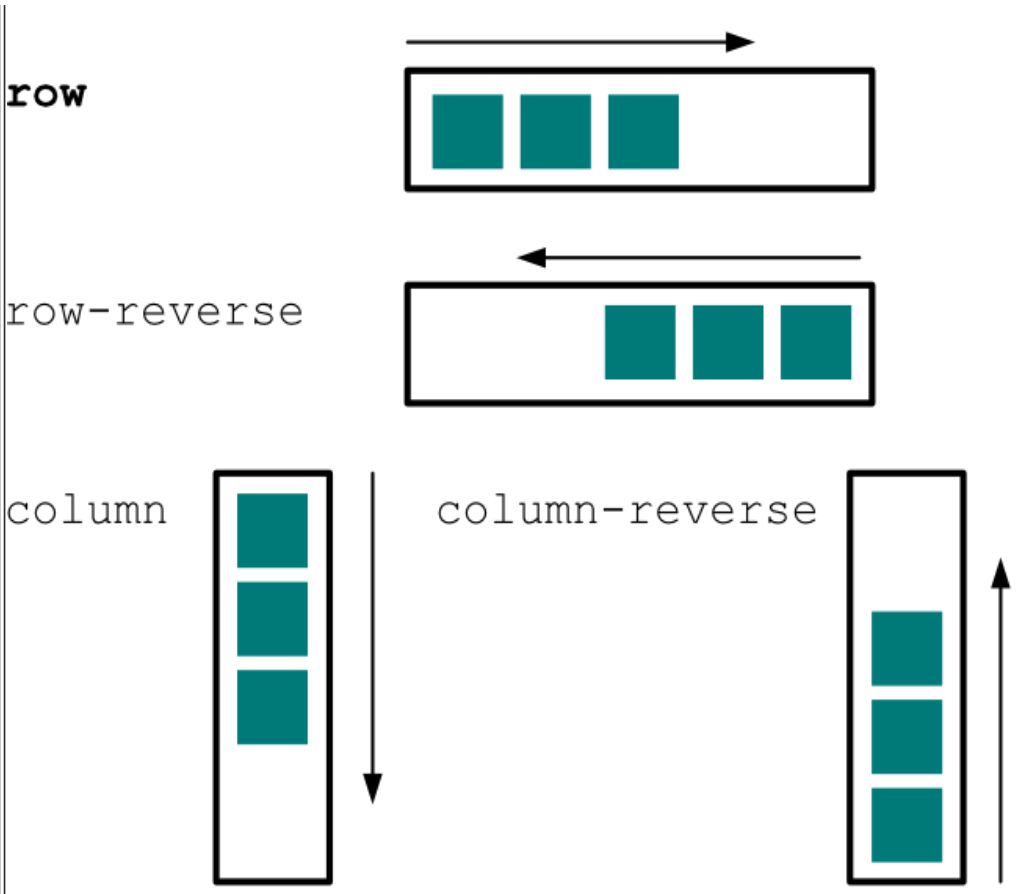
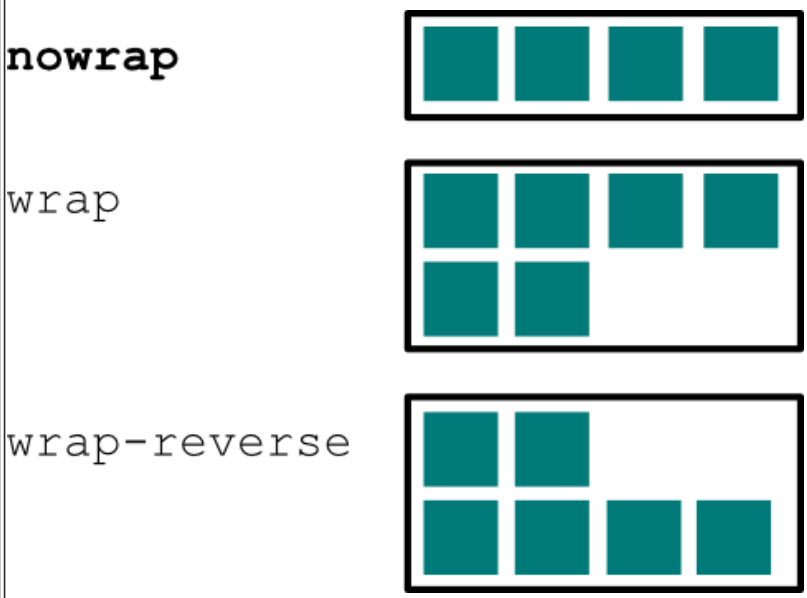
Once you've put elements roughly where they belong, you can add other flexbox properties where necessary. My suggestion is to get familiar with the concepts we've covered thus far. Go ahead and read through the rest of this chapter to get a sense for the other options flexbox supplies, but don't worry about committing them all to memory until you need them. When you find you do need them, return here as a reference. Most of these final options are fairly straightforward, though you'll only need them occasionally.

4.4.1 Understanding flex container properties

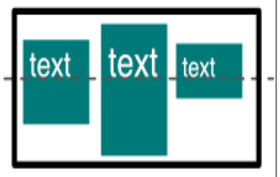
Several properties can be applied to a flex container to control the layout of its flex items. The first is `flex-direction`, which I've already covered in section 4.3. Let's look at some others.

Table 4.1 Flex container properties

Property	Values (initial values in bold)
`flex-direction` This specifies the direction of the main axis. The cross axis will be perpendicular to the main axis.	

	
`flex-wrap` This specifies whether flex items will wrap on to a new row inside the flex container (or on to a new column if `flex-direction` is `column` or `column-reverse`).	

`flex-flow`	Shorthand for `<flex-direction>` `<flex-wrap>`
`gap`	Specifies space between each flex item. Shorthand for `<row-gap>` `<column-gap>`.
`justify-content` Controls how items are positioned along the main axis.	**flex-start** start left  flex-end end right  center  space-between  space-around  space-evenly 
`align-items` Controls how items are positioned along the cross axis.	

	flex-start start self-start  flex-end end self-end  center  **stretch**  baseline 
`align-content` If flex-wrap is enabled, this controls the spacing of the flex rows along the cross axis. If items don’t wrap, this property is ignored.	

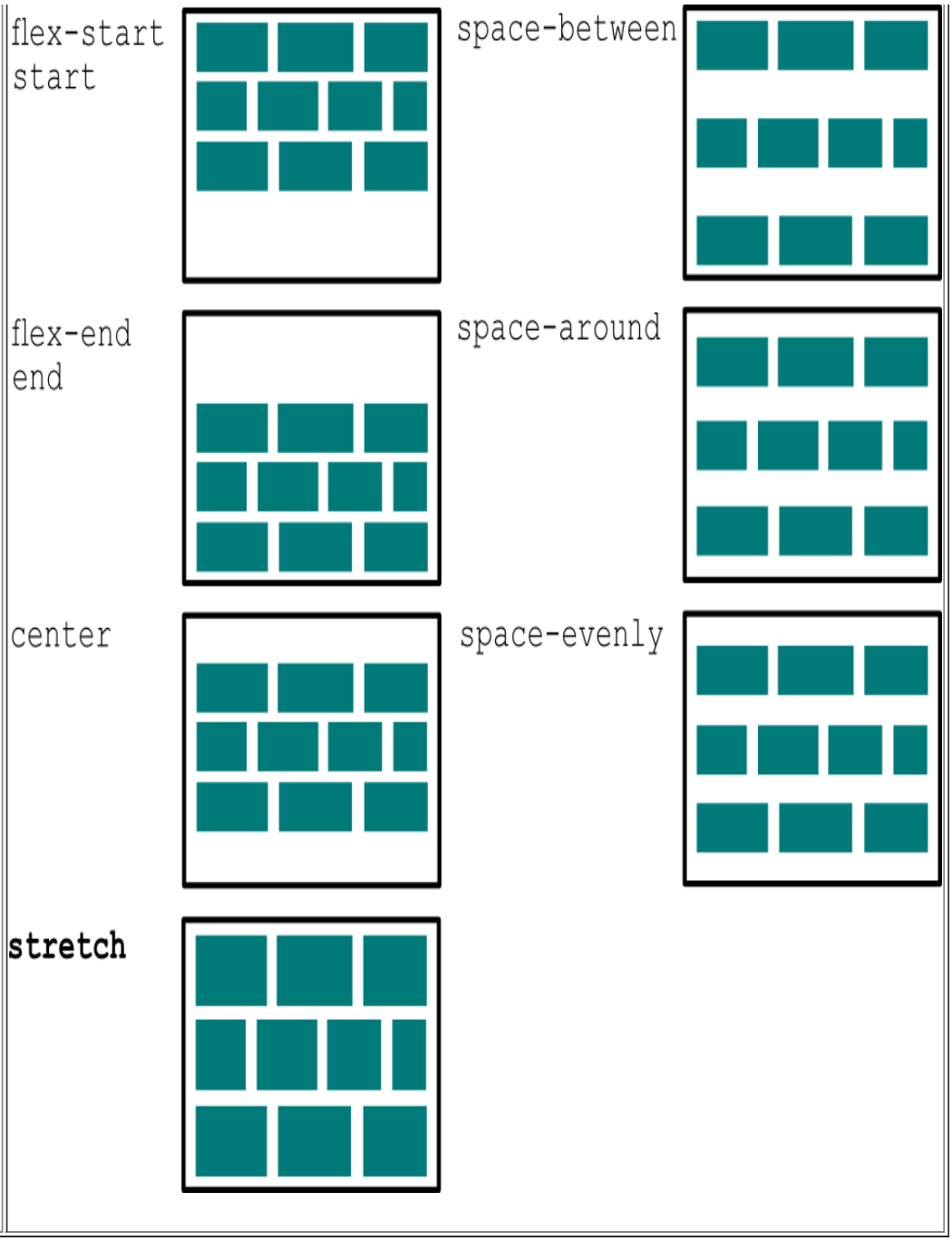
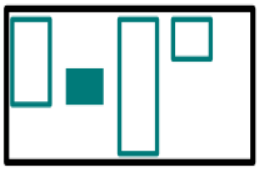
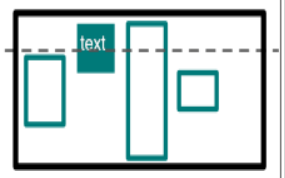


Table 4.2 Flex item properties

Property	Values (initial values in bold)
`flex-grow`	

An integer that specifies the “growth factor,” determining how much the item will grow along the main axis to fill unused space	
`flex-shrink` An integer that specifies the “shrink factor,” determining how much the item will shrink along the main axis, if needed, to prevent overflow. Ignored if the container has flex wrap enabled.	
`flex-basis` Specifies the initial size of the item before	`<length>` or `<percent>`

`flex-grow` or `flex-shrink` is applied.	
`flex`	Shorthand for: `<flex-grow>` `<flex-shrink>` `<flex-basis>`
`align-self` Controls how the item is aligned on the cross axis. This will override the container's `align-items` value for the specific item(s). Ignored if the item has an `auto` margin set on the cross axis.	**auto**  center  flex-start start self-start  stretch  flex-end end self-end  baseline 
`order` An integer that moves a flex item to a specific position among its siblings, disregarding source order.	

Flex-wrap property

The `flex-wrap` property can be used to allow flex items to wrap to a new row (or rows). This can be set to `nowrap` (the initial value), `wrap`, or `wrap-reverse`. When wrapping is enabled, the items don't shrink according to their `flex-shrink` values. Instead, any items that would overflow the flex container wrap onto a new line.

If the flex direction is `column` or `column-reverse`, then `flex-wrap` will allow the flex items to overflow into a new column. However, this only happens if something constrains the height of the container; otherwise, it grows to contain its flex items.

Flex-flow property

The `flex-flow` property is shorthand for both `flex-direction` and `flex-wrap`. For example, `flex-flow: column wrap` specifies that the flex items will flow from top to bottom, wrapping onto a new column if necessary.

Justify-content property

The `justify-content` property controls how the items are spaced along the main axis if they don't fill the size of the container. Supported values include a number of new keywords: `flex-start`, `flex-end`, `center`, `space-between`, `space-around`, and `space-evenly`. A value of `flex-start` (the default behavior) stacks the items against the beginning of the main axis—the left side in a normal row direction. There will be no space between them unless there is a `gap` or items have margins specified. A value of `flex-end` stacks the items at the end of the main axis and, accordingly, `center` centers them.

The values `start` and `end` are logical values that align items left or right, depending on the writing mode, regardless of the direction of the main axis. The `left` and `right` values are absolute, always aligning left or right, respectively, independent of writing mode or flex direction.

There are three ways to evenly space flex items, each providing a different amount of space before the first item and after the last item. The value `space-between` puts the first flex item flush to the beginning of the main axis and the last item flush to the end. The value `space-around` adds space outside the first and last items that is equal to half the space between items. And `space-evenly` adds space outside the first and last items that is equal to the space between items.

Spacing is applied after margins and flex-grow values are calculated. This means if any items have a non-zero `flex-grow` value, or any items have an `auto` margin on the main axis, then `justify-content` has no effect.

Align-items property

Whereas `justify-content` controls item alignment along the main axis, `align-items` adjusts their alignment along the cross axis. The initial value for this is `stretch`, which causes all items to fill the container's height in a row layout, or width in a column layout. This provides columns of equal height.

The other values allow flex items to size themselves naturally, rather than filling the container size. (This is similar conceptually to the `vertical-align` property.)

- `flex-start` and `flex-end` align the items along the start or end of the cross axis (top or bottom of a row, respectively).
- `start` and `end` are logical values that align the items based on the writing mode of the flex container.
- `self-start` and `self-end` are logical values that align each item based on its writing mode; these only differ from `start` and `end` if a flex item has a different writing mode than its flex container.
- `center` centers the items.
- `baseline` aligns the items so that the baseline of the first row of text in each flex item is aligned. The baseline is the line where the bottom edge of the text sits.

The value `baseline` is useful if you want the baseline of a header in one flex item with a large font to line up with the baseline of smaller text in the other flex items. You can also add `first` or `last` to this, to specify which line of text to align. For example, `align-items: last baseline` will align the last line of text in each flex item to one another.

Tip

It's easy to confuse the names of the properties `justify-content` and `align-items`. I remember them by thinking of styling text in a word processor: you can "justify" text to spread it horizontally from edge to edge. And, much like `vertical-align`, you can "align" inline items vertically.

Align-content property

If you enable wrapping (using `flex-wrap`), this property controls the spacing of each row inside the flex container along the cross axis. Supported values are `flex-start`, `flex-end`, `start`, `end`, `center`, `stretch` (the initial value), `space-between`, `space-around`, and `space-evenly`. These values apply spacing similar to the way described above for `justify-content`.

4.4.2 Understanding flex item properties

I've described `flex-grow`, `flex-shrink`, `flex-basis`, and their collective shorthand, `flex` (section 4.2). We'll next look at two additional properties for flex items: `align-self` and `order`.

Align-self property

This property controls a flex item's alignment along its container's cross axis. This does the same thing as the flex container property `align-items`, except it lets you align individual flex items differently. Specifying the value `auto` will defer to the container's `align-items` value—this is the initial value. Any other value overrides the container's setting. The `align-self` property supports the same keyword values as `align-items`: `flex-start`, `flex-end`, `start`, `end`, `self-start`, `self-end`, `center`, `stretch`, and `baseline`.

Order property

Normally, flex items are laid out in the order they appear in the HTML source. They are stacked along the main axis, beginning at the start of the axis. By using the `order` property, you can change the order the items are stacked. You may specify any integer, positive or negative. If multiple flex items have the same value, they'll appear according to source order.

Initially, all flex items have an order of 0. Specifying a value of -1 to one item will move it to the beginning of the list, and a value of 1 will move it to the end. You can specify order values for each item to rearrange them however you wish. The numbers don't necessarily need to be consecutive.

Warning

Be careful with the use of `order`. Making the visual layout order on the screen drastically different from the source order can harm accessibility of your site. Navigation using the Tab key will still follow the source order in most browsers, which can be confusing. Screen-reading software for visually impaired users will also follow the source order in most cases.

4.4.3 Using alignment properties

I'll show you how to use a few of these properties to finish your page. The final tile has a stylized price and a call-to-action (CTA) button. When you're done, the last step in the page should render like figure 4.18.

Figure 4.18 Stylized text using flexbox

The markup for this section is already in your page. It's as follows:

```
<div class="tile centered">
  <small>Starting at</small>
  <div class="cost">
    <span class="cost-currency">$</span>
    <span class="cost-dollars">20</span>
    <span class="cost-cents">.00</span>
  </div>
  <a class="cta-button" href="/pricing">
    Sign up
  </a>
</div>
```

The text $20.00 is wrapped in a `<div class="cost">`, which you'll use as the flex container. It has three flex items for the three different parts of the text you want to align ($, 20, and .00). I've chosen spans for these, rather than divs, because these are inline by default. If for some reason the CSS fails to load, the text $20.00 will still appear on one line.

In the next listing, you'll use `justify-content` to horizontally center the flex items within the container. Then you'll use `align-items` and `align-self` to control their vertical alignment. Add this code to your stylesheet.

Listing 4.11 Setting styles for the cost tile

```
.centered {
  text-align: center;
}

.cost {
  display: flex;
  justify-content: center;     #A
  align-items: center;         #A
  line-height: 0.7;
}

.cost-currency {
  font-size: 2rem;          #B
}
.cost-dollars {
  font-size: 4rem;          #B
}
.cost-cents {
  font-size: 1.5rem;        #B
  align-self: flex-start;      #C
}

.cta-button {
  display: block;
  background-color: #cc6b5a;
  color: white;
  padding: 0.5em 1em;
  text-decoration: none;
}
```

This code lays out the flexbox for the stylized $20.00, as well as defining a `centered` class to center the rest of the text, and a `cta-button` class for the CTA button.

The one strange declaration here is `line-height: 0.7`. This is because the line height of the text inside each flex item is what determines the height of each item. This means that the elements had a little more height than the height of the text itself—an em-height includes descenders, which this text doesn't have, so the characters here are actually a little less than 1-em tall. I arrived at this value purely by trial-and-error until the tops of the 20 and .00 aligned visually. See chapter 13 for more on working with text.

Flexbox is a huge step forward for CSS. Once you're familiar with it, you might be tempted to start using it for everything on the page. I caution you to trust the normal document flow and add flexbox only where you know you'll need it. There's no reason to avoid it; but don't go crazy treating everything as a nail to its hammer.

4.5 Summary

- Children of a flex container become flex items. The browser arranges flex items in a row or column.
- The `flex` shorthand property specifies flex-grow, flex-shrink, and flex-basis values to specify how the size of a flex item is determined.
- Flexboxes can be nested to piece together more complicated layouts and to fill the heights of naturally sized boxes.
- Flexbox automatically creates columns of equal height.
- Flexbox provides several properties that allow you to align and space flex items in countless ways.
- The `align-items` or `align-self` properties can vertically center a flex item inside its flex container.

5 Grid layout

This chapter covers

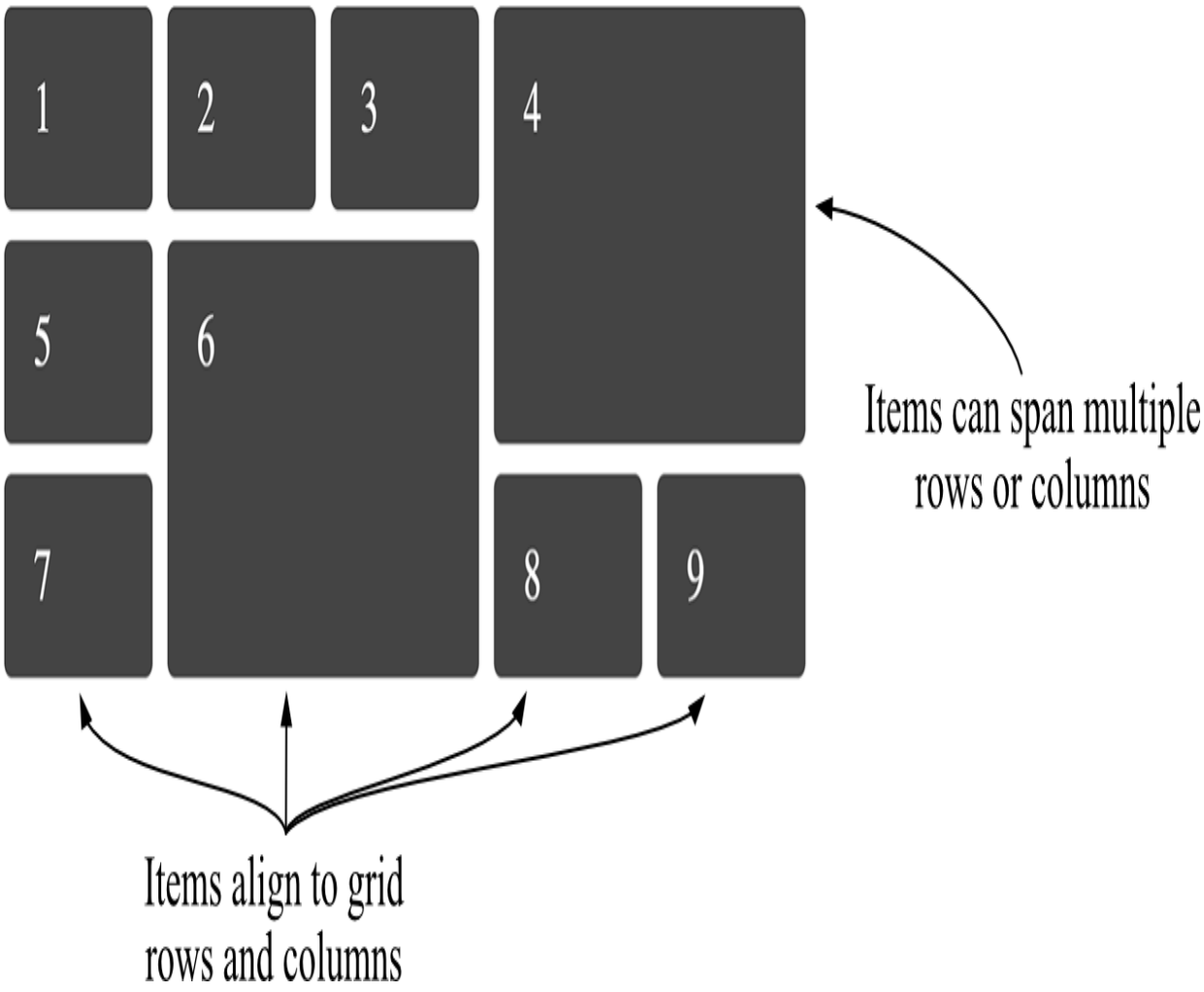
- Using grid to build page layouts
- Understanding grid layout options
- Placing items using grid lines and named grid areas
- Explicit and implicit grid
- Using flexbox and grid together to build a cohesive web page layout

Flexbox has revolutionized the way we do layout on the web, but it’s only a piece of the picture. It has a big brother: a specification called the Grid Layout Module (*grid*). Together, these two specifications provide a full-featured layout engine.

In this chapter, I’ll show you how you can start using grid layout. I’ll give you an overview of how it works, then take you through several examples to illustrate the different things grid layout can do. Building a basic grid is simple. It’s also powerful enough to enable complex layouts, but doing so requires learning new properties and keywords. This chapter will guide you through them.

The CSS grid lets you define a two-dimensional layout of columns and rows and then place elements within the grid. Some elements may fill only one cell of the grid; others can span multiple columns or rows. The size of the grid can be defined precisely, or you can allow it to automatically size itself as needed to fit the contents within. You can place items precisely within the grid, or allow them to flow naturally to fill in the gaps. A grid lets you build complex layouts like the one shown in figure 5.1.

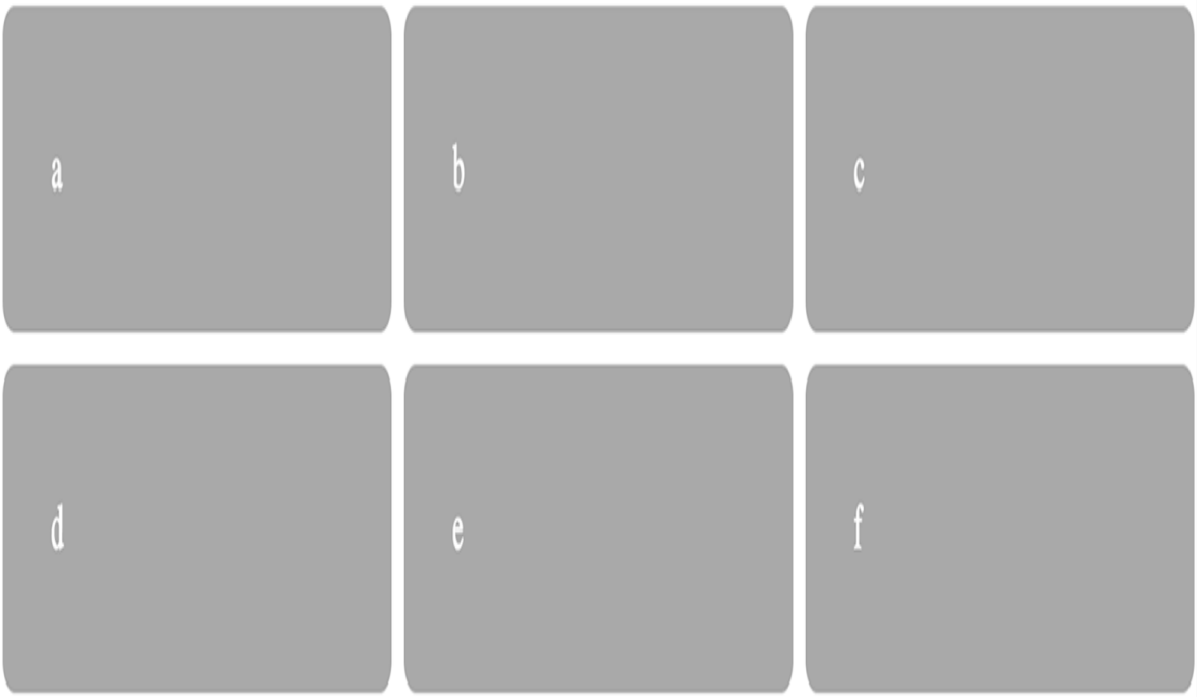
Figure 5.1 Boxes in a sample grid layout



5.1 Building a basic grid

Grid is extremely versatile, so I will walk you through several examples in this chapter to illustrate its capabilities. To get started, you'll build a very basic example. You'll lay out six boxes in three columns, as shown in figure 5.2. The markup for this grid is shown in listing 5.1.

Figure 5.2 A simple grid you'll construct with three columns and two rows



Create a new page and link it to a new stylesheet. Add the code in the following listing to your page. In the code, I've added the letters *a* through *f* so it's apparent where each element ends up in the grid.

Listing 5.1 A grid with six items

```
<div class="grid">    #A
  <div class="a">a</div>     #B
  <div class="b">b</div>     #B
  <div class="c">c</div>     #B
  <div class="d">d</div>     #B
```

```
  <div class="e">e</div>     #B
  <div class="f">f</div>     #B
</div>
```

As with flexbox, grid layout applies to two levels of the DOM hierarchy. An element with `display: grid` becomes a *grid container*. Its child elements then become *grid items*.

Next, you'll apply a few new properties to define the specifics of the grid. Add the styles from this listing to your stylesheet.

Listing 5.2 Laying out a basic grid

```
.grid {
  display: grid;          #A
  grid-template-columns: 1fr 1fr 1fr;     #B
  grid-template-rows: 1fr 1fr;     #C
  gap: 0.5em;    #D
}

.grid > * {
  background-color: darkgray;
  color: white;
  padding: 2em;
  border-radius: 0.5em;
}
```

This code renders six equal-sized boxes in three columns (figure 5.2). A number of new things are going on here. Let's take a closer look at them.

First, you've applied `display: grid` to define a grid container. The container behaves like a block display element, filling 100% of the available width. Although not shown in this listing, you could also use the value `inline-grid`; in which case, the element will flow inline and will only be as wide as is necessary to contain its children. You'll most likely not use inline-grid as often.

Next come the new properties: `grid-template-columns` and `grid-template-rows`. These define the size of each of the columns and rows in the grid. This example uses a new unit, `fr`, which represents each column's (or row's) *fraction unit*. This unit behaves much like the `flex-grow` factor in

flexbox. The declaration `grid-template-columns: 1fr 1fr 1fr` declares three columns with an equal size.

You don't necessarily have to use fraction units for each column or row. You can also use other measures such as px, em, or percent. Or, you could mix and match. For instance, `grid-template-columns: 300px 1fr` would define a fixed-size column of 300-px followed by a second column that will grow to fill the rest of the available space. A 2-fr column would be twice as wide as a 1-fr column.

The `gap` property defines the amount of space to add to the gutter between each grid cell, just like with flexbox. You can optionally provide two values to specify vertical and horizontal spacing individually (for example, `gap: 0.5em 1em`).

Note

When the grid specification was first finalized, the `gap` property was named `grid-gap`, so some early examples of grid use `grid-gap`, which works just the same way. The more general `gap` property was added later and flexbox was updated to support it as well.

I encourage you to experiment with these values to see how they affect the final layout. Add new columns or change their widths. Add or remove grid items. Continue to experiment with the other layouts throughout this chapter. This will be the best way to get the hang of things.

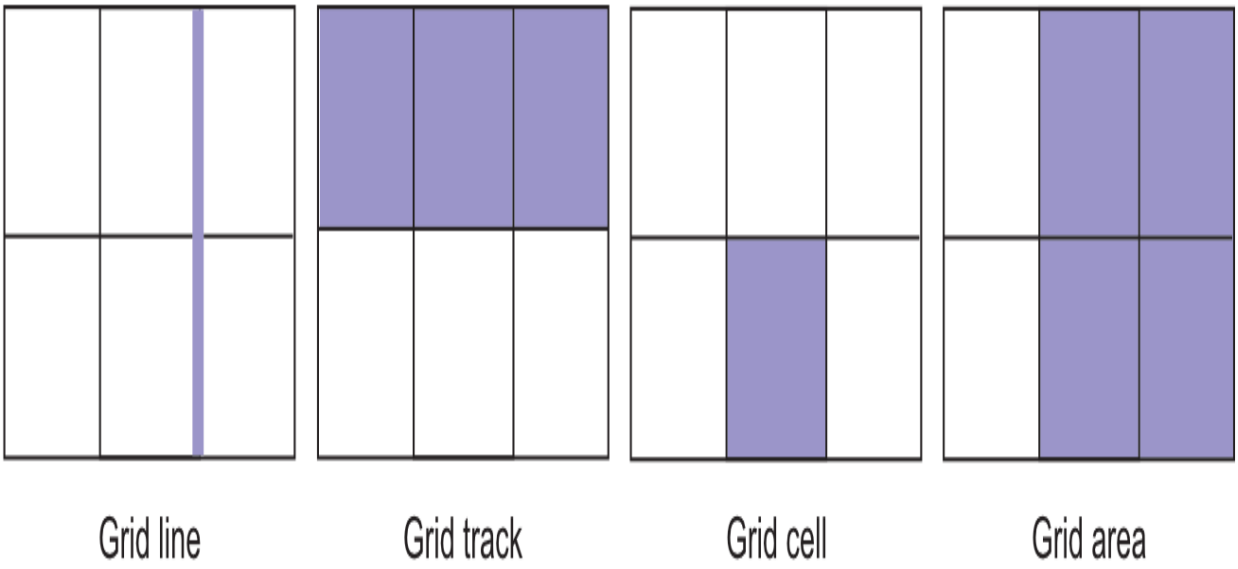
5.2 Anatomy of a grid

It's important to understand the various parts of a grid. I've already mentioned grid containers and grid items, which are the elements that make up the grid. Four other important terms to know are illustrated in figure 5.3.

- *Grid line*—These make up the structure of the grid. A grid line can be vertical or horizontal and lie on either side of a row or column. The `gap`, if defined, lies atop the grid lines.
- *Grid track*—A grid track is the space between two adjacent grid lines. A grid has horizontal tracks (rows) and vertical tracks (columns).

- *Grid cell*—A single space on the grid, where a horizontal grid track and a vertical grid track overlap.
- *Grid area*—A rectangular area on the grid made up by one or more grid cells. The area is between two vertical grid lines and two horizontal grid lines.

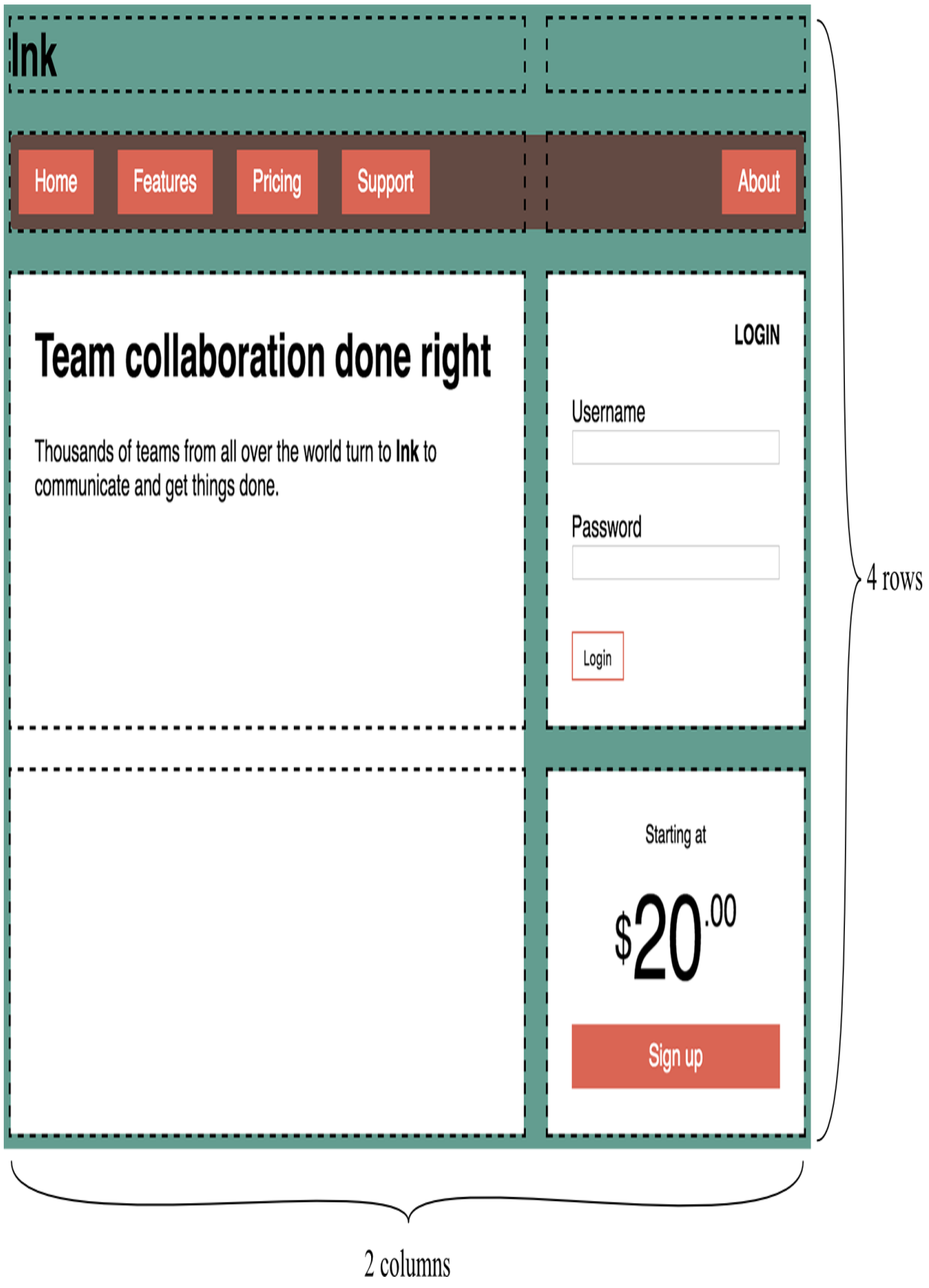
Figure 5.3 The parts of a grid



You'll refer to these parts of the grid as you build grid layouts. For instance, declaring `grid-template-columns: 1fr 1fr 1fr` defines three vertical *grid tracks* of equal width. It also defines four vertical *grid lines*: one down the left edge of the grid, two more between each grid track, and one more along the right edge.

In the previous chapter, you built a page using flexbox. Let's take another look at that design and consider how you could implement it using grid layout. The design is shown in figure 5.4. I've added dashed lines to indicate the location of each grid cell. Notice that some of the sections span multiple cells—filling a larger *grid area*.

Figure 5.4 Page layout created with grid. The dashed lines are added to indicate location of each grid cell.


Ink
Home
Features
Pricing
Support
About
Team collaboration done right
Thousands of teams from all over the world turn to Ink to communicate and get things done.
LOGIN
Username
Password
Login
Starting at
$20.00
Sign up
4 rows
2 columns

This grid has two columns and four rows. The top two horizontal grid tracks are each dedicated to the page title (Ink) and the main navigational menu. The main area fills the remaining two cells in the first vertical track, and the two sidebar tiles are each placed in one of the remaining cells in the second vertical track.

Tip

Your design doesn't need to fill every cell of the grid. Leave a cell empty where you want to add whitespace.

When you built this page using flexbox, you had to nest the elements in a certain way. You used one flexbox to define columns and nested another flexbox inside it to define rows (listing 5.1). To build this layout with grid you'll use a different HTML structure: You'll flatten the HTML. Each item you place on the grid must be a child of the main grid container. The new markup is shown next. Create a new page (or modify your page from chapter 5) to match this listing.

Listing 5.3 HTML structure for a grid layout

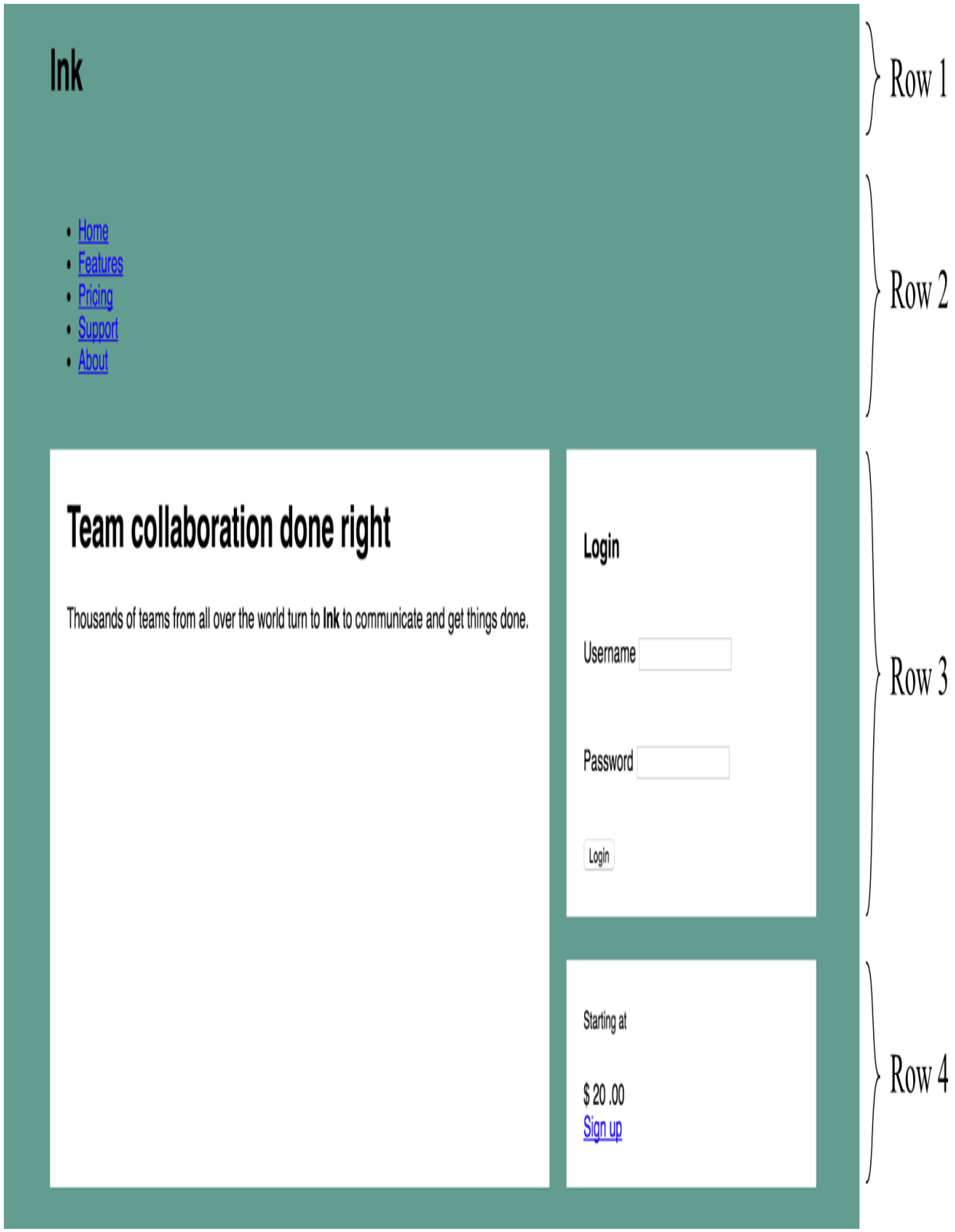
```
<body>
  <div class="container">     #A
    <header>                                #B
      <h1 class="page-heading">Ink</h1>
    </header>

    <nav>                                   #B
      <ul class="site-nav">
        <li><a href="/">Home</a></li>
        <li><a href="/features">Features</a></li>
        <li><a href="/pricing">Pricing</a></li>
        <li><a href="/support">Support</a></li>
        <li class="nav-right">
          <a href="/about">About</a>
        </li>
      </ul>
    </nav>

    <main class="main tile">                    #B
      <h1>Team collaboration done right</h1>
      <p>Thousands of teams from all over the
        world turn to <b>Ink</b> to communicate
```

```
        and get things done.</p>
    </main>

    <div class="sidebar-top tile">                #B
      <form class="login-form">
        <h3>Login</h3>
        <p>
          <label for="username">Username</label>
          <input id="username" type="text"
            name="username"/>
        </p>
        <p>
          <label for="password">Password</label>
          <input id="password" type="password"
            name="password"/>
        </p>
        <button type="submit">Login</button>
      </form>
    </div>

    <div class="sidebar-bottom tile centered stack">     #B
      <small>Starting at</small>
      <div class="cost">
        <span class="cost-currency">$</span>
        <span class="cost-dollars">20</span>
        <span class="cost-cents">.00</span>
      </div>
      <a class="cta-button" href="/pricing">
        Sign up
      </a>
    </div>
  </div>
</body>
```

This version of the page has placed each section of the page as a grid item: the header, the menu (nav), the main, and the two sidebars. I've also added the `tile` class to the main and the two sidebars, as this class provides the white background color and the padding that these elements have in common.

Let's apply grid layout to the page, and put each section in place. We'll pull in a lot of styles from the version in chapter 5 momentarily, but first let's get a general shape of the page in place. After building the basic grid, the page will appear as in figure 5.5.

Figure 5.5 Page with basic grid structure in place



Create an empty stylesheet and link to it from the page. Add this listing to the new stylesheet. This code introduces a few new concepts, which I'll walk you through in a bit.

Listing 5.4 Applying a top-level page layout using grid

```
*,
::before,
::after {
  box-sizing: border-box;
}

:root {
  --gap-size: 1.5rem;
}

body {
  background-color: #709b90;
  font-family: Helvetica, Arial, sans-serif;
}

.stack > * + * {
  margin-block-start: 1.5em;
}

.container {
  display: grid;
  grid-template-columns: 2fr 1fr;     #A
  grid-template-rows: repeat(4, auto);  #B
  gap: var(--gap-size);
  max-inline-size: 1080px;
  margin-inline: auto;
}

header,
nav {
  grid-column: 1 / 3;   #C
  grid-row: span 1;      #D
}

.main {
  grid-column: 1 / 2;         #E
  grid-row: 3 / 5;            #E
}

.sidebar-top {
```

```
  grid-column: 2 / 3;         #E
  grid-row: 3 / 4;            #E
}

.sidebar-bottom {
  grid-column: 2 / 3;         #E
  grid-row: 4 / 5;            #E
}

.tile {
  padding: 1.5em;
  background-color: #fff;
}

.tile > :first-child {
  margin-top: 0;
}
```

This listing provides a number of new concepts. Let's take them one at a time.

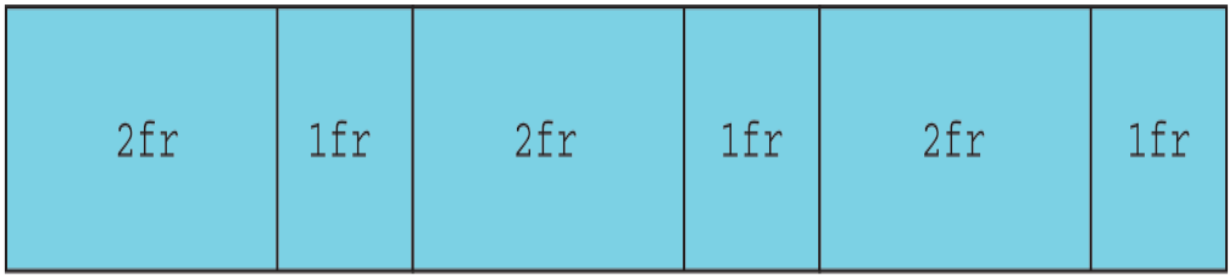
You've set the grid container and defined its grid tracks using `grid-template-columns` and `grid-template-rows`. The columns are defined using the fraction units 2 fr and 1 fr, so the first column will grow twice as much as the second. The rows use something new, the `repeat()` function. This function provides a shorthand for declaring multiple grid tracks.

The declaration `grid-template-rows: repeat(4, auto)` defines four horizontal grid tracks of height `auto`. It's equivalent to `grid-template-rows: auto auto auto auto`. The track size of `auto` will grow as necessary to the size of its content.

You can also define a repeating pattern with the `repeat()` notation. For instance, `repeat(3, 2fr 1fr)` defines six grid tracks by repeating the pattern three times, resulting in `2fr 1fr 2fr 1fr 2fr 1fr`. Figure 5.6 illustrates the resulting columns.

Figure 5.6 Using the `repeat()` function to define a repeating pattern in a template definition

```
grid-template-columns: repeat(3, 2fr 1fr);
```



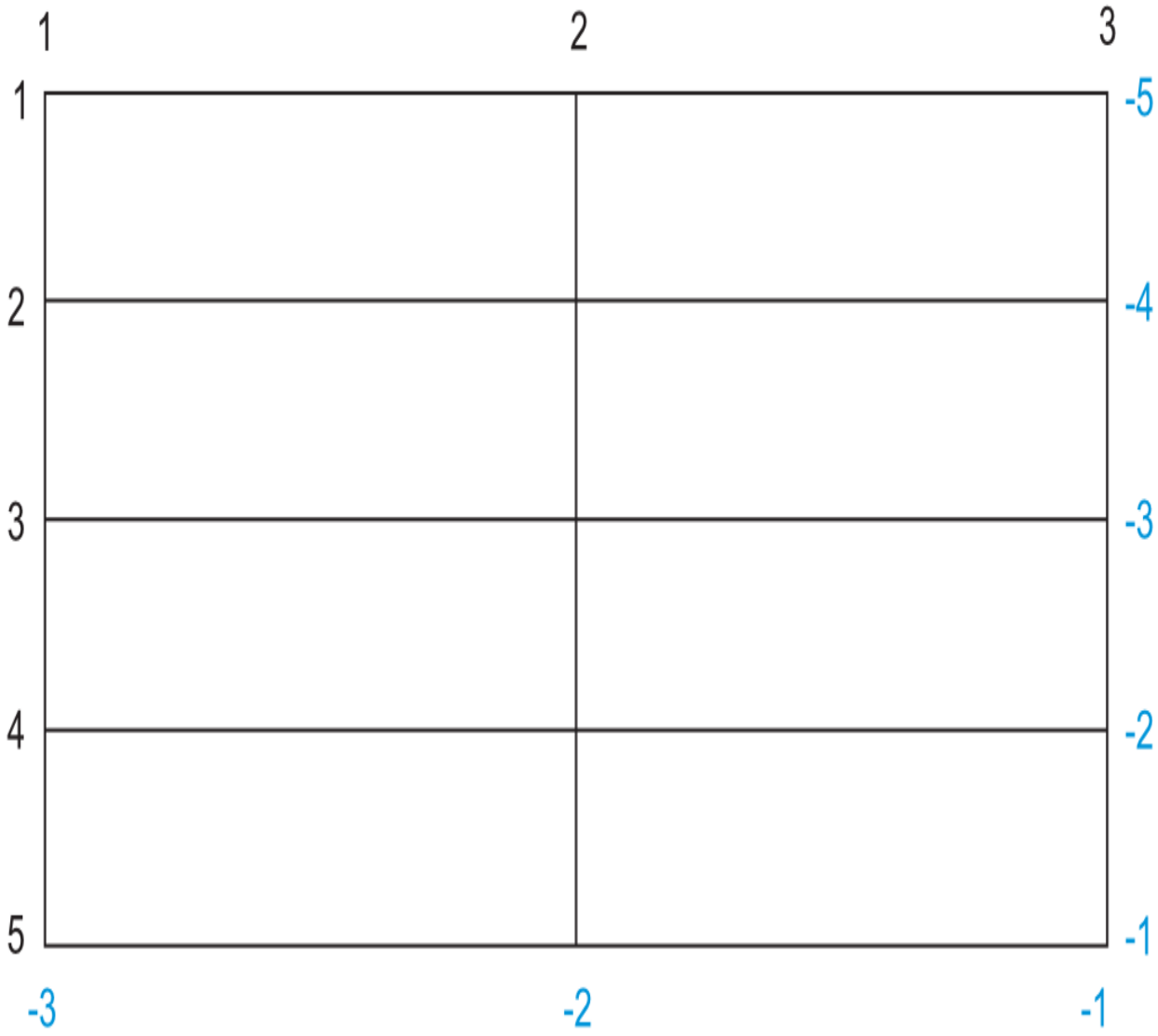

```
grid-template-columns: 1fr repeat(3, 3fr) 1fr;
```

Or you can use `repeat()` as part of a longer pattern. `grid-template-columns: 1fr repeat(3, 3fr) 1fr`, for instance, defines a 1 fr column followed by three 3 fr columns then another 1 fr column (or `1fr 3fr 3fr 3fr 1fr`). As you can see, the longhand is a bit tricky to parse visually, which is why the `repeat()` shorthand comes in handy.

5.2.1 Numbering grid lines

With the grid tracks defined, the next portion of the code places each grid item into a specific location on the grid. The browser assigns numbers to each grid line in a grid, as shown in figure 5.7. The CSS uses these numbers to indicate where each item should be placed.

Figure 5.7 Grid lines are numbered beginning with 1 on the top left. Negative numbers refer to the position from the bottom right.



You can use the grid numbers to indicate where to place each grid item using the `grid-column` and `grid-row` properties. If you want a grid item to span from grid line 1 to grid line 3, you'll apply `grid-column: 1 / 3` to the element. Or, you can apply `grid-row: 3 / 5` to a grid item to make it span from the horizontal grid line 3 to grid line 5. These two properties together specify the grid area you want for an element.

In your page, several grid items are placed this way:

```
.main {
  grid-column: 1 / 2;
  grid-row: 3 / 5;
}
```

```
.sidebar-top {
  grid-column: 2 / 3;
  grid-row: 3 / 4;
}

.sidebar-bottom {
  grid-column: 2 / 3;
  grid-row: 4 / 5;
}
```

This code places the `main` in the first column (between grid lines 1 and 2), spanning the third and fourth rows (between grid lines 3 and 5). It places each sidebar tile in the right column (between grid lines 2 and 3), stacked atop each other in the third and fourth rows.

Note

These properties are in fact shorthand properties: `grid-column` is short for `grid-column-start` and `grid-column-end`; `grid-row` is short for `grid-row-start` and `grid-row-end`. The forward slash is only needed in the shorthand version to separate the two values. The space before and after the slash is optional.

The ruleset that places the `header` and `nav` at the top of the page is a little bit different. Here I've used the same ruleset to target both:

```
header,
nav {
  grid-column: 1 / 3;
  grid-row: span 1;
}
```

This example uses `grid-column` as you've seen previously, making the grid item span the full width of the grid. You can also specify `grid-row` and `grid-column` using a special keyword, `span` (used in this example for `grid-row`). This tells the browser that the item will span one grid track. I didn't specify an explicit row with which to start or end, so the grid item will be placed automatically using the grid item *placement algorithm*. The placement algorithm will position items to fill the first available space on the

grid where they fit; in this case, the first and second rows. We'll look closer at auto-placement later in the chapter.

5.2.2 Working together with flexbox

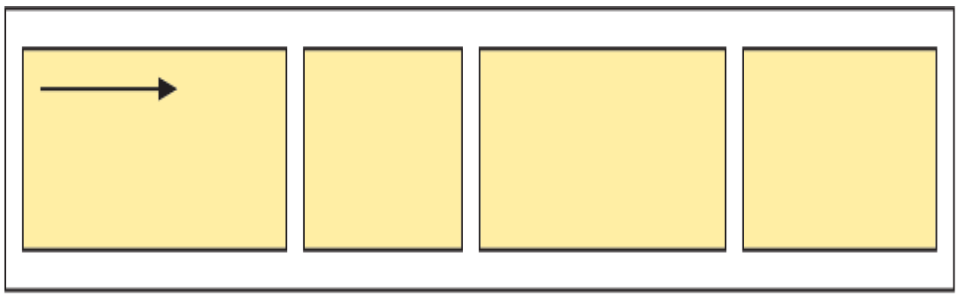
After learning about grid, developers often ask about flexbox. Specifically, are these two competing layout methods? The answer is no; they're complementary. They were largely developed in conjunction. Although there's some overlap in what each can accomplish, they each shine in different scenarios. Choosing between flexbox and grid for a piece of a design is going to come down to your particular needs. The two layout methods have two important distinctions:

- Flexbox is basically one-dimensional, whereas grid is two-dimensional.
- Flexbox works from the content out, whereas grid works from the layout in.

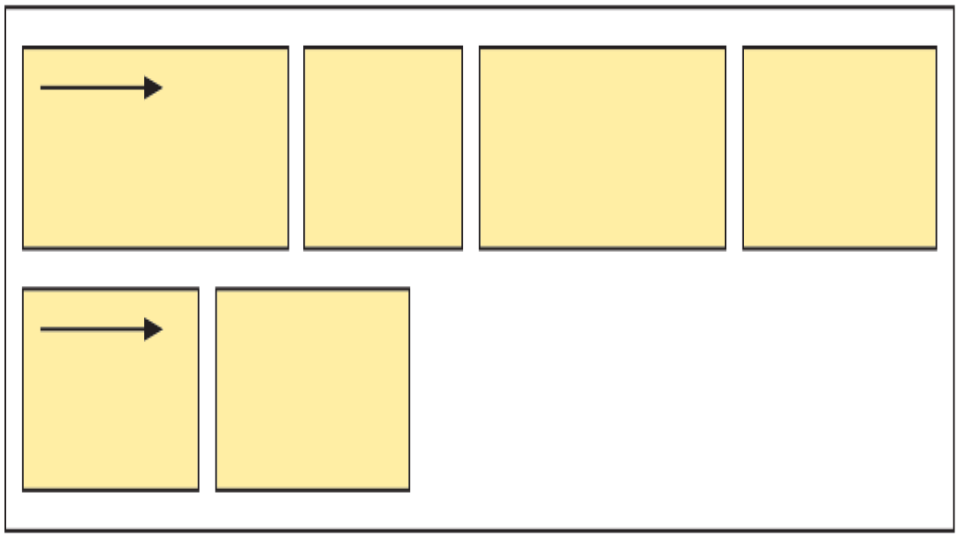
Because flexbox is one-dimensional, it's ideal for rows (or columns) of similar elements. It supports line wrapping using `flex-wrap`, but there's no way to align items in one row with those in the next. Grid, on the contrary, is two-dimensional. It's intended to be used in situations where you want to align items in one track with those in another. This distinction is illustrated in figure 5.8.

Figure 5.8 Flexbox aligns items in one direction, while grid aligns items in two directions.

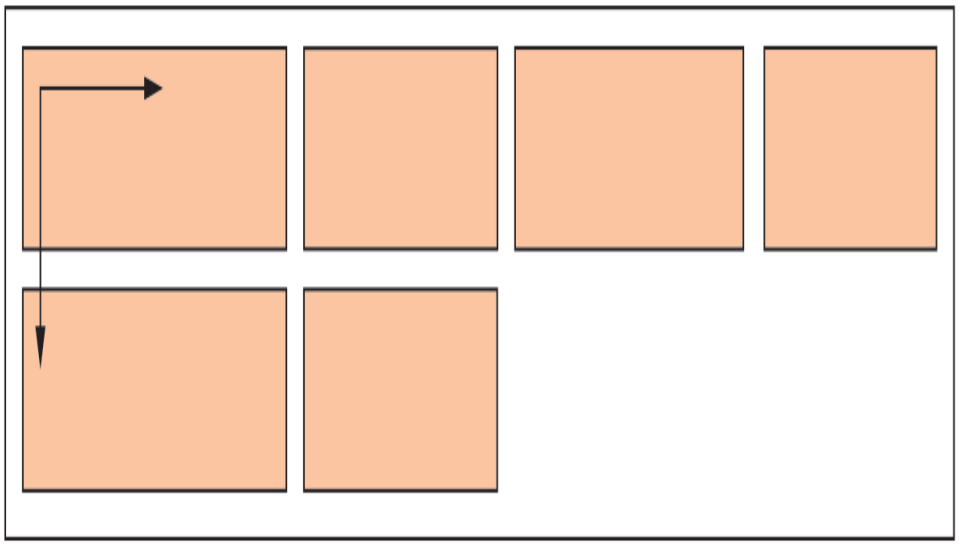
Flexbox:
one-dimensional
alignment



If lines wrap, items
in one row don't
necessarily align with
items in another row



Grid:
two-dimensional
alignment



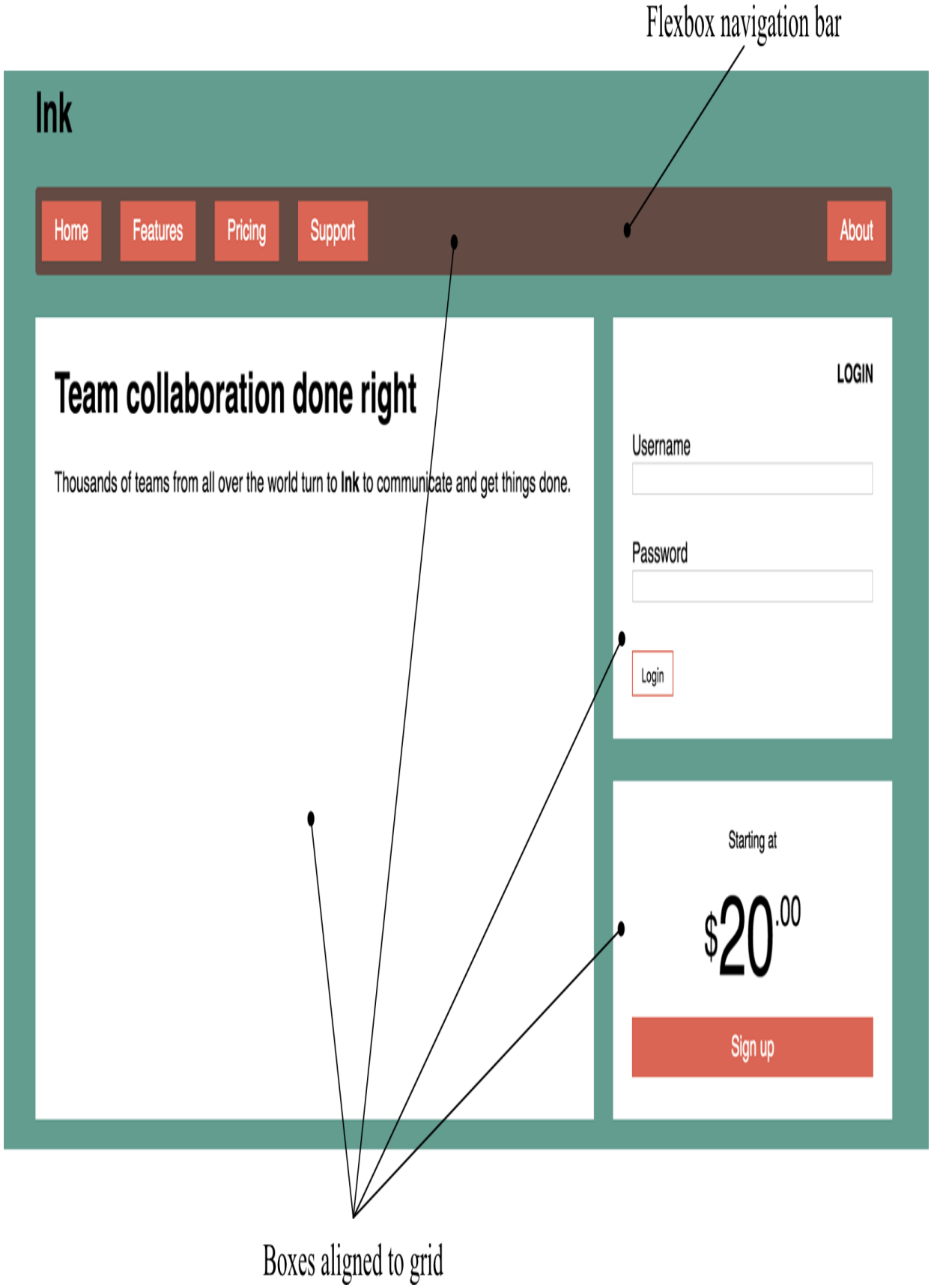
The second major distinction, as articulated by CSS WG member Rachel Andrew, is that flexbox works from the content out, whereas a grid works from the layout in. Flexbox lets you arrange a series of items in a row or column, but their sizes don’t need to be explicitly set. Instead, the content determines how much space each item needs.

With a grid, however, you are first and foremost describing a layout, then placing items into that structure. While the content of each grid item has the ability to influence the size of its grid track, this will affect the size of the entire track and, therefore, the size of other grid items in the track.

You’ve positioned the main regions of the page using a grid because you want the contents to adhere to the grid as it is defined. But for some other items on the page, such as the navigation menu, you can allow the contents to have a greater influence on the outcome; that is, items with more text can be wider, and items with less text can be narrower. It’s also a horizontal (one-dimensional) layout. For these reasons, flexbox is a more appropriate solution for these items. Let’s style these items using flexbox to finish the page.

Figure 5.9 shows the page with a top navigation menu that consists of a list of links aligned horizontally. We’ll also use flexbox for the stylized pricing number on the lower right. After adding these and a few other styles, we’ll arrive at the page’s final look and feel.

Figure 5.9 Fully styled page


Flexbox navigation bar
Ink
Home
Features
Pricing
Support
About
Team collaboration done right
Thousands of teams from all over the world turn to Ink to communicate and get things done.
LOGIN
Username
Password
Login
Starting at
$20.00
Sign up
Boxes aligned to grid

The styles to do this are identical to those from the styles in chapter 4, minus the high-level layout you've applied using grid (listing 5.4). I've repeated them in the next listing. Add this to your stylesheet.

Listing 5.5 Remaining styling for the page

```
.page-heading {
  margin: 0;
}

.site-nav {
  display: flex;    #A
  gap: var(--gap-size);
  margin: 0;
  padding: 0.5em;
  background-color: #5f4b44;
  list-style-type: none;
}

.site-nav > li > a {
  display: block;
  padding: 0.5em 1em;
  background-color: #cc6b5a;
  color: white;
  text-decoration: none;
}

.site-nav > .nav-right {
  margin-inline-start: auto;
}

.login-form h3 {
  margin: 0;
  font-size: 0.9em;
  font-weight: bold;
  text-align: right;
  text-transform: uppercase;
}

.login-form input:not([type="checkbox"]):not([type="radio"]) {
  display: block;
  width: 100%;
}

.login-form button {
  margin-block-start: 1em;
```

```
  border: 1px solid #cc6b5a;
  background-color: white;
  padding: 0.5em 1em;
  cursor: pointer;
}

.centered {
  text-align: center;
}

.cost {
  display: flex;      #B
  justify-content: center;
  align-items: center;
  line-height: 0.7;
}

.cost-currency {
  font-size: 2rem;
}
.cost-dollars {
  font-size: 4rem;
}
.cost-cents {
  font-size: 1.5rem;
  align-self: flex-start;
}

.cta-button {
  display: block;
  background-color: #cc6b5a;
  color: white;
  padding: 0.5em 1em;
  text-decoration: none;
}
```

When your design calls for an alignment of items in two dimensions, use a grid. When you're only concerned with a one-directional flow, use flexbox. In practice, this will often (but not always) mean grid makes the most sense for a high-level layout of the page, and flexbox makes more sense for certain elements within each grid area. As you continue to work with both, you'll begin to get a feel for which is appropriate in various instances.

Note

Both grid and flexbox prevent margin collapsing between grid items or flex items. With the addition of a gap, margins added by user agent styles can sometimes produce too much space between items. This is why you had to reset margins to 0 in a couple places on this page.

5.3 Alternate syntaxes

There are two other alternate syntaxes for laying out grid items: named grid lines and named grid areas. Choosing between them is a matter of preference. In some designs, one syntax may be easier to read and understand than the others. Let's look at both.

5.3.1 Naming grid lines

Sometimes it can be a bit tricky to keep track of all the numbered grid lines, especially when working with a lot of grid tracks. To make this easier, you can name the grid lines and use the names instead of numbers. When declaring grid tracks, place a name in brackets to name a grid line between any two tracks.:

```
grid-template-columns: [start] 2fr [center] 1fr [end];
```

This declaration defines a two-column grid with three vertical grid lines named start, center, and end. You can then reference these names instead of the numbers when placing grid items in your grid. For example:

```
grid-column: start / center;
```

This declaration places a grid item so it spans from grid line 1 (start) to grid line 2 (center). You can also provide multiple names for the same grid line as shown in this example (I've added line breaks to aid readability):

```
grid-template-columns:
  [left-start] 2fr
  [left-end right-start] 1fr
  [right-end];
```

In this declaration, grid line 2 is named both left-end and right-start. You can then use either of these names when placing a grid item. This declaration

allows for another trick here as well: by naming grid lines left-start and left-end, you've defined an area called left that spans between them. The `-start` and `-end` suffixes act as a sort of keyword defining an area in between. If you apply `grid-column: left` to an element, it'll span from `left-start` to `left-end`.

The CSS in the next listing uses named grid lines to lay out the page. This produces the same result as the approach in listing 5.4. Update this portion of your stylesheet to match.

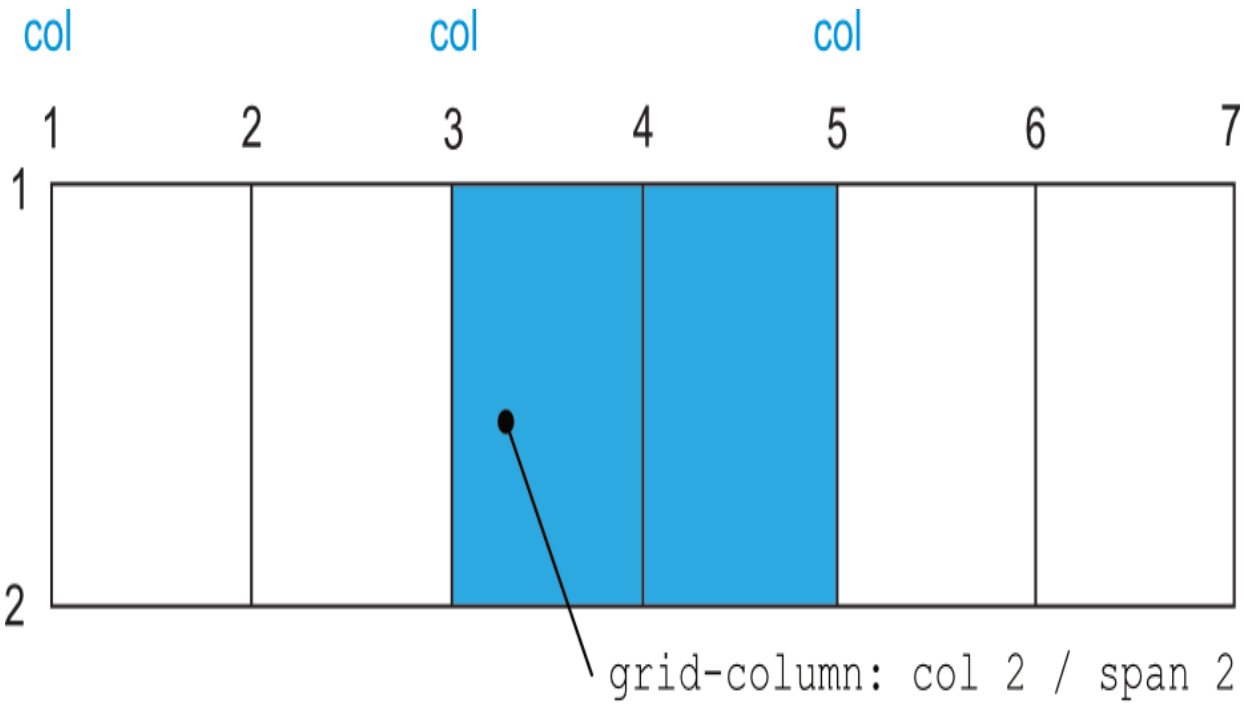
Listing 5.6 Grid layout using named grid lines

```
.container {
  display: grid;
  grid-template-columns:           #A
    [left-start] 2fr               #A
    [left-end right-start] 1fr     #A
    [right-end];                   #A
  grid-template-rows: repeat(4, [row] auto);     #B
  gap: var(--gap-size);
  max-inline-size: 1080px;
  margin-inline: auto;
}

header,
nav {
  grid-column: left-start / right-end;
  grid-row: span 1;
}

.main {
  grid-column: left;               #C
  grid-row: row 3 / span 2;      #D
}

.sidebar-top {
  grid-column: right;      #E
  grid-row: 3 / 4;
}

.sidebar-bottom {
  grid-column: right;      #E
  grid-row: 4 / 5;
}
```

This example places each item into the appropriate grid columns using the named grid lines. It also declares a named horizontal grid line inside the `repeat()` function. Doing this names each horizontal grid line `row` (except for the last one). This may seem peculiar, but it's perfectly valid to use the same name repeatedly. You then place the `main` element so it begins at `row 3` (the third grid line named row) and spans two grid tracks from there.

You can use named grid lines in countless ways. How you use them can vary from one grid to the next, depending on the particular structure of each grid. One possible example is shown in figure 5.10.

Figure 5.10 Placing a grid item at the second "col" grid line, spanning two tracks (col 2 / span 2)



This scenario presents a repeating pattern of two grid columns, naming the grid line before each pair of grid tracks (`grid-template-columns: repeat(3, [col] 1fr 1fr)`). Then you can use named grid lines to place an item in the second set of columns (`grid-column: col 2 / span 2`).

5.3.2 Naming grid areas

Another approach you can take is to name the grid areas. Instead of counting or naming the grid lines, you can use these named areas to position items in the grid. This is done with the `grid-template-areas` property on the grid container and a `grid-area` property on the grid items.

The code in listing 5.7 shows an example of this. Again, this code produces exactly the same result as the previous layout (listings 5.4 and 5.6). It's an alternate syntax that can be used instead. Update your stylesheet to match these styles.

Listing 5.7 Using named grid areas

```
.container {
  display: grid;
  grid-template-areas:
    "title title"       #A
    "nav   nav"         #A
    "main  aside1"      #A
    "main  aside2";     #A
  grid-template-columns: 2fr 1fr;        #B
  grid-template-rows: repeat(4, auto);   #B
  grid-gap: var(--gap-size);
  max-inline-size: 1080px;
  margin-inline: auto;
}

header {
  grid-area: title;        #C
}

nav {
  grid-area: nav;          #C
}

.main {
  grid-area: main;         #C
}

.sidebar-top {
  grid-area: aside1;       #C
}

.sidebar-bottom {
  grid-area: aside2;       #C
}
```

The `grid-template-areas` property lets you draw a visual representation of the grid directly into your CSS, using a sort of “ASCII art” syntax. This declaration provides a series of quoted strings, each one representing a row of the grid, with whitespace between each column.

In this example, the first row is assigned entirely to the grid area `title`. The second row is assigned to `nav`. The left column of the next two rows is assigned to `main`, and each sidebar tile is assigned to `aside1` and `aside2`. Each grid item is then placed into these named areas using the `grid-area` property.

Warning

Each named grid area must form a rectangle. You cannot create more complex shapes like an *L* or a *U*.

You can also leave a cell empty by using a period as its name. For example, this code defines four grid areas surrounding an empty grid cell in the middle:

```
grid-template-areas:
  "top  top    right"
  "left .      right"
  "left bottom bottom";
```

When you build a grid, use whichever syntax is most comfortable for you, given the design: numbered grid lines, named grid lines, or named grid areas. The latter is a favorite for a lot of developers, and it shines when you know exactly where you want to place each grid item.

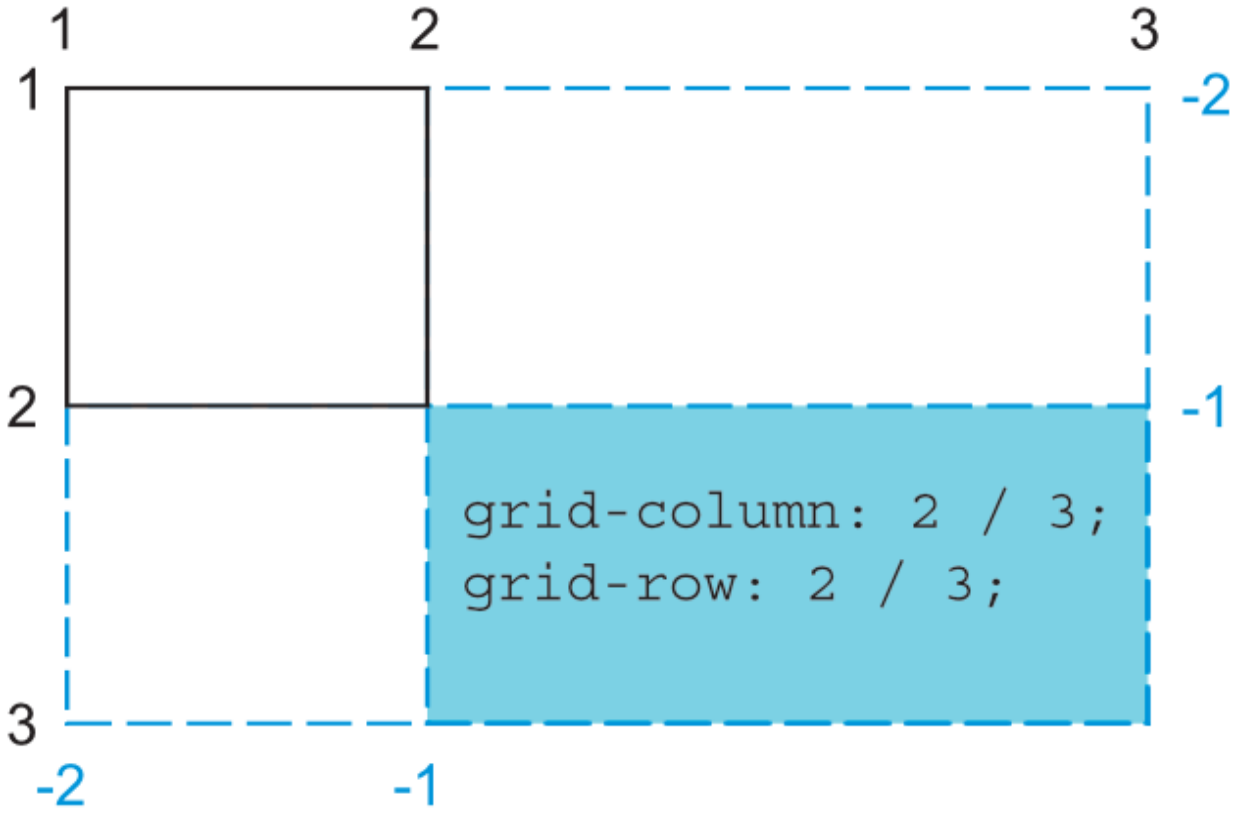
5.4 Explicit and implicit grid

In some instances, you may not know exactly where you want to place each item in the grid. Perhaps you’re working with a large number of grid items and placing each one explicitly is unwieldy. You might even have an unknown number of items populated by a database. In these cases, it’ll probably make more sense to loosely define a grid, and then allow the grid item placement algorithm to fill it for you.

This will require you to rely on an *implicit grid*. When you use the `grid-template-*` properties to define grid tracks, you're creating an *explicit grid*. But grid items can still be placed outside of these explicit tracks; in which case, implicit tracks will be automatically generated, expanding the grid so it contains these elements.

Figure 5.11 illustrates a grid with only one explicit grid track in each direction. When a grid item is placed in the second track (between grid lines two and three), additional tracks are added to include it.

Figure 5.11 If a grid item is placed outside the declared grid tracks, implicit tracks will be added to the grid until it can contain the item.



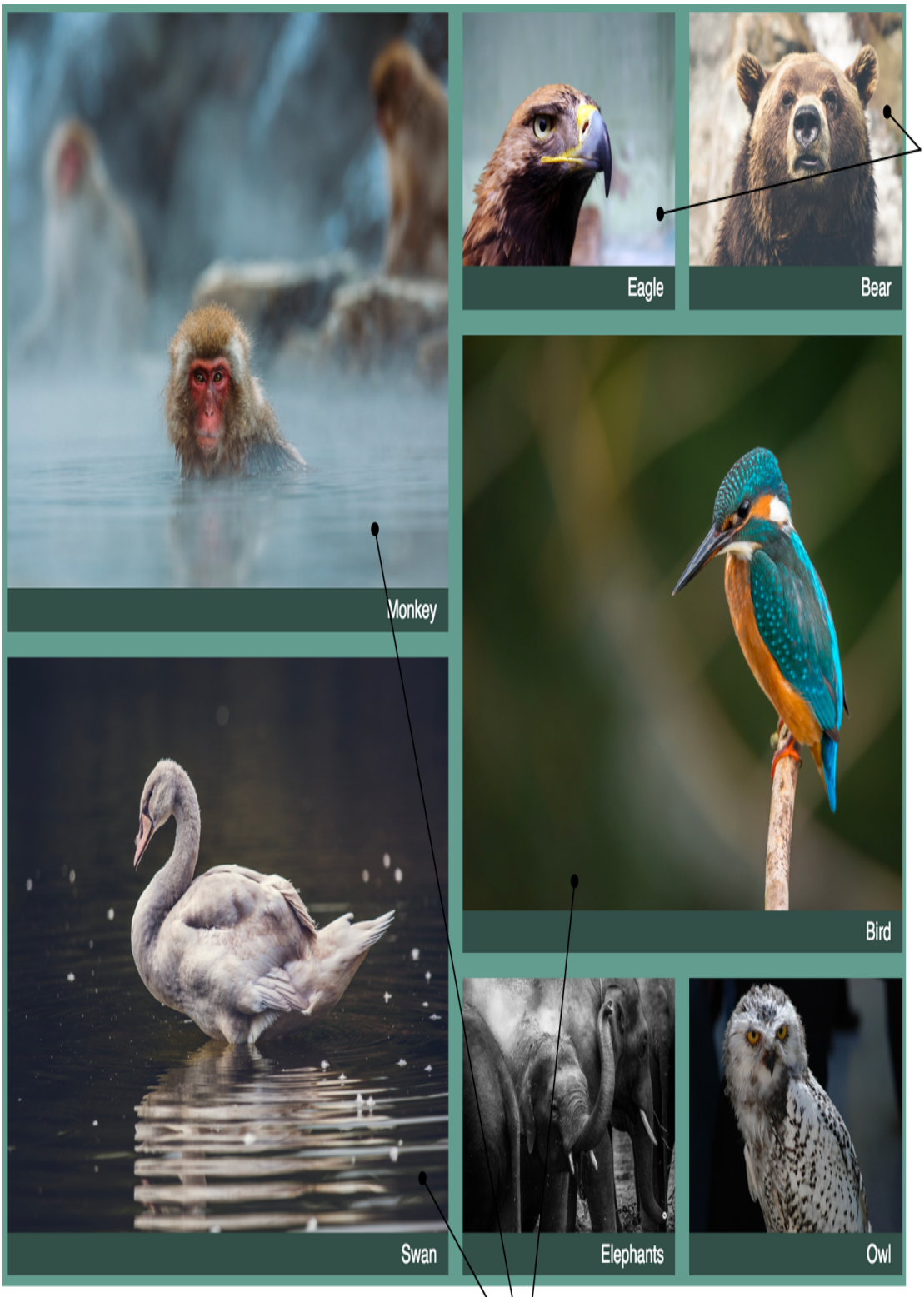
By default, implicit grid tracks will have a size of `auto`, meaning they'll grow to the size necessary to contain the grid item contents. The properties `grid-auto-columns` and `grid-auto-rows` can be applied to the grid container to specify a different size for all implicit grid tracks (for example, `grid-auto-columns: 1fr`).

Note

Implicit grid tracks don’t change the meaning of negative numbers when referencing grid lines. Negative grid-line numbering still begins at the bottom/right of the explicit grid.

Let’s lay out another page using an implicit grid. This will be a photography portfolio, as shown in figure 5.12. For this layout, you’ll set grid tracks for the columns, but the grid rows will be implicit. This way, the page isn’t structured for any specific number of images; it’ll be adaptable for any number of grid items. Any time the images need to wrap onto a new row, another row will be added implicitly.

Figure 5.12 A series of photographs laid out in a grid using implicit grid rows


Monkey
Eagle
Bear
Bird
Swan
Elephants
Owl
Photos in grid cells
Some images selected to span a larger grid area

This is a fun layout because it would be difficult to achieve with flexbox or floats. It showcases the unique power of grids.

To build this, you'll need a new page. Create a blank page and a new stylesheet and link them. The markup for this is shown here. Add it to the page.

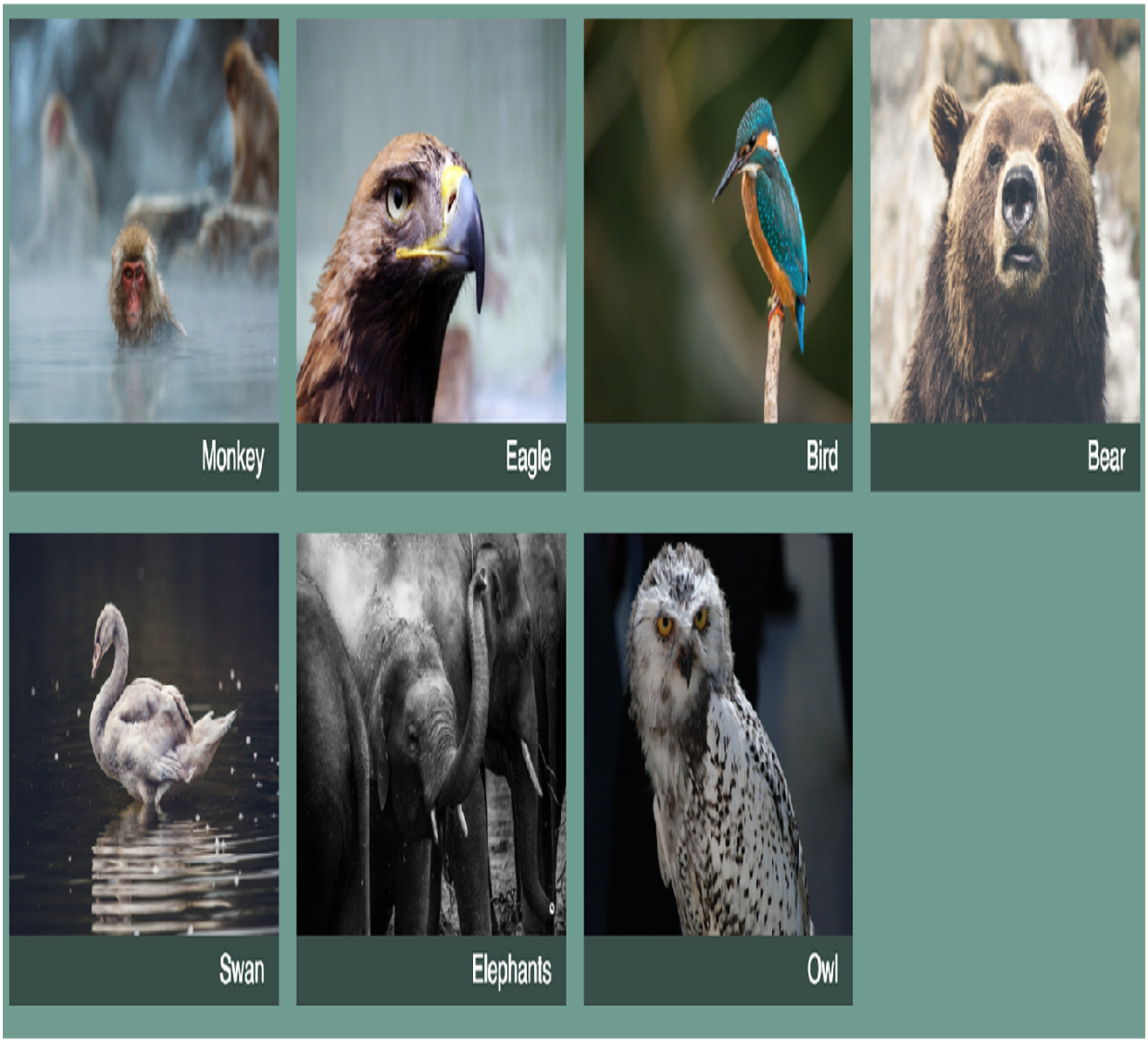
Listing 5.8 Markup for a portfolio

```
<!doctype html>
<head>
  <link href="styles.css" rel="stylesheet" type="text/css" />
</head>
<body>
<div class="portfolio">
  <figure class="featured">             #A
    <img src="images/monkey.jpg" alt="monkey" />
    <figcaption>Monkey</figcaption>
  </figure>
  <figure>
    <img src="images/eagle.jpg" alt="eagle" />    #B
    <figcaption>Eagle</figcaption>                #B
  </figure>
  <figure class="featured">                      #C
    <img src="images/bird.jpg" alt="bird" />
    <figcaption>Bird</figcaption>
  </figure>
  <figure>
    <img src="images/bear.jpg" alt="bear" />
    <figcaption>Bear</figcaption>
  </figure>
  <figure class="featured">                      #C
    <img src="images/swan.jpg" alt="swan" />
    <figcaption>Swan</figcaption>
  </figure>
  <figure>
    <img src="images/elephants.jpg" alt="elephants" />
    <figcaption>Elephants</figcaption>
  </figure>
  <figure>
    <img src="images/owl.jpg" alt="owl" />
    <figcaption>Owl</figcaption>
  </figure>
</div>
</body>
</html>
```

This markup includes a portfolio element (which will be the grid container) and a series of figures (which will be the grid items). Each figure contains an image and a caption. I've added the class `featured` to a few items, which you'll use to make those larger than the other images.

I'll walk you through this in a few phases. First, you'll shape the grid tracks and see the images in a basic grid formation (figure 5.13). After that, you'll enlarge the "featured" images and apply a few other finishing touches.

Figure 5.13 Items automatically placed in grid cells from left to right



The styles for this are shown in listing 5.9. It uses `grid-auto-rows` to specify a 1 fr size for all implicit grid rows, so each row will be the same height. It also introduces two new concepts: `auto-fill` and the `minmax()` function, which I'll explain in a moment. Add these styles to your stylesheet.

Listing 5.9 A grid with implicit grid rows

```
body {
  background-color: #709b90;
  font-family: Helvetica, Arial, sans-serif;
}

.portfolio {
  display: grid;
  grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));
#A
  grid-auto-rows: 1fr;                #B
  grid-gap: 1em;
}

.portfolio > figure {
  margin: 0;          #C
}

.portfolio img {
  max-inline-size: 100%;
}

.portfolio figcaption {
  padding: 0.3em 0.8em;
  background-color: rgb(0 0 0 / 0.5);   #D
  color: #fff;
  text-align: right;
}
```

Sometimes you won't want to set a fixed size on a grid track, but you'll want to constrain it within certain minimum and maximum values. This is where the `minmax()` function comes in. It specifies two values—a minimum size and a maximum size. The browser will ensure the grid track falls between these values. (If the maximum size is smaller than the minimum size, then the maximum is ignored.) By specifying `minmax-(200px, 1fr)`, the browser ensures that all tracks are at least 200 px wide.

The `auto-fill` keyword is a special value you can provide for the `repeat()` function. With this set, the browser will place as many tracks onto the grid as it can fit, without violating the restrictions set by the specified size (the `minmax()` value).

Together, `auto-fill` and `minmax(200px, 1fr)` mean your grid will place as many grid columns as the available space can hold, without allowing any of

them to be smaller than 200 px. And because no track can be larger than 1 fr (our maximum value), all the grid tracks will be the same size.

In figure 5.13, the viewport has room for four columns of 200 px, so that's how many tracks were added. If the screen is wider, more may fit. If it's narrower, then fewer will be created.

Note that `auto-fill` can also result in some empty grid tracks, if there are not enough grid items to fill them all. If you don't want empty grid tracks, you can use the keyword `auto-fit` instead of `auto-fill`. This causes the non-empty tracks to stretch to fill the available space. See http://gridbyexample.com/examples/example37/ for an example of the difference.

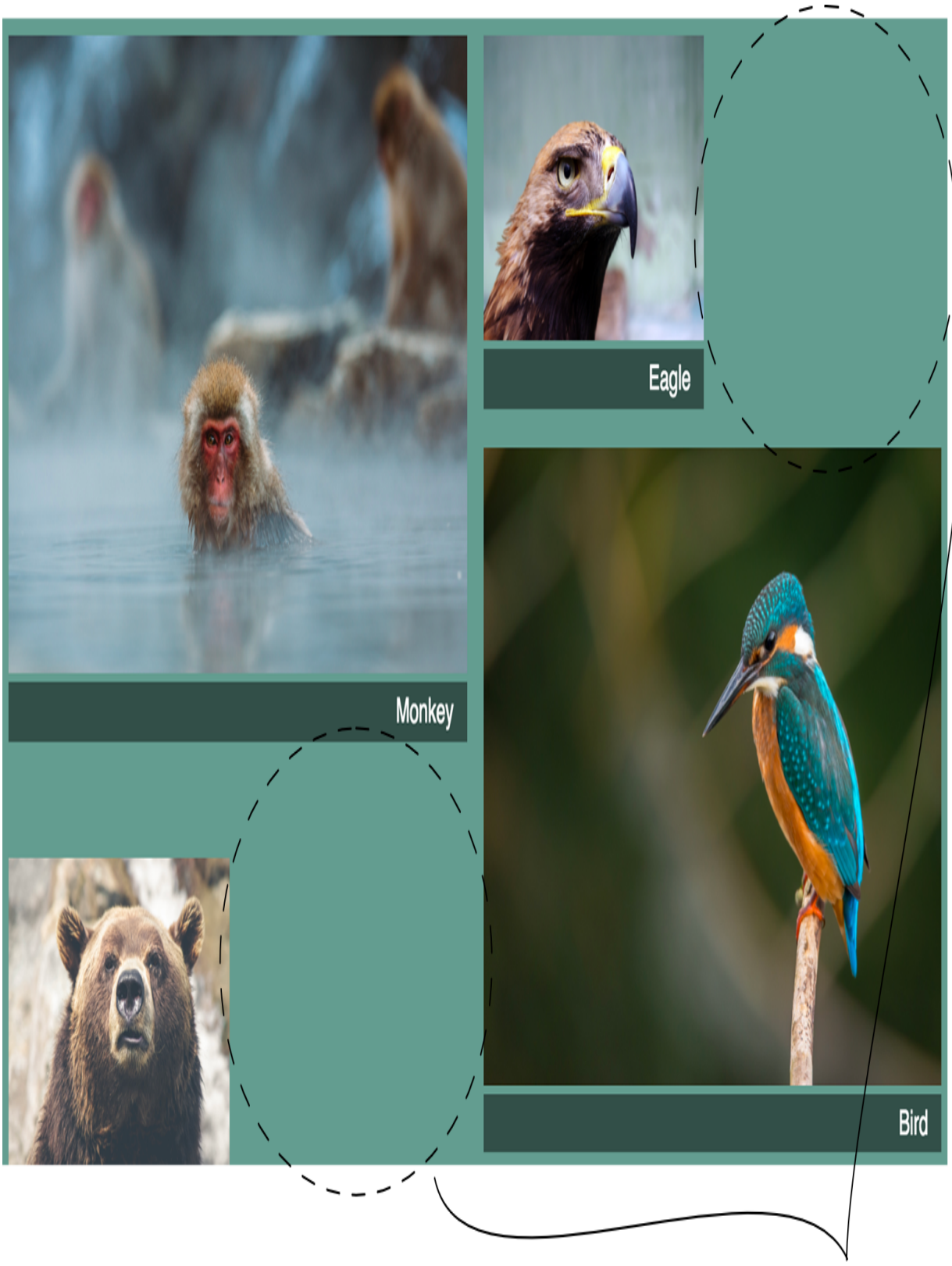
Whether you use `auto-fill` or `auto-fit` depends on whether you want to ensure you get the expected grid track size or whether you want to make certain the width of the entire grid container is filled.

5.4.1 Adding variety

Next, let's add visual interest to your grid by increasing the size of the featured images (the bird and the swan in this example). Each grid item currently fills a 1 x 1 area on the grid. You'll increase the size of featured images to fill a 2 x 2 grid area. You can target these items with the `featured` class and make them span two grid tracks in each direction.

This introduces a problem, however. Depending on the order of the items, increasing the size for some grid items could result in gaps in the grid. Figure 5.14 illustrates these gaps. The bird in this figure is the third item in the grid. But because it's a larger item, it doesn't fit in the space to the right of the second image, the eagle. Instead, it has dropped down to the next grid track.

Figure 5.14 Increasing the size of some grid items introduced gaps in the layout where the large items don't fit.



Empty grid cells where larger items can’t fit

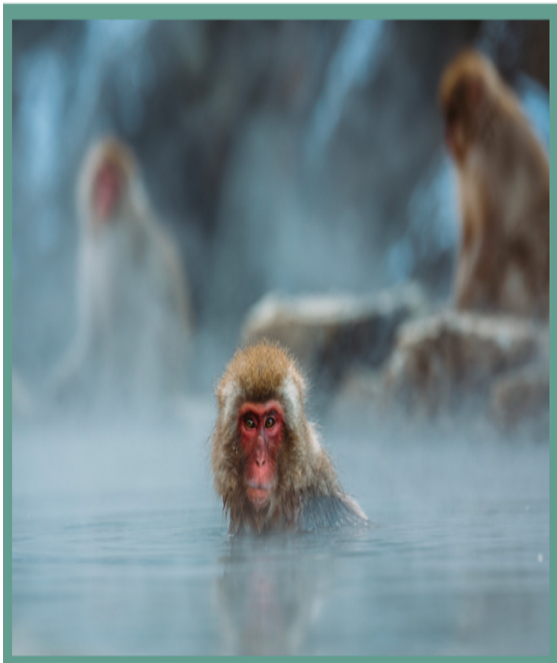
When you don’t specifically position items on a grid, they are positioned automatically by the grid item placement algorithm. By default, this algorithm places grid items column by column, row by row, according to the order of the items in the markup. When an item doesn’t fit in one row (that is, it spans too many grid tracks), the algorithm moves to the next row, looking for space large enough to accommodate the item. In this case, the bird is moved down to the second row, beneath the eagle.

The Grid Layout Module provides another property, `grid-auto-flow`, that can be used to manipulate the behavior of the placement algorithm. Its initial value, `row`, behaves as I’ve described. Given the value `column`, it instead places items in the columns first, moving to the next row only after a column is full.

You can also add the keyword `dense` (for example, `grid-auto-flow: column dense`). This causes the algorithm to attempt to fill gaps in the grid, even if it means changing the display order of some grid items. If you apply this to your page, smaller grid items will “backfill” the gaps created by the larger grid items. The result is shown in figure 5.15.

Figure 5.15 Using dense `grid-auto-flow` allows small grid items to backfill gaps in the grid.

Items that can fit in gaps are moved up to fill them



Monkey

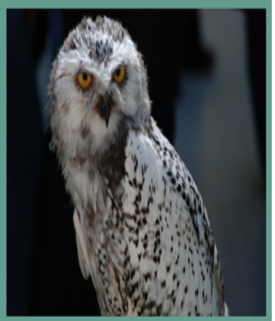
Eagle

Bear

Bird

Swan

Elephants



Owl

With the `dense` auto-flow option, smaller grid items fill the gaps left by larger items. The source order here is still the monkey, eagle, bird, then bear, but the bear is moved into position before the bird, thus filling the gap.

Add the next listing to your stylesheet. This enlarges featured images to fill two grid tracks in each direction and applies a dense auto-flow.

Listing 5.10 Enlarging featured images

```
.portfolio {
  display: grid;
  grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));
  grid-auto-rows: 1fr;
  gap: 1em;
  grid-auto-flow: dense;   #A
}

.portfolio .featured {
  grid-row: span 2;        #B
  grid-column: span 2;     #B
}
```

This listing uses the declaration `grid-auto-flow: dense`, which is equivalent to `grid-auto-flow: row dense`. (In the first part of the value, row is implied because it's the initial value.) Then it targets the featured items and sets them to span two grid tracks in each direction. Note this example uses only the `span` keyword and doesn't expressly place any grid items on a specific grid track. This allows the grid item placement algorithm to position the grid items where it sees fit.

Depending on your viewport size, your screen may not match figure 5.12 exactly. That's because you used `auto-fill` to determine the number of vertical grid tracks. A larger screen will have room for more tracks; a smaller screen will have fewer. I took this screenshot with a viewport about 1,000 px wide, producing four grid tracks. Resize your browser width to various sizes to see how the grid automatically responds, filling available space.

Use caution with a dense auto-flow because items may not be displayed in the same order as they appear in the HTML. This can cause some confusion

for users navigating via the keyboard (Tab key) or a screen reader as those methods navigate according to source order, not display order.

5.4.2 Adjusting grid items to fill the grid track

You now have a fairly complex layout. You didn’t have to do a lot of work to place each item in a precise location, but rather allowed the browser to figure that out for you.

One last issue remains: the larger images aren’t completely filling the grid cells, which leaves a small gap beneath them. Ideally, both the top and bottom edges of each grid item should align with others on the same grid track. Our top edges align, but the bottom edges don’t as shown in figure 5.16.

Figure 5.16 Images are not entirely filling the grid cells, leaving an unwanted gap.

Monkey

Eagle

Bear

Bird

Swan

Elephants

Owl

Some items don’t fill their entire grid cell

Let’s fix that gap. If you recall, each grid item is a `<figure>` that contains two child elements—an image and a caption:

```
<figure class="featured">
  <img src="images/monkey.jpg" alt="monkey" />
  <figcaption>Monkey</figcaption>
</figure>
```

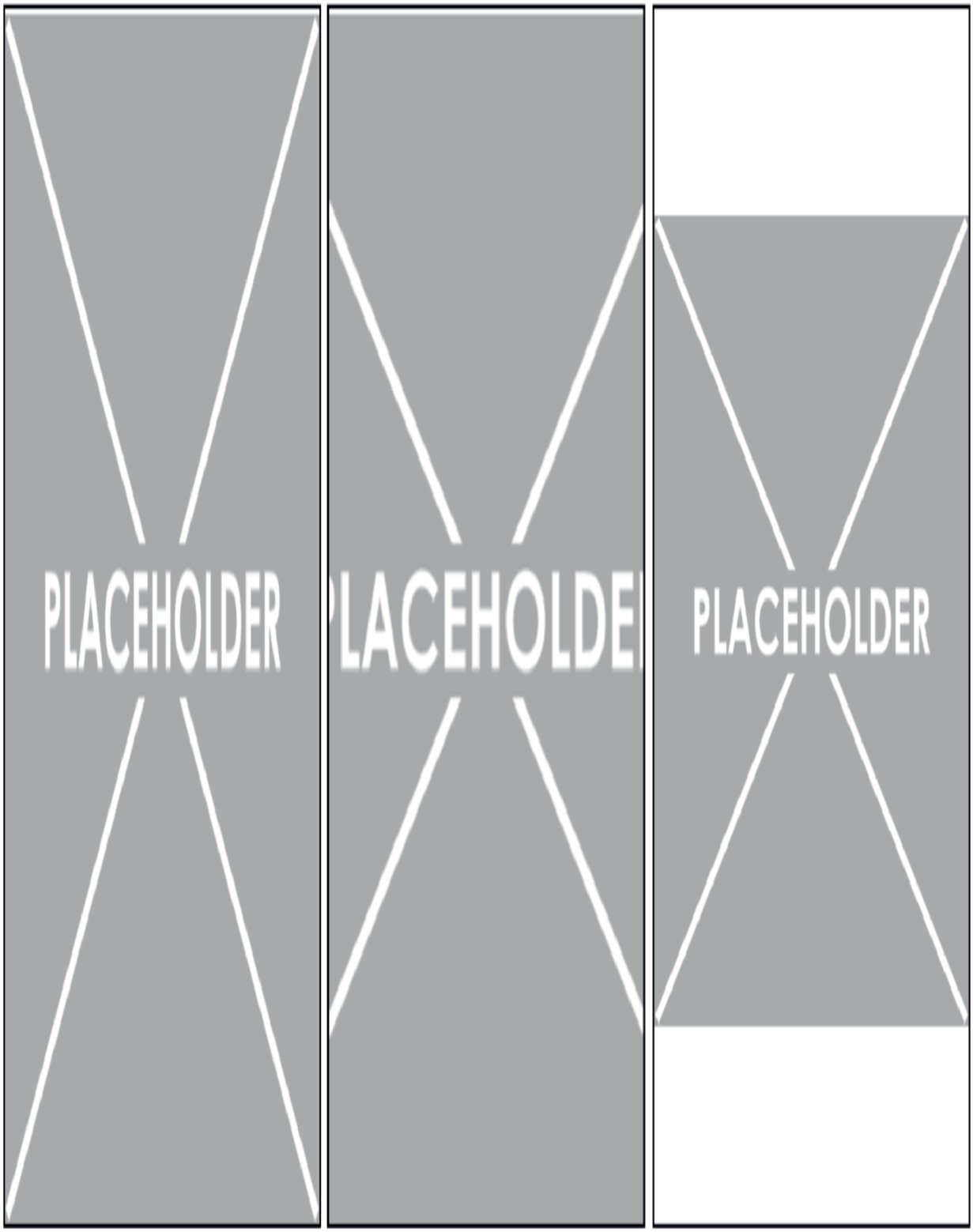
By default, each grid item is stretched to fill the entire grid area, but its child elements are not stretched to fill it, so the grid area has some unused height. An easy way to fix this is with flexbox. In listing 5.11, you’ll make each `<figure>` a flex container with a direction of `column` so items stack vertically, atop one another. You can then apply a flex grow to the image, forcing it to stretch to fill the space.

Stretching an image is problematic, however. This will change its height-to-width ratio, distorting the picture. Fortunately, CSS provides a special property for controlling this behavior, `object-fit`. By default, an `<img>` has an `object-fit` value of `fill`, meaning the entire image will be resized to fill the `<img>` element. You can also set other values to change this.

For example, the `object-fit` property accepts the values `cover` and `contain` (illustrated in figure 5.17). These values tell the browser to resize the image within the rendered box, without distorting its aspect ratio.

- To expand the image to fill the box (resulting in part of the image being cut off), use `cover`.
- To resize the image so that it fits entirely in the box (resulting in empty space within the box), use `contain`.

Figure 5.17 Using `object-fit` to control how an image is rendered in its box



fill (default) cover contain

There’s an important distinction to make here: there is the box (determined by the `<img>` element’s height and width), and there is the rendered image. By default, these are the same size. The `object-fit` property lets you manipulate the size of the rendered image within that box, but the size of the box itself remains unchanged.

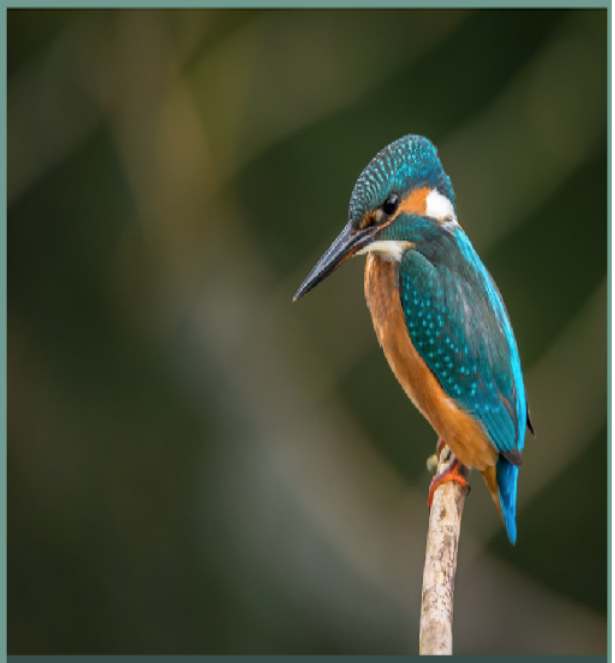
Because you’ll use the `flex-grow` property to stretch the images, you should also apply `object-fit: cover` to prevent the images from being distorted. This will crop off a small bit of the edge of the images, which is a compromise you’ll have to make. The end result is shown in figure 5.18. For a more detailed look at this property, see https://css-tricks.com/on-object-fit-and-object-position/.

Figure 5.18 All images now fill their grid areas and align cleanly.

Monkey

Eagle

Bear



Bird

Swan

Elephants

Owl

Now the top and bottom edges of all the images and their captions align in each grid track. The code for this is shown here. Add it to your stylesheet.

Listing 5.11 Using a column flexbox to stretch images to fill the grid area

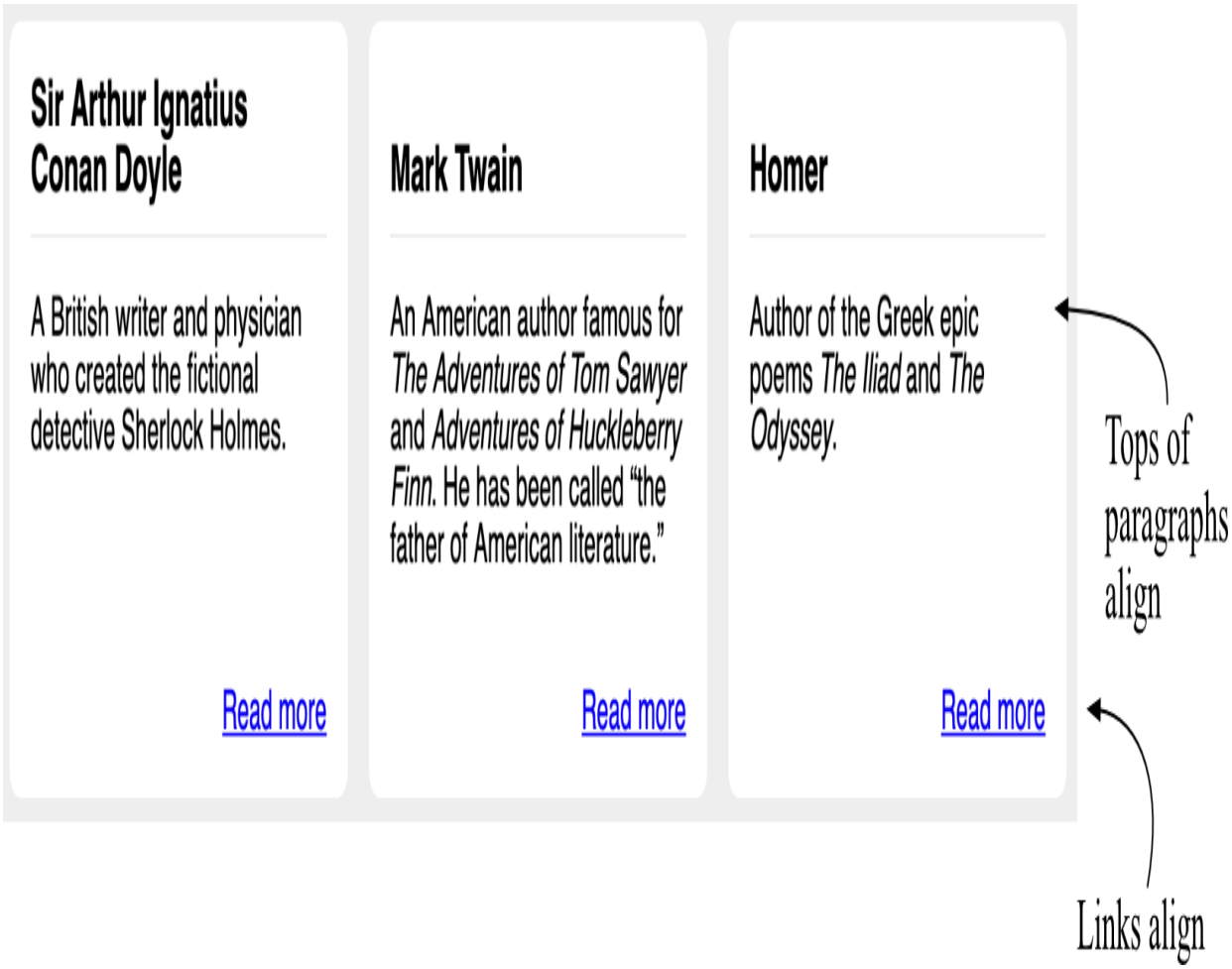
```
.portfolio > figure {
  display: flex;                #A
  flex-direction: column;       #A
  margin: 0;
}

.portfolio img {
  flex: 1;                  #B
  object-fit: cover;         #C
  max-inline-size: 100%;
}
```

This completes the design of your photography portfolio. Everything aligns in a neat grid, and the browser makes decisions for you regarding the number and size of each vertical grid track. Using a dense auto-flow allows the browser to fill in gaps neatly.

5.5 Subgrid

In the previous examples, the grids have been restricted to two levels of the DOM: a grid container and its children. For some designs, you will need to align items that span a greater range of elements. For instance, you may need to align two elements that share a common grandparent element, or give them an equal size. See figure 5.19 for an example of one such design.

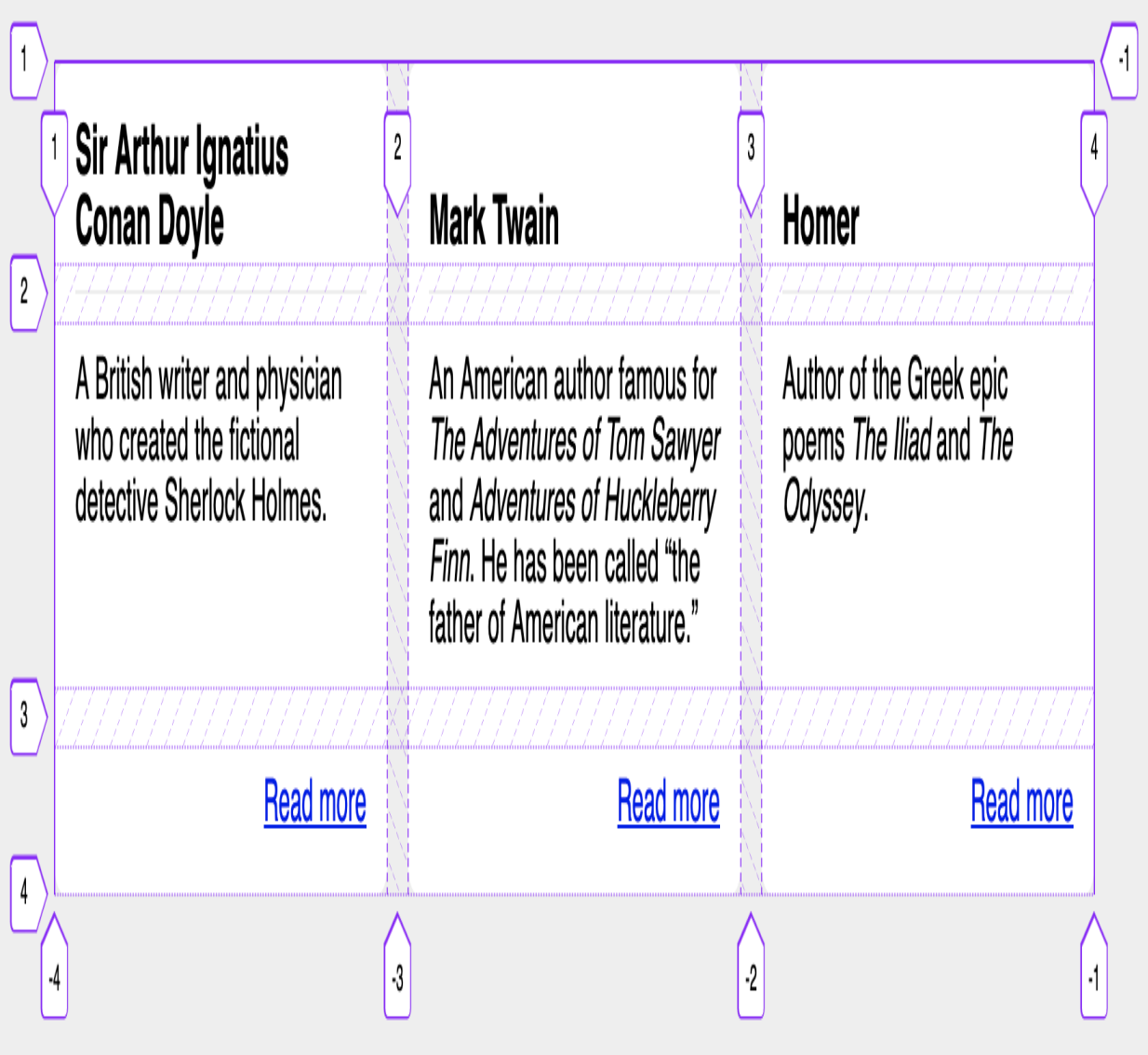
Figure 5.19 Cards with multiple aligned elements



In this design, the cards belong to a grid, so they appear aligned in a row and each have the same height. But the elements within the cards also align to one another. In the cards with shorter headings, extra space at the top brings them in alignment with the longer heading. The paragraphs of each card begin at the same height. And the Read more links are aligned along the bottom.

The best way to accomplish this is with a newer feature called *subgrid*. With subgrid, you can place a grid within a grid, and then position the inner grid's items on the grid lines of the parent grid. In this page, each card spans three rows, and the contents of each card align to each of those rows. Figure 5.20 shows the grid lines and gaps highlighted.

Figure 5.20 Grid lines and gaps shown



You can see here how the same grid defines the positions of the cards as well as the elements inside the cards.

Tip

Figure 5.20 is a screenshot of grid rows as shown in Firefox using DevTools. In most browsers, you can debug a grid like this by clicking the small "grid" label that appears beside the element in the Inspector pane. There is also a similar feature for flexbox.

Be aware that subgrid is a newer feature of CSS, and it is not as broadly supported as grid. At the time of writing, only Firefox and Safari support it,

but Chrome is expected to follow soon. See https://caniuse.com/css-subgrid for the latest support information.

I'll show you how to build this layout. Begin by starting a new page and adding the markup as shown in listing 5.12.

Listing 5.12 HTML for three author cards in a grid

```
<!doctype html>
<html lang="en-US">
<head>
  <link href="styles.css" rel="stylesheet" type="text/css" />
</head>
<body>
  <div class="author-bios">     #A
    <div class="card">            #B
      <h3>Sir Arthur Ignatius Conan Doyle</h3>
      <div>
        <p>
          A British writer and physician who created
          the fictional detective Sherlock Holmes.
        </p>
      </div>
      <div class="read-more"><a href="/conan-doyle">
        Read more
      </a></div>
    </div>
    <div class="card">           #B
      <h3>Mark Twain</h3>
      <div>
        <p>
          An American author famous for <i>The
          Adventures of Tom Sawyer</i> and
          <i>Adventures of Huckleberry Finn</i>.
          He has been called "the father of
          American literature."
        </p>
      </div>
      <div class="read-more"><a href="/mark-twain">
        Read more
      </a></div>
    </div>
    <div class="card">           #B
      <h3>Homer</h3>
      <div>
        <p>
```

```
          Author of the Greek epic poems <i>The
          Iliad</i> and <i>The Odyssey</i>.
        </p>
      </div>
      <div class="read-more"><a href="/homer">
        Read more
      </a></div>
    </div>
  </div>
</body>
</html>
```

As you can see, this layout primarily deals with aligning three levels of DOM elements: the main grid container (`<div class="author-bios">`), its grid items (three `<div class="card">`), and the children elements of the cards.

Begin by some basic styles including background colors and spacing, as well as the top-level grid to put the cards in place. This is shown in listing 5.13. These should mostly be familiar styles at this point, with the possible exception of the grid, if you're still getting comfortable with that.

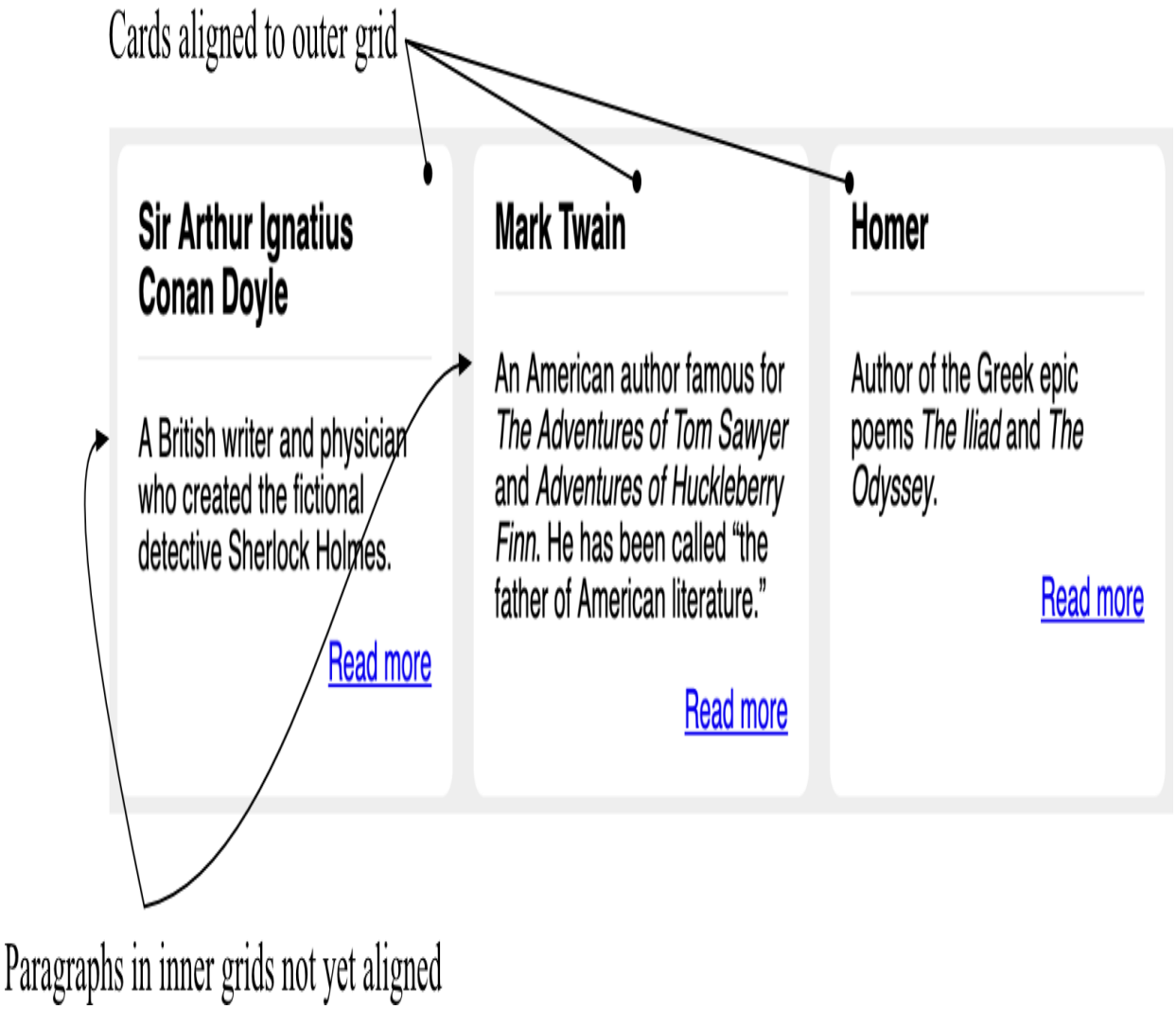
Listing 5.13 Laying out the author cards in a grid

```
body {
  background-color: #eee;
  font-family: Helvetica, Arial, sans-serif;
}

.author-bios {
  display: grid;                              #A
  grid-template-columns: repeat(3, 1fr);   #A
  gap: 1em;
  max-inline-size: 800px;
  margin-inline: auto;
}

.card {
  padding: 1rem;
  border-radius: 0.5rem;
  background-color: #fff;
}

.card > h3 {
  margin-block: 0;
```

```
  padding-block-end: 0.5rem;
  border-block-end: 1px solid #eee;
}

.read-more {
  margin-block-start: 1em;
  text-align: right;
}
```

With these styles in place, the cards are laid out in a row and they look pretty close to the design you want (figure 5.21). The last thing left to do is apply the subgrid.

Figure 5.21 Three author tiles styled before adding subgrid



To add a subgrid to the page, you must first apply `display: grid` to the grid items, to create an inner grid inside the first one. Then, you can use the keyword `subgrid` on the inner grid's `grid-template-rows` and/or `grid-template-columns`, to indicate it should use its parent grid's gridlines.

The CSS for this is shown in listing 5.14. Update your stylesheet to include these changes.

Listing 5.14 Adding a subgrid for the card contents

```
.card {
  display: grid;              #A
  gap: 0;
  grid-row: span 3;      #B
  grid-template-rows: max-content auto max-content;   #C
  grid-template-rows: subgrid;          #D
  padding: 1rem;
  border-radius: 0.5rem;
  background-color: #fff;
}

.card > h3 {
  align-self: end;       #E
  margin-block: 0;
  padding-block-end: 0.5rem;
  border-block-end: 1px solid #eee;
}
```

Because subgrid still has spotty browser support, I've supplied a fallback definition here for `grid-template-rows` that will approximate the subgrid behavior (it will push the Read more links to the bottom, but it will not align the bottoms of the headings). Alternatively, you could place these styles inside a feature query that checks for subgrid support: `@supports (grid-template-rows: subgrid) { … }`.

This example uses subgrid to align to the parent grid's columns, but you can also use it to align to rows using `grid-template-rows: subgrid`. Or you can do both. You can also continue the pattern down the DOM tree, nesting a subgrid inside another subgrid.

5.5.1 Additional options

In a subgrid, line numbers, line names, and grid area names are inherited from the parent grid, so you can use them to place items in the subgrid, for example, this will place all the card titles on the second row:

```
.card > h3 {
  grid-row: 2;
}
```

You can even supply new grid line names to the subgrid that are not present in the parent grid. Do this by providing a series of line names in brackets after the `subgrid` keyword:

```
.card {
  grid-template-rows: subgrid [line1] [line2] [line3] [line4];
}
```

You can refer to these the same as grid lines defined on a normal grid, as shown earlier in section 5.3.1. For example, `grid-row: line3 / line4` will place an item in the subgrid between the third and fourth horizontal grid lines.

Note

One grid feature that may be coming in the future is a *masonry* layout, which is a layout that is popular for photo albums, but requires the help of JavaScript to achieve. A masonry layout places grid items in equal width columns, but allows each item to be its natural height, so grid rows don’t necessarily align. See https://www.matuzo.at/blog/2023/100daysof-day72/ for an overview from developer Manuel Matuzović.

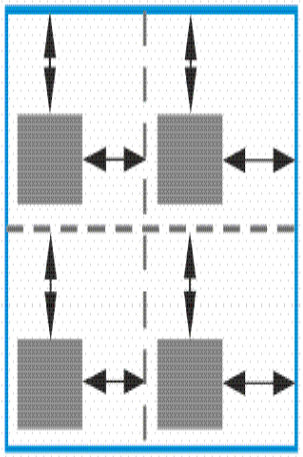
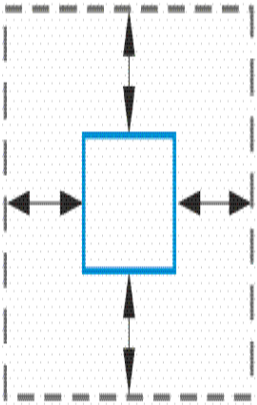
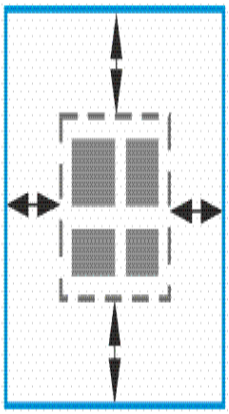
5.6 Alignment properties

The grid module makes use of several of the same alignment properties that flexbox uses, as well as a few of new ones. I’ve covered most of these in the previous chapter, but let’s look to see how they apply to a grid. If you need more control over various aspects of a grid layout, these properties may come in handy.

CSS provides three justify properties: `justify-content`, `justify-items`, `justify-self`. These properties control horizontal placement. I remember this by thinking about justifying text in a word processor, which spreads out text horizontally.

And there are three alignment properties: `align-content`, `align-items`, `align -self`. These control vertical placement. I remember this by thinking about the `vertical-align` property from inline alignment. These properties are each illustrated in figure 5.22.

Figure 5.22 Alignment properties for a grid

Properties	Applies to	Aligns
`justify-items` `align-items`	Grid container	Items within grid areas 
`justify-self` `align-self`	Grid item	Item within grid area 
`justify-content` `align-content`	Grid container	Grid tracks within container 

You can use `justify-content` and `align-content` to place the grid tracks horizontally and vertically within the grid container. This becomes necessary

if the total size of the grid doesn't fill the size of the grid container. Consider this code:

```
.grid {
  display: grid;
  height: 1200px;
  grid-template-rows: repeat(4, 200px);
}
```

It explicitly sets the height of a grid container to 1,200 px, but only defines 800 px worth of horizontal grid tracks. The `align-content` property specifies how to distribute the remaining 400 px of space. Supported values are as follows:

- `start`—Places grid tracks to the top/left of the grid container
- `end`—Places grid tracks to the bottom/right of the grid container
- `center`—Places grid tracks in the middle of the grid container
- `stretch`—Resizes the tracks to fill the size of the grid container
- `space-between`—Evenly distributes remaining space between each grid track (effectively overriding any `gap`)
- `space-around`—Distributes space between each grid track, with a half-sized space on each end
- `space-evenly`—Distributes space between each grid track, with an equal amount of space on each end

For detailed examples of all these justify/alignment properties, visit http://gridbyexample.com/. This is an excellent resource. It's a large collection of grid examples put together by Rachel Andrew, a developer and member of the W3C.

Because there's a lot to grid layout, I've taken you through the essential concepts you'll need to know. I encourage you to experiment with grids further. There are far more ways to mix and match these than I could cover in one chapter, so challenge yourself to try new things. When you come across an interesting page layout on the web, see if you can replicate it using grids.

5.7 Summary

- Grid defines a row- and column-based layout for your page, so that items can be placed in relation to one another.
- Grid works in conjunction with flexbox to provide a complete layout system
- Placement of grid items can be done using one of three syntaxes: numbered grid lines, named grid lines, and named grid areas. In any given situation, you can use whichever is the most intuitive for you and the needs of the layout.
- Auto-fill/auto-fit and implicit grid placement works well when laying out a large or unknown number of grid items.
- Subgrid can align more deeply nested elements than a simple set of parent-child elements.
- The same alignment properties from flexbox apply to a grid.

6 Positioning and stacking contexts

This chapter covers

- The types of element positioning: fixed, relative, absolute, and sticky
- Building modal dialogs and dropdown menus
- Understanding z-index and stacking contexts
- Drawing a simple triangle with CSS

We've now looked at multiple ways to control the layout of the page, from normal document flow to flexbox and grid. In this chapter, we'll look at one important technique: the `position` property, which you can use to build dropdown menus, modal dialogs, and other essential effects for modern web applications.

Positioning can get complicated. It's a subject where many developers only have a cursory understanding. Without a complete grasp on positioning, and the ramifications involved, it's easy to dig yourself into a hole. You can find yourself with the wrong elements in front of others and correcting the problem isn't always straightforward.

As we look at the various types of positioning, I'll make sure you understand precisely what they do. Then, you'll learn about something called a stacking context, which is a sort of hidden side effect of positioning. Understanding the stacking context will keep you out of trouble, and if you ever find yourself "out in the weeds" with a page layout, this understanding will give you the tools you need to get back on track.

The initial value of the `position` property is `static`. Everything we've done in previous chapters is with static positioning. When you change this value to anything else, the element is said to be *positioned*. An element with static positioning is thus *not positioned*.

The layout methods we've covered in previous chapters do various things to manipulate the way document flow behaves. Positioning is different: it

removes elements from the document flow entirely. This allows you to place the element somewhere else on the screen. It can place elements in front of or behind one another, thus overlapping one another.

6.1 Fixed positioning

Fixed positioning, although not as common as some of the other types of positioning, is probably the simplest to understand, so we'll start there. Applying `position: fixed` to an element lets you position the element arbitrarily within the viewport. This is done with four companion properties: `top`, `right`, `bottom`, and `left`. The values you assign to these properties specify how far the fixed element should be from each edge of the browser viewport. For example, `top: 3em` on a fixed element means its top edge will be 3 em from the top of the viewport.

By setting these four values, you also implicitly define the width and height of the element. For example, specifying `left: 2em; right: 2em` means the left edge of the element will be 2 em from the left side of the viewport, and the right edge will be 2 em from the right side; thus, the element will be 4 em less than the total viewport width. Likewise with `top`, `bottom`, and the viewport height.

You can also use the shorthand counterpart, `inset,` to specify the location of the elements. This property can conveniently specify all four sides at once. For example, `inset: 0` is equivalent to `top: 0; right: 0; bottom: 0; left: 0`. The `inset` property is shorthand for the following related logical properties:

- inset-block-start
- inset-block-end
- `inset-block` — shorthand for `inset-block-start` and `inset-block-end`
- inset-inline-start
- inset-inline-end
- `inset-inline` — shorthand for `inset-inline-start` and `inset-inline-end`

These shorthand properties tend to be useful when you would otherwise need a verbose set of declarations for the `top`, `right`, `bottom`, and `left` properties. The `inset` property itself also accepts up to four values to set all four values individually.

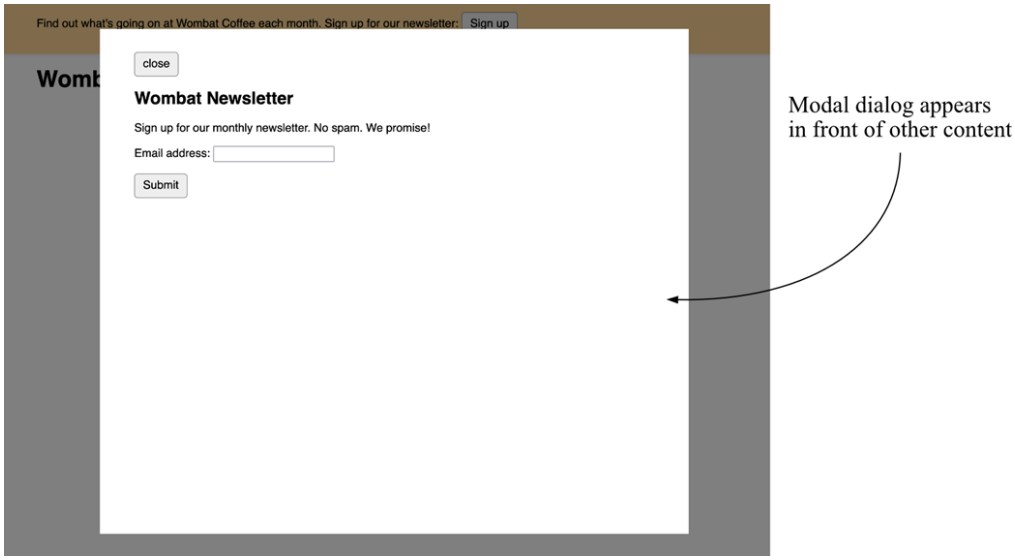
6.1.1 Creating a modal dialog with fixed positioning

You'll use these properties to build the *modal* dialog box shown in figure 6.1. This dialog will pop up in front of the page content, masking it from view until the dialog is closed.

Definition

A modal dialog box is a window that appears in front of the main page. While it is present, the UI behind it is disabled and the user must interact with the modal in some fashion in order to return to the main page.

Figure 6.1 A modal dialog box



Typically, you'll use a modal dialog to require the user to read something or to input something before continuing. For example, this modal (figure 6.1) displays a form where a user can sign up for a newsletter. You'll initially hide the dialog with `display: none`, and you'll then use a bit of JavaScript to change its value to `block` to open the modal.

Create a new page and add the following listing inside the `<body>` element. It places the content inside two containing elements and uses a `<script>` tag with some JavaScript to provide basic functionality.

Listing 6.1 Creating a modal dialog box

```
<header class="top-banner">
  <div class="top-banner-inner">
    <p>Find out what's going on at Wombat Coffee each
      month. Sign up for our newsletter:
      <button id="open" type="button">Sign up</button>    #A
    </p>
  </div>
</header>
<div class="modal" id="modal" role="dialog" aria-modal="true">
#B
  <div class="modal-backdrop"></div>       #C
  <div class="modal-body">                 #D
    <button class="modal-close" id="close"
type="button">close</button>
    <h2>Wombat Newsletter</h2>
    <p>Sign up for our monthly newsletter. No spam.
       We promise!</p>
    <form>
      <p>
        <label for="email">Email address:</label>
        <input type="text" name="email"/>
      </p>
      <p><button type="submit">Submit</button></p>
    </form>
  </div>
</div>
<main class="container">
  <h1>Wombat Coffee Roasters</h1>
</main>

<script type="text/javascript">
  var button = document.getElementById('open');
  var close = document.getElementById('close');
  var modal = document.getElementById('modal');

   button.addEventListener('click', function (event) {    #E
    modal.classList.add('is-open');                       #E
  });                                                     #E

  close.addEventListener('click', function (event) {   #F
```

```
    modal.classList.remove('is-open');                #F
  });                                                 #F
</script>
```

The first element in this listing is a banner across the top of the page. It contains the button that opens the modal. The second element is the modal dialog. It includes an empty `modal-backdrop`, which you'll use to obscure the rest of the page, drawing the user's focus to the contents of the dialog. The contents are inside a `modal-body` element.

The JavaScript here adds and removes an `is-open` class from the modal, which you use to toggle its visibility in the CSS.

Note

HTML now has a `<dialog>` element that provides a lot of modal dialog behavior automatically. I am not using it in this example in order to more fully illustrate how fixed positioning behaves. For more on the dialog element, see my article at https://keithjgrant.com/posts/2018/01/meet-the-new-dialog-element/.

The CSS to begin styling the page this is shown next. I want to point out a few things in these styles before moving on to positioning the modal. The body has a min-height of 200vh here. This is here to force scrolling on the page; it will be helpful in order to see how the positioning examples you will be adding throughout the chapter interact with scrolling. Alternatively, you could add several paragraphs of content inside the `<main>` element until the page scrolls naturally.

Add listing 6.2 to your stylesheet. It includes basic styling for the top banner and buttons.

Listing 6.2 Initial page styles

```
body {
  font-family: Helvetica, Arial, sans-serif;
  min-height: 200vh;              #A
  margin: 0;
}
```

```
button,
input {
  font: inherit;    #B
}

button {
  padding: 0.5em 0.7em;
  border: 1px solid #8d8d8d;
  background-color: #eee;
  border-radius: 5px;
  cursor: pointer;
}

.top-banner {
  padding: 0.5em;
  background-color: #ffd698;
  box-shadow: 0 1px 5px rgb(0 0 0 / 0.1);
}

.top-banner-inner {
  width: 80%;
  max-inline-size: 1000px;
  margin-inline: auto;
}

.container {
  width: 80%;
  max-inline-size: 1000px;
  margin: 1em auto;
}

.modal {
  display: none;       #C
}

.modal.is-open {
  display: block;      #D
}
```

You have also added `font: inherit` to buttons and input fields. This overrides user agent styles that apply some default font settings to form elements. This is a common step that I recommend adding to your stylesheets.

Finally, the modal is removed from the page with `display: none` until the `is-open` class is added, at which point it will be added back to the page by

the `display: block` declaration in the following ruleset.

To style the modal itself, you'll use fixed positioning twice. First on the `modal-backdrop` with the inset set to 0. This makes the backdrop fill the entire viewport. You'll give it a background color of `rgb(0 0 0 / 0.5)`. This color notation specifies red, green, and blue values of 0, which evaluates to black. The fourth value is the "alpha" channel, which specifies its transparency: a value of 0 is completely transparent, and 1 is completely opaque. The value `0.5` is half transparent. This serves to darken all of the page contents behind the element.

The second place you'll use fixed positioning is on the `modal-body`. You'll position each of its sides inside the viewport: 3 em from the top and bottom edges and 20% from the left and right sides. You'll also set a white background color. This makes the modal a white box, centered on the screen. You can scroll the page however you wish, but the backdrop and the modal body remain in place.

Add the styles shown in listing 6.3 to your stylesheet.

Listing 6.3 Adding modal styles

```
.modal-backdrop {
  position: fixed;                        #A
  inset: 0;                               #A
  background-color: rgba(0, 0, 0, 0.5); #A
}

.modal-body {
  position: fixed;      #B
  inset-block: 3em;     #B
  inset-inline: 20%;    #B
  padding: 2em 3em;
  background-color: white;
  overflow: auto;
}
```

Load the page and you'll see a pale yellow banner across the top of the screen with a button. Click the button to open the positioned modal. Because of fixed positioning, the modal remains in place, even when you scroll the page.

Click the close button at the top of the modal to close it. This button is in an odd location now, but we'll come back and position it in a bit.

6.1.2 Preventing the screen from scrolling while the modal dialog is open

Scrolling the page while the modal dialog is open is helpful for observing how fixed positioning works. However, it is not a great user experience. When a modal is in front of the main content of the page like this, the user usually expects to be able to interact with only the modal, not the other content.

You can fix this by applying `overscroll: hidden` to the body while the modal is open. Add these styles to your stylesheet:

```
body.no-scroll {
  overflow: hidden;
}
```

Then edit the JavaScript as shown in listing 6.4 to apply these styles to the body by adding and removing the `no-scroll` class on the body.

Listing 6.4 Update JavaScript to disable page scrolling

```
<script type="text/javascript">
  const button = document.getElementById("open");
  const close = document.getElementById("close");
  const modal = document.getElementById("modal");

  button.addEventListener("click", function (event) {
    modal.classList.add("is-open");
    document.body.classList.add("no-scroll");
  });

  close.addEventListener("click", function (event) {
    modal.classList.remove("is-open");
    document.body.classList.remove("no-scroll");
  });
</script>
```

Now when you open the modal, the page will no longer scroll. Close the modal, and scrolling will become re-enabled.

For interactive elements on the page, this pattern of adding and removing classes to and from elements is very powerful. It often allows you to keep your JavaScript relatively simple, leaving the key logic for visual treatment in your stylesheet.

Tip

In some browsers, this is possible to accomplish without the use of JavaScript by using the `:has()` pseudo-class. Instead of using the selector `body.no-scroll`, you can use `body:has(.modal.is-open)`. This will target the body element only while the modal has the `is-open` class applied, to disable page scrolling. At the time of writing, however, Firefox does not yet support the `:has()` pseudo-class. See Appendix A for more on `:has()`.

6.1.3 Controlling the size of positioned elements

When positioning an element, you're not required to specify values for all four sides. You can specify only the sides you need and then use `width` and/or `height` to help determine its size. You can also allow the element to be sized naturally. Consider these declarations:

```
position: fixed;
top: 1em;
right: 1em;
width: 20%;
```

These would affix the element 1 em from the top and right edges of the viewport with a width 20% of the viewport width. By omitting both `bottom` and `height` properties, the element's height will be determined naturally by its contents. This can be used to affix a navigational menu to the screen, for example. It'll remain in place even as the user scrolls down through the content on the page.

Because a fixed element is removed from the document flow, it no longer affects the position of other elements on the page. They'll follow the normal

document flow as if it’s not there, which means they’ll often flow behind the fixed element, hidden from view. This is usually fine with a modal dialog because you want it to be front and center until the user dismisses it.

With something persistent, such as a side-navigation menu, you’ll need to take care that other content doesn’t flow behind it. This is usually easiest to do by adding a margin to the content. For example, place all your content in a container with a `right-margin: 20%`. This margin will flow behind your fixed element, and the content won’t overlap.

6.2 Absolute positioning

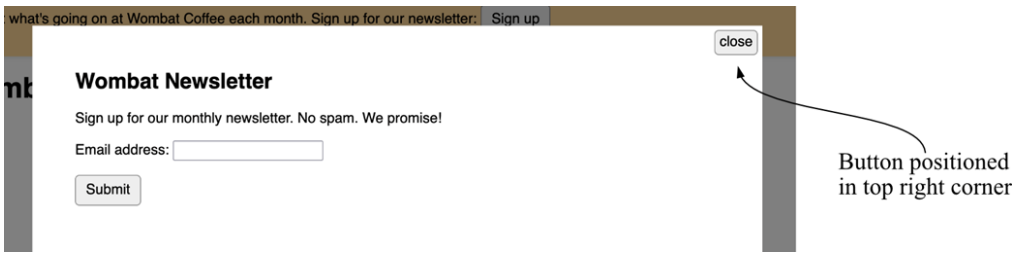
Fixed positioning, as you’ve just seen, lets you position an element relative to the viewport. This is called its *containing block*. The declaration `left: 2em`, for example, places the left edge of a positioned element 2 em from the left edge of the containing block.

Absolute positioning works the same way, except it has a different containing block. Instead of its position being based on the viewport, its position is based on the closest-positioned ancestor element. As with a fixed element, the inset properties (including `top`, `right`, `bottom`, and `left`) place the edges of the element within its containing block.

6.2.1 Absolutely positioning the Close button

To see how this works, you’ll reposition the Close button to the top right corner of your modal dialog. After doing so, the modal will look like figure 6.2.

Figure 6.2 Close button positioned at top right corner of modal dialog



To do this, you'll declare absolute positioning for the Close button. Its parent element is `modal-body`, which has fixed positioning, so it becomes the containing block for the button. Edit your button's styles to match listing 6.3.

Listing 6.5 Absolutely positioned Close button

```
.modal-close {
  position: absolute;   #A
  top: 0.3em;           #A
  right: 0.3em;         #A
  padding: 0.3em;
}
```

This listing places the button 0.3 em from the top and 0.3 em from the right of the `modal-body`. Typically, as in this example, the containing block will be the element's parent. In cases where the parent element is not positioned, then the browser will look up the DOM hierarchy at the grandparent, great-grandparent, and so on until a positioned element is found, which is then used as the containing block.

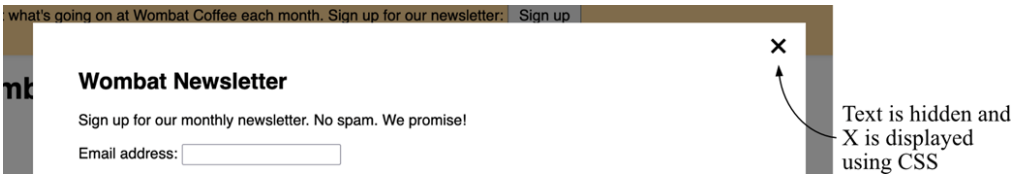
Note

If none of the element's ancestors are positioned, then the absolutely positioned element will be positioned based on something called the *initial containing block*. This is an area with dimensions equal to the viewport size, anchored at the top of the page.

6.2.2 Positioning a pseudo-element

You've positioned the Close button where you want it, but it's rather spartan. For a Close button such as this, users typically expect to see some graphical indication such as an *x*, as in figure 6.3.

Figure 6.3 Close button replaced with an *x*



Your first temptation may be to remove the word *close* from your markup and replace it with an *x*, but this would introduce an accessibility problem: Assistive screen readers read the text of the button, so it should give some meaningful indication of the button's purpose. The HTML must make sense on its own before the CSS is applied.

Instead, you can use CSS to hide the word *close* and display an *x*. You'll accomplish this by doing two things. First, you'll push the button's text outside the button and hide the overflow. Second, you'll use the `content` property to add the *x* to the button's `::after` pseudo-element and absolute positioning to center it within the button. Update your button's styles to match listing 6.4.

Tip

Instead of the letter *x*, I recommend the unicode character for the multiplication sign. It's more symmetric and usually more esthetically pleasing for this purpose. The HTML character `&times;` will render as this character, but in the CSS `content` property, you must use the escaped unicode number: `\00D7`.

Listing 6.6 Replacing the Close button with an *x*

```
.modal-close {
  position: absolute;
  top: 0.3em;
  right: 0.3em;
  padding: 0.3em;
  border: 0;
  font-size: 2em;
  height: 1em;           #A
  width: 1em;            #A
  text-indent: 10em;       #B
  overflow: hidden;        #B
  background-color: transparent;
```

```
}

.modal-close::after {
  position: absolute;
  line-height: 0.5;
  top: 0.2em;
  left: 0.1em;
  text-indent: 0;
  content: "\00D7";   #C
}
```

This listing explicitly sets the button size to 1 em square. The `text-indent` property then pushes the text to the right, outside the element. The exact value doesn't matter as long as it's more than the width of the button. Then, because `text-indent` is an inherited property, you reset it to 0 on the pseudo-class so the *x* isn't also indented.

The pseudo-class is now absolutely positioned. It behaves like a child element of the button, so the button being positioned becomes the containing block for its pseudo-element. The short `line-height` keeps the pseudo-element from being too tall, and the top and left properties position it in the center of the button. I arrived at the exact values here through some trial and error; I encourage you to experiment with those values in your browser's DevTools to see how they affect the positioning.

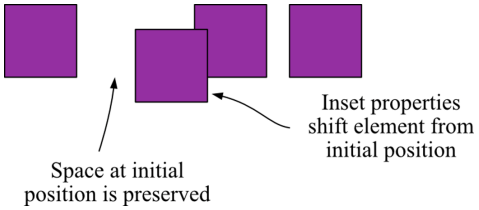
Absolute positioning is the heavy hitter among the positioning types. It's used often in conjunction with JavaScript for popping up menus, tooltips, and "info" boxes. I'll show you how to use it to build a dropdown menu, but to do that, we'll need to first take a look at its companion, relative positioning.

6.3 Relative positioning

Relative positioning is probably the least understood positioning type. When you first apply `position: relative` to an element, you won't usually see any visible change on the page. The relatively positioned element, and all elements on the page around it, will remain exactly where they were (though you may see some changes regarding which elements are in front of which. We'll get to that in a bit).

The inset properties, if applied, will shift the element from its original position, but it won't change the position of any elements around it. See figure 6.4 for an example. It shows four inline-block elements. I've applied three additional properties to the second element: `position: relative; top: 1em; left: 2em`. As you can see, this has shifted the element from its initial position, but the other elements are unaffected. They still follow normal document flow around the initial position of the shifted element.

Figure 6.4 The second element shifted using relative positioning



Applying `top: 1em` moves the element down 1 em (away from its original top edge). And `left: 2em` shifts the element right 2 em (away from its original left edge). These shifts may cause that element to overlap elements below or beside it. When positioning, negative values are supported as well, so `bottom: -1em` would shift the element down 1 em, just as `top: 1em` does.

Note

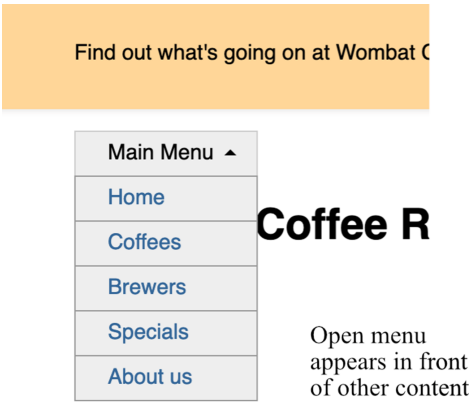
Unlike fixed and absolute positioning, you cannot use the inset properties to change the size of a relatively positioned element. Those values will only shift the position of the element up or down, left or right. You can use `top` or `bottom`, but not both together (`bottom` will be ignored); likewise, you can use `left` or `right`, but not both (`right` will be ignored).

Using these properties to adjust the position of a relative element may be useful to nudge an element into place on occasion, but truth be told, this is rarely why you'll use relative positioning. Far more often, you'll use `position: relative` to establish the containing block for an absolutely positioned element inside it.

6.3.1 Creating a dropdown menu

I'll show you how to use relative and absolute positioning to create a dropdown menu. Initially it'll be a simple rectangular button, but when the user clicks it, it pops open a list of links like those shown in figure 6.5.

Figure 6.5 Dropdown menu



Listing 6.5 shows the markup for this menu. Add it to your HTML at the beginning of the `<main class="container">` element. This code includes a container that you'll use to center the content and align it with the banner's content. The `<h1>` below the popup will illustrate how the popup appears in front of other content on the page.

Listing 6.7 Adding the dropdown menu's HTML

```
<main class="container">
  <nav>
    <div class="dropdown" id="dropdown">              #A
      <button type="button" class="dropdown-toggle"   #B
        id="dropdown-toggle">Main Menu</button>       #B
      <div class="dropdown-menu">                       #C
        <ul class="submenu">                            #C
          <li><a href="/">Home</a></li>                 #C
          <li><a href="/coffees">Coffees</a></li>       #C
          <li><a href="/brewers">Brewers</a></li>       #C
          <li><a href="/specials">Specials</a></li>     #C
          <li><a href="/about">About us</a></li>        #C
        </ul>                                           #C
      </div>                                            #C
    </div>
  </nav>
```

```
  <h1>Wombat Coffee Roasters</h1>
</main>
```

The dropdown container includes two children: the toggle button which is always visible, and the dropdown menu that will show and hide as the dropdown is opened and closed. The dropdown menu will be absolutely positioned so it doesn't shift around the layout of the page when it's revealed. This means it appears in front of other content when it's visible.

You'll need a bit of JavaScript to open and close the menu when the toggle button is pressed. Add the script shown in listing 6.8 to the page to add this functionality.

Listing 6.8 JavaScript to open and close the dropdown menu

```
<script type="text/javascript">
  const dropdownToggle = document.getElementById("dropdown-
toggle");
  const dropdown = document.getElementById("dropdown");

  dropdownToggle.addEventListener("click", function (event) {
    dropdown.classList.toggle("is-open");
  });
</script>
```

Similar to the modal you built earlier in the chapter, this JavaScript adds and removes an `is-open` class from the dropdown. Based on that, the CSS will be able to define the opened and closed styles.

Next, you'll apply relative positioning to the dropdown container. This establishes the containing block for the absolutely positioned menu within. Add these styles to your stylesheet, as shown in listing 6.9.

Listing 6.9 Displaying the open dropdown menu

```
.dropdown {
  display: inline-block;
  position: relative;        #A
}

.dropdown-toggle {
  padding: 0.5em 1.5em;
```

```
  border: 1px solid #ccc;
  background-color: #eee;
  border-radius: 0;
}

.dropdown-menu {
  display: none;          #B
  position: absolute;
  left: 0;
  top: 2.1em;                  #C
  inline-size: max-content;
  min-inline-size: 100%;
  background-color: #eee;
}

.dropdown.is-open .dropdown-menu {  #D
  display: block;                   #D
}                                   #D

.submenu {
  padding-inline-start: 0;
  margin: 0;
  list-style-type: none;
  border: 1px solid #999;
}

.submenu > li + li {
  border-top: 1px solid #999;
}

.submenu > li > a {
  display: block;
  padding: 0.5em 1.5em;
  background-color: #eee;
  color: #369;
  text-decoration: none;
}

.submenu > li > a:hover {
  background-color: #fff;
}
```

Now, when you click the Main Menu toggle button, the dropdown menu pops open beneath it. Clicking the toggle again will close it. Note that similar functionality could be accomplished using a `:hover` pseudo-class

instead, but that approach would not be compatible with keyboard navigation nor touchscreen devices.

On the absolutely positioned `dropdown-menu`, you used `left: 0` to align its left side with the left side of the dropdown. Then you used `top: 2.1em` to place its top edge beneath the label (with the padding and border, the label is about 2.1 em tall). A `min-width` of 100% ensures it will be at least as wide as the dropdown container (whose width is determined by `dropdown-label`). You then used the `submenu` class to style the menu inside the dropdown. (If you open the modal dialog at this point, you might notice it appears behind the dropdown menu in an odd way. That's okay; we'll address that issue soon.)

Tip

When a dropdown menu is on the right edge of the screen, it often makes sense to position it using `right: 0` rather than `left: 0`. That way, wide menu items will extend to the left of the toggle button instead of overflowing outside the right edge of the viewport.

Accessibility is an important concern when building web pages, and this is doubly so for interactive elements such as dropdown menus. At a bare minimum, always ensure you can fully interact with your page via the keyboard. For this dropdown menu, you can use the Tab key to focus on the menu's toggle button and space bar to open and close the menu. When the menu is open, you can use Tab to cycle through the menu items. (On Mac OS, you may need to enable keyboard navigation in your System Preferences.) For more on building accessible dropdowns, see https://www.webaxe.org/accessible-custom-select-dropdowns/.

6.3.2 Creating a CSS triangle

There is one finishing touch you can add to your dropdown menu. It works as is, but it may not be immediately apparent to the user that there's more content to discover along with the Main Menu label. You'll add a small down arrow to the label to indicate there's more to explore.

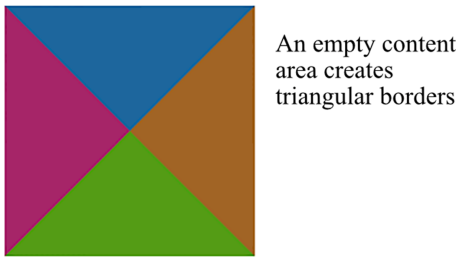
You can use a little trick with borders to draw a triangle that serves as the down-pointing arrow. You'll use the dropdown label's `::after` pseudo-element to draw the triangle, then use absolute positioning to put it on the right side of the label.

Most of the time, when you add borders to an element, you make them thin: usually 1 px or 2 px is enough. But observe what happens when you make a border much thicker as shown in figure 6.6. I've made each side of the border a unique color to highlight where one edge ends and the next begins.

Figure 6.6 An element with thick borders

Notice what happens in the corners where the borders from two edges meet: they form a diagonal edge. Now observe what happens if you shrink the element to a height and width of zero (figure 6.7). The borders all come together and join in the middle.

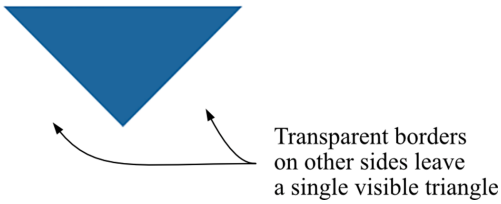
Figure 6.7 When the element has no height or width, each border forms a triangle.



The border for each side of the element forms a triangle. The top border points down; the right border points left, and so on. With this in mind, you can use one border to give you the triangle you need and then set the color of the remaining borders to transparent. An element with transparent left

and right borders and a visible top border will look like figure 6.8—a simple triangle.

Figure 6.8 Triangle formed from an element's top border



Apply styles to the `dropdown-label::after` pseudo-element to make a triangle and position it using absolute positioning. Add this to your stylesheet.

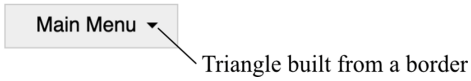
Listing 6.10 Absolutely positioning a triangle on the dropdown label

```
.dropdown-toggle {
  padding: 0.5em 2em 0.5em 1.5em;   #A
  border: 1px solid #ccc;
  background-color: #eee;
  border-radius: 0;
}

.dropdown-toggle::after {
  content: "";
  position: absolute;        #B
  right: 1em;                #B
  top: 0.9em;                #B
  border: 0.3em solid;                          #C
  border-color: black transparent transparent;    #C
}

.dropdown.is-open .dropdown-toggle::after {
  top: 0.6em;
  border-color: transparent transparent black;   #D
}
```

The pseudo-element has no content, so it has no height or width. You then used the `border-color` shorthand property to set the top border black and the side and right borders transparent, giving you the down arrow. An extra

bit of padding on the `dropdown-label` makes space on the right, where you can place the triangle. The final result is shown in figure 6.9.

Figure 6.9 Dropdown label with a down arrow



When you open the menu, the arrow reverses directions, pointing upward to indicate the menu can be closed. The slight change in the `top` value (from 0.9 em to 0.6 em) helps the up arrow to optically appear in the same spot as the down arrow.

Alternatively, you can add this arrow with the use of an image or background image, but a few extra lines of CSS can save your users the extraneous request over the network. You can also inline a small SVG in the HTML, but I find if I'm not already using an icon pack of some sort in a project, this CSS-only approach can be convenient. The result of this is a small finishing touch that can add a lot of polish to your application or website.

You can use this technique to build other complicated shapes, such as trapezoids, hexagons, and stars. For a list of dozens of shapes built in CSS, see https://css-tricks .com/examples/ShapesOfCSS/. These are often not needed, especially with the rise of SVG icons, but they illustrate the versatility of CSS.

6.4 Stacking contexts and z-index

Positioning is useful, but it's important to know the ramifications involved. When you remove an element from the document flow, you become responsible for all the things the document flow normally does for you.

You need to ensure the element doesn't accidentally overflow outside the browser viewport, thus becoming hidden from the user. And, you need to make sure it doesn't inadvertently cover important content you need visible.

Eventually, you'll encounter problems with stacking. When you position multiple elements on the same page, you may run into a scenario where two different positioned elements overlap. You may occasionally be surprised to find the "wrong" one appearing in front of the other. In fact, in this chapter, I've intentionally set up such a scenario to illustrate this.

On the page you've built, click the Sign up button in the page header to open the modal dialog. If you placed the markup for the dropdown after the modal in your HTML, it'll look like figure 6.10. Notice the dropdown you added to the page now appears in front of the modal.

Figure 6.10 The modal dialog incorrectly appears behind the dropdown menu

You can address this problem in a couple of ways. Before we get to that, it's important to understand how the browser determines the stacking order. For that, we need to take a closer look at how the browser renders a page.

6.4.1 Understanding the rendering process and stacking order

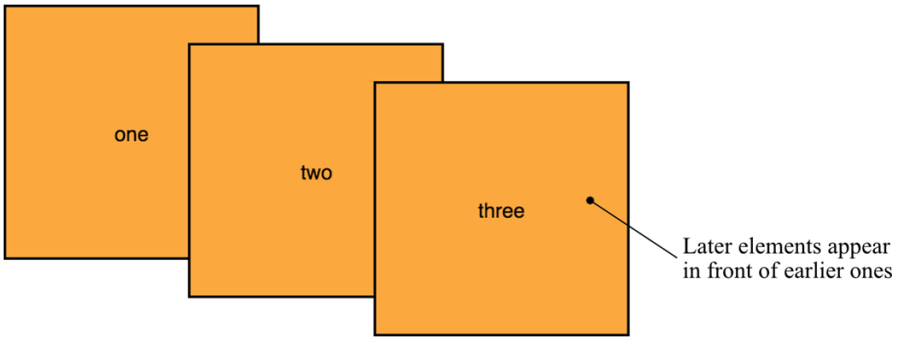
As the browser parses HTML into the DOM, it also creates another tree structure called the *render tree*. This represents the visual appearance and position of each element. It's also responsible for determining the order in which the browser will *paint* the elements. This order is important because elements painted later appear in front of elements painted earlier, should they happen to overlap.

Under normal circumstances (that is, before positioning is applied), this order is determined by the order the elements appear in the HTML. Consider the three elements in this bit of markup:

```
<div>one</div>
<div>two</div>
<div>three</div>
```

Their stacking behavior is illustrated in figure 6.11. I've used some negative margins to force them to overlap, but I haven't applied any positioning to them. The elements appearing later in the markup are painted over the top of the previous ones.

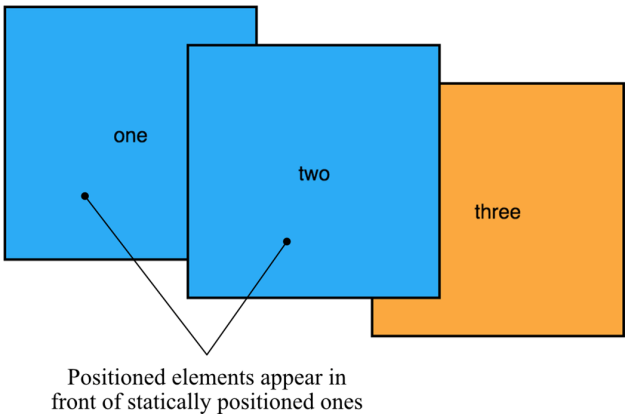
Figure 6.11 Three elements stacking normally



This behavior changes when you start positioning elements. The browser first paints all non-positioned elements, then it paints the positioned ones. By default, any positioned elements appear in front of any non-positioned elements. In figure 6.12, I've added `position: relative` to the first two elements. This brings them to the front, covering the statically positioned third element, even though the order of the elements in the HTML is unchanged.

Notice that among the positioned elements, the second element still appears in front of the first. Positioned elements are brought to the front, but the source-dependent stacking between them remains unchanged.

Figure 6.12 Positioned elements are painted in front of static elements.



On your page, this means that both the modal and the dropdown menu appear in front of the static content (which you want), but whichever appears later in your markup displays in front of the other. One way to fix your page would be to move the `<div class="modal">` and all of its contents to somewhere after the dropdown menu.

Typically, modals are added to the end of the page as the last bit of content before the closing `</body>` tag. Most JavaScript libraries for building modal dialogs will do this automatically. Because the modal uses fixed positioning, it doesn't matter where in the markup it appears; it'll always be positioned in the center of the screen.

Moving the element to somewhere else in the markup tends to be harmless for fixed positioning, but this isn't a solution you can normally use for relative or absolute positioning. Relative positioning depends on the document flow, and absolute positioning depends on its positioned ancestor element. We need a way to control their stacking behavior. This is done with the `z-index` property.

6.4.2 Manipulating stacking order with z-index

The `z-index` property can be set to any integer (positive or negative). *Z* refers to the depth dimension in a Cartesian X-Y-Z coordinate system. Elements with a higher z-index appear in front of elements with a lower z-index. Elements with a negative z-index appear behind static elements.

Using z-index is the second approach you can use to fix your page's stacking problem. This approach gives you more freedom in how you structure the HTML. Apply a z-index of `1` to the `modal-backdrop` and a z-index of `2` to the `modal-body` (this will ensure the body appears in front of the backdrop). Update this portion of your stylesheet to match the next listing.

Listing 6.11 Adding z-indexes to the modal dialog to bring it in front of the dropdown

```
.modal-backdrop {
  position: fixed;
  inset: 0;
  background-color: rgba(0 0 0 / 0.5);
  z-index: 1;           #A
}

.modal-body {
  position: fixed;
  inset-block: 3em;
  inset-inline: 20%;
  padding: 2em 3em;
  background-color: white;
  overflow: auto;
  z-index: 2;         #B
}
```

Z-index seems straightforward, but using it introduces two gotchas. First, z-index works only on positioned elements. You cannot manipulate the stacking order of static elements. Second, applying a z-index to a positioned element establishes something called a stacking context.

6.4.3 Understanding stacking contexts

A *stacking context* consists of an element or a group of elements that are painted together by the browser. One element is the root of the stacking context, so when you add a z-index to a positioned element that element becomes the root of a new stacking context. All of its descendant elements are then part of that stacking context.

The fact that all the elements of the stacking context are painted together has an important ramification: No element outside the stacking context can

be stacked between any two elements that are inside it. Put another way, if an element is stacked in front of a stacking context, no element within that stacking context can be brought in front of it. Likewise, if an element on the page is stacked behind the stacking context, no element within that stacking context can be stacked behind that element.

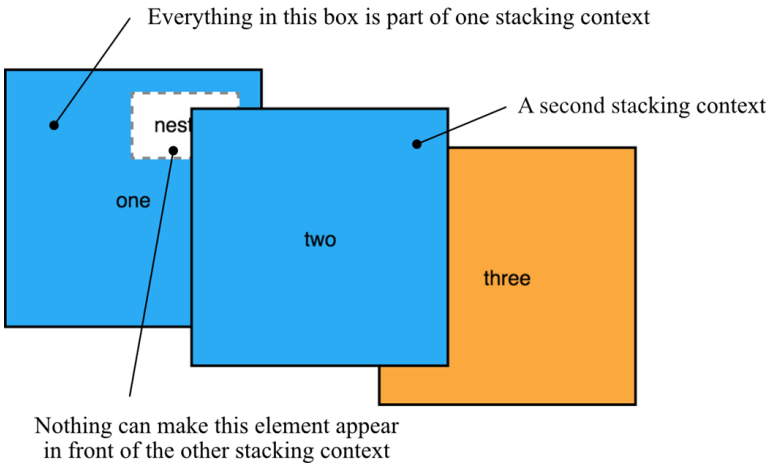
This can be a little tricky to get your head around, so I'll show you how to put together a minimal scenario to illustrate this. In a fresh HTML page, add this markup.

Listing 6.12 Stacking contexts example

```
<div class="box one positioned">
  one
  <div class="absolute">nested</div>
</div>
<div class="box two positioned">two</div>
<div class="box three">three</div>
```

This markup consists of three boxes, two of which will be positioned and given a z-index of 1. The `absolute` element inside the first box will be positioned and given a z-index of 100. Despite its high z-index, it still appears behind the second box because its parent forms a stacking context behind the second box (figure 6.13).

Figure 6.13 The entire stacking context is stacked together relative to other elements on the page.



The styles for this scenario are in listing 6.10. Apply this to your page. Most of these styles are for sizing and coloring to make the exact stacking order plainly visible. The negative margins force the elements to overlap. The only essential declarations are `position` and `z-index` for the various elements.

Listing 6.13 Creating the stacking contexts

```
body {
  margin: 40px;
}

.box {
  display: inline-block;
  width: 200px;
  line-height: 200px;
  text-align: center;
  border: 2px solid black;
  background-color: #ea5;
  margin-left: -60px;
  vertical-align: top;
}

.one { margin-left: 0; }
.two { margin-top: 30px; }
.three { margin-top: 60px; }

.positioned {                #A
  position: relative;        #A
  background-color: #5ae;    #A
  z-index: 1;                #A
}                            #A

.absolute {
  position: absolute;
  top: 1em;
  right: 1em;
  background-color: #fff;
  border: 2px dashed #888;
  z-index: 100;            #B
  line-height: initial;
  padding: 1em;
}
```

Box one, stacked behind box two, is the root of a stacking context. Because of this, the absolutely positioned element within it cannot be made to appear in front of box two, even with a high z-index value. Play around with this example in your browser's developer tools to get a feel for things. Manipulate the z-index of each element to see what happens.

Note

Adding a z-index is not the only way to create a stacking context. An `opacity` below 1 creates one, as do the `transform` or `filter` properties. Fixed positioning and sticky positioning (section 6.5) always create a stacking context, even without a z-index. The document root (`<html>`) also creates a top-level stacking context for the whole page.

All the elements within a stacking context are stacked in this order, from back to front:

- The root element of the stacking context
- Positioned elements with a negative z-index, along with their children
- Non-positioned elements
- Positioned elements with a z-index of `auto`, and their children
- Positioned elements with a positive z-index, and their children

Use variables to keep track of z-indexes

It's easy for a stylesheet to devolve into a z-index war, with no clear order as to the priority of various components. Without clear guidance, developers adding to the stylesheet might add something like a modal and, for fear of it appearing behind something else on the page, they'll give it a ridiculously high z-index, like 999999. After this happens a few times, it's anybody's guess what the z-index for another new component should be.

To keep this under control, use custom properties to your advantage. Put all your z-index values into variables, all in one place. That way you can see at a glance what is supposed to appear in front of what:

```
--z-loading-indicator: 100;
--z-nav-menu:          200;
```

```
--z-dropdown-menu:     300;
--z-modal-backdrop:    400;
--z-modal-body:        410;
```

Use increments of 10 or 100, so you can insert new values in between should the need arise.

If you ever find that z-index isn't behaving how you'd expect, look up the DOM tree at the element's ancestors until you find one that is the root of a stacking context. Then use z-index to bring the entire stacking context forward or backward. Be careful, as you may find multiple stacking contexts nested within one another.

When the page is complicated, it can be difficult to determine exactly which stacking context is responsible for the behavior you see. For this reason, always be cautious when creating a stacking context. Don't create one unless you have a specific reason for it. This is doubly true for elements that encompass large portions of the page. When possible, bring standalone positioned elements like modals to the top level of the DOM, right before the closing `</body>` tag, so you don't have to break them free from any outer stacking contexts.

Some developers fall into the trap of positioning a large number of elements all over the page. You should fight this urge. The more you use positioning, the more complicated the page becomes, and the harder it is to debug. If you find you're positioning dozens of elements, step back and reassess the situation. This is particularly important if you find yourself fighting to get the layout to behave the way you want. When your layout can be accomplished using other methods, you should use those methods instead.

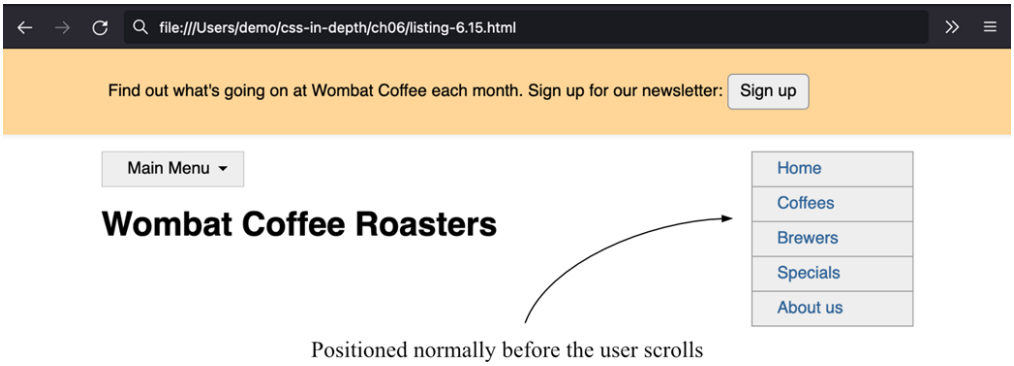
If you can get the document flow to do the work for you, rather than explicitly positioning things, the browser will take care of a lot of edge-cases for you. Remember, positioning takes elements out of the document flow. Generally speaking, you should do this only when you need to stack elements in front of one another.

6.5 Sticky positioning

The last type of positioning is *sticky positioning*. It's sort of a hybrid between relative and fixed positioning: The element scrolls normally with the page until it reaches a specified point on the screen, at which point it will "lock" in place as the user continues to scroll. A common use-case for this is sidebar navigation.

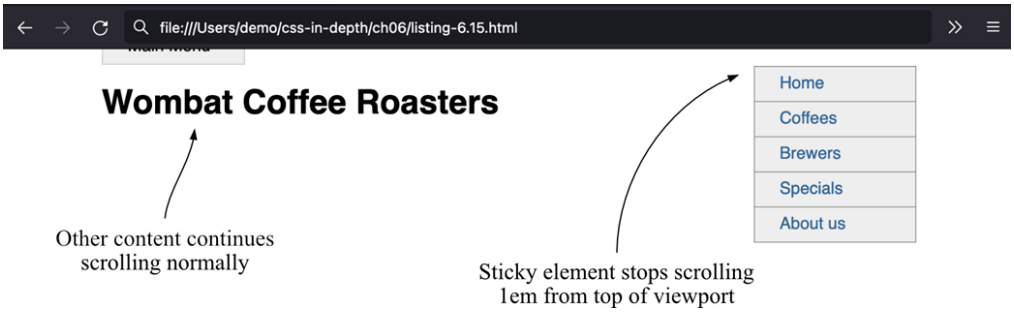
Return to your page that has the modal dialog and dropdown menu. You'll change it to a two-column layout, adding a sticky-positioned sidebar as the right-hand column. This will look like the screenshot in figure 6.14.

Figure 6.14 The sticky-positioned sidebar is initially positioned normally.



When the page initially loads, the position of the sidebar appears routine. It'll scroll normally with the page—up to a point. As soon as it's about to leave the viewport, it'll lock into place. It'll then stay there like a fixed-positioned element, remaining on the screen as the rest of the page continues to scroll. After scrolling down the page a little, the page will look like figure 6.15.

Figure 6.15 The sidebar remains "stuck" to the viewport.



To illustrate this, restructure your page a bit to define two columns. Edit the container in your HTML to match the next listing. This places your existing content (the dropdown menu and page heading) into a left column and adds a right column with an “affix” menu.

Listing 6.14 Changing to a two-column layout with a sidebar

```
<main class="container">
  <div class="col-main">          #A
    <nav>
      <div class="dropdown">
        <div class="dropdown-label">Main Menu</div>
        <div class="dropdown-menu">
          <ul class="submenu">
            <li><a href="/">Home</a></li>
            <li><a href="/coffees">Coffees</a></li>
            <li><a href="/brewers">Brewers</a></li>
            <li><a href="/specials">Specials</a></li>
            <li><a href="/about">About us</a></li>
          </ul>
        </div>
      </div>
    </nav>
    <h1>Wombat Coffee Roasters</h1>
  </div>

  <aside class="col-sidebar">       #B
    <div class="affix">             #B
      <ul class="submenu">
        <li><a href="/">Home</a></li>
        <li><a href="/coffees">Coffees</a></li>
        <li><a href="/brewers">Brewers</a></li>
        <li><a href="/specials">Specials</a></li>
        <li><a href="/about">About us</a></li>
      </ul>
    </div>
  </aside>
</main>
```

Next, you’ll update the CSS so the container becomes a flex container to size the two columns. For this demo, you’ll repurpose the submenu styles from the dropdown menu, though you could alternately add whatever elements and styles you want into the sidebar. Add these styles to your stylesheet.

Listing 6.15 Creating a two-column layout and sticky-position side menu

```
.container {
  display: flex;     #A
  width: 80%;
  max-width: 1000px;
  margin: 1em auto;
  min-height: 100vh;  #B
}

.col-main {
  flex: 1 80%;     #C
}

.col-sidebar {
  flex: 20%;       #C
}

.affix {
  position: sticky;    #D
  top: 1em;            #D
}
```

Most of these changes are setting up the two-column layout. With that established, it takes only two declarations to position the `affix` element. The `top` value sets the location where the element will lock into place: 1 em from the top of the viewport.

A sticky element will always remain within the bounds of its parent element —the `col-sidebar` in this case. As you scroll down the page, the `col-sidebar` will always scroll normally, but the `affix` element will scroll until it locks into place. Continue scrolling far enough and it'll unlock and resume scrolling. This occurs when the bottom edge of the parent reaches the bottom edge of the sticky menu. Note that the parent must be taller than the sticky element in order for it to stick into place, which is why I artificially increased its height by adding a `min-height` to its flex container. If the parent element were to scroll all the way out of the viewport, the sticky element would resume scrolling again when the bottom of its parent reaches it.

6.6 Summary

- Fixed positioning places elements relative to the viewport.
- Relative positioning places elements relative to their initial static position. Other elements on the page will continue to flow around its initial position.
- Absolute positioning places elements relative to their nearest positioned ancestor element.
- Sticky positioning allows elements to stay on screen as the rest of the page scrolls. They will stay contained within their parent element.
- `z-index` only works on positioned elements and using it creates a new stacking context.

Appendix A. Selectors reference

A.1 Basic selectors

- *`tagname`*—Type selector or tag selector. This selector matches the tag name of the elements to target. It has a specificity of 0,0,1. Examples: `p`; `h1`; `strong`.
- *`.class`*—Class selector. This targets elements that have the specified class name as part of their class attribute. It has a specificity of 0,1,0. Examples: `.media`; `.nav-menu`.
- *`#id`*—ID selector. This targets the element with the specified ID attribute. It has a specificity of 1,0,0. Example: `#sidebar`.
- `*`—Universal selector, which targets all elements. This has a specificity of 0,0,0.

A.2 Combinators

Combinators join multiple simple selectors into one complex selector. In the selector `.nav-menu li`, for example, the space between the two simple selectors is known as a *descendant combinator*. It indicates the targeted `<li>` is a descendant of an element that has the `nav-menu` class. This is the most common combinator, but there are a few others, each of which indicates a particular relationship between the elements indicated:

- *Child combinator* (`>`)—Targets elements that are a direct descendant of another element. Example: `.parent > .child`.
- *Adjacent sibling combinator* (`+`)—Targets elements that immediately follow another. Example: `p + h2` targets an `<h2>` that immediately follows a `<p>`.
- *General sibling combinator* (`~`)—Targets all sibling elements following a specified element. Note this doesn't target siblings that appear before the indicated element. Example: `li.active ~ li`.

Note that the two sibling combinators only target elements that follow a specified sibling; they cannot select elements that precede it. To select preceding siblings, use the `:has()` selector instead. For example `li:has(~ li.featured)` will target list items before `<li class="featured">`, or `ul:has(> li.featured) > li` will target all list items in a list that contains an `<li class="featured">`.

Compound selectors

Multiple simple selectors can be joined (without spaces or other combinators) to form a *compound* selector (for example, `h1.page-header`). A compound selector targets elements that match *all* its simple selectors. For example, `.dropdown.is-active` targets `<div class="dropdown is-active">` but not `<div class="dropdown">`.

A.3 Pseudo-class selectors

Pseudo-class selectors are used to target elements when they're in a certain state. This state can be due to user interaction or the element's position in relation to its parent or sibling elements in the document. Pseudo-class selectors always begin with a colon (`:`). These have the same specificity as a class selector (0,1,0), except `:is()` and `:where()`, which have special behavior regarding their specificity, as noted below.

There are quite a lot of pseudo-class selectors available. I have grouped these into three categories: general purpose, those that select based on position among siblings, and those that pertain to forms.

This list of pseudo-classes is not exhaustive. See the MDN documentation at https://developer.mozilla.org/en-US/docs/Web/CSS/Pseudo-classes for a complete list.

General purpose pseudo-classes

- `:active`—Targets an element that is currently being activated by the user by mouse or keyboard interaction. This usually pertains to links or buttons.

- `:any-link`—Targets elements that match either `:link` or `:visited`.
- `:empty`—Targets elements that have no children. Beware that this doesn't target an element that contains whitespace as the whitespace is represented in the DOM as a text node child.
- `:focus`—Targets elements that have received focus, whether from a mouse click, screen tap, or Tab key navigation.
- `:focus-visible`—Targets elements that have received focus when the browser has determined the focus should be made visually evident. Typically, this means an element that has been navigated to via keyboard but not a mouse click. This is useful because you may not want a "focus ring" (usually an outline) to appear when a user clicks a button or input. But for accessibility, the focus ring should still appear for users using keyboard navigation, so it is clear where they are on the page.
- `:focus-within`—Targets elements that either match `:focus` or contain descendant elements that do.
- `:has(<selector list>)`—Sometimes called a "parent selector", this targets an element if the selector list targets at least one element relative to that element. For example, `article:has(h2)` will target an `<article>` if it contains an `<h2>` anywhere among its descendant elements. The relative selector may begin with a combinator, so `h1:has(+ h2)` will target an `<h1>` that is immediately followed by an `<h2>`. At the time of writing, this selector is not yet supported in Firefox; see https://caniuse.com/css-has for the latest support information.
- `:hover`—Targets elements while the mouse cursor hovers over them.
- `:is(<selector list>)`—Targets any elements matched by the selector list. Specificity is equal to the highest specificity selector in the list. Example: `:is(h1, h2, h3)`. See Chapter 8 for more details.
- `:lang(<language>)`—Targets elements based on their language. Language can be set either via the `lang` HTML attribute or via a `<meta>` tag. Example: `article:lang(en-US)`.
- `:link`—Targets a link (`<a>`) that has not yet been visited by the user.
- `:modal`—Targets a `<dialog>` element that has been opened with JavaScript's `showModal()` method. Also see the `::backdrop` pseudo-element below.

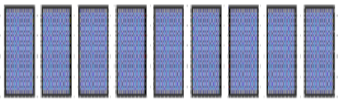
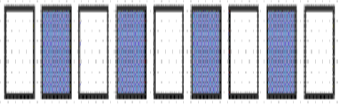
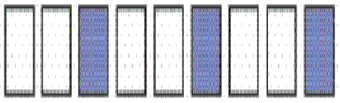
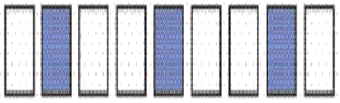
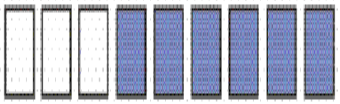
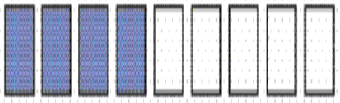
- `:not(<selector>)`—Targets elements that don't match the selector within the parentheses. The selector inside the parentheses must be simple: it can only refer to the element itself; you can't use this selector to exclude ancestors. It also mustn't contain another negation selector.
- `:root`—Targets the root element of the document. In HTML, this is the `<html>` element. But CSS can be applied to other XML or XML-like documents, such as SVG; in which case, this selector works more generically.
- `:visited`—Targets a link (`<a>`) that has already been visited by the user.
- `:where(<selector list>)`—Behaves the same as `:is()`, but selector specificity is always 0,0,0. Example: `:is(:hover, .is-selected)`. See Chapter 8 for more details.

Selecting based on position among sibling elements

Several pseudo-class selectors involve targeting elements based on where they reside among sibling elements. Some of these are straightforward, targeting the first or last element. Others use the formula *an+b* to specify a specific subset of elements to target.

In this formula, `a` and `b` are integers. To know exactly how a formula works, imagine solving it for all integer values of `n`, beginning with zero (the keywords `even` or `odd` can also be substituted for the equation). The results of this equation indicate which elements are targeted. This figure illustrates some examples:

Figure A.1 Examples of the an+b pseudo-class formula

Selector	Elements targeted	Result	Description
`:nth-child(n)`	0, 1, 2, 3, 4 ...		Every element
`:nth-child(2n)`	0, 2, 4, 6, 8 ...		Even elements
`:nth-child(3n)`	0, 3, 6, 9, 12 ...		Every third element
`:nth-child(3n+2)`	2, 5, 8, 11, 14 ...		Every third element beginning with element 2
`:nth-child(n+4)`	4, 5, 6, 7, 8 ...		All elements beginning with element 4
`:nth-child(-n+4)`	4, 3, 2, 1, 0 ...		First four elements

- `:first-child`—Targets elements that are the first element within their parent element.
- `:first-of-type`—Similar in nature to `:first-child`, except instead of considering the position among all children, it considers an element's numeric position only among other children with the same tag name.
- `:last-child`—Targets elements that are the last element within their parent element.
- `:last-of-type`—Targets the last child element of its type.
- `:nth-child(`*an*+*b*`)`—Targets elements based upon their position among their siblings. For example, `.card:nth-child(2n)` targets every even-numbered element that also has the class `card`.

- `:nth-child(`*`a`*`n+`*`b`*` of <`*`selector`*`>)`—By adding the `of` keyword and a selector, this will target the nth element among those that match the given selector. For example, `li:nth-child(3 of .featured)` will target the third `<li>` with the `featured` class.
- `:nth-last-child(`*`a`*`n+`*`b`*`)`—Similar to `:nth-child()`, but instead of counting forward from the first element, this selector counts backward from the last element. The formula in the parentheses follows the same pattern as in `:nth-child()`, including support for the optional `of <`*`selector`*`>` syntax.
- `:nth-last-of-type(`*`a`*`n+`*`b`*`)`—Targets elements of their type based on a specified formula, counting from the last of those elements backward; similar to `:nth-last-child`.
- `:nth-of-type(`*`a`*`n+`*`b`*`)`—Targets elements of their type based on their numerical order and the specified formula; similar to `:nth-child`.
- `:only-child`—Targets elements that are the only element within their parent element (no siblings).
- `:only-of-type`—Targets elements that are the only child of their type.

Form field pseudo-classes

The following pseudo-class selectors all relate to forms and form fields.

- `:checked`—Targets selected checkboxes, radio buttons, or select box options.
- `:disabled`—Targets disabled elements, including inputs, selects, and buttons.
- `:enabled`—Targets enabled elements, meaning they can be activated or accept focus.
- `:invalid`—Targets elements with invalid input values, as defined by the input type. For example, an `<input type="email">`, whose value is not a valid email address.
- `:optional`—Targets elements that do not have a `required` attribute set.
- `:placeholder-shown`—Targets an input or textarea that’s currently displaying placeholder text.
- `:required`—Targets elements with a `required` attribute set.
- `:user-invalid`—Targets form fields with invalid input, but only after the user has interacted with them. This has limited browser support at

the time of writing.
- `:user-valid`—Targets form fields with valid input, but only after the user has interacted with them. This has limited browser support at the time of writing.
- `:valid`—Targets form fields with valid values.

A.4 Pseudo-element selectors

Pseudo-elements are similar to pseudo-classes, but instead of selecting elements with a special state, they target a certain part of the document that doesn't directly correspond to a particular element in the DOM. They may target only portions of an element or even inject content into the page where the markup defines none.

These selectors begin with a double-colon (`::`), though most browsers also support a single-colon syntax for some pseudo-elements for backward-compatibility reasons. Pseudo-elements have the same specificity as a type selector (0,0,1).

- `::after`—Creates a pseudo-element that becomes the last child of the matched element. This element is inline by default. Can be used to insert text, images, or other shapes. The `content` property must be specified to make this element appear. Example: `.menu::after`.
- `::backdrop`—Targets a box the size of the viewport, which is rendered immediately behind the corresponding element. The corresponding element must either be a `<dialog>` or an element made full screen via the JavaScript `requestFullscreen()` method (such as a video). Example: `dialog::backdrop`.
- `::before`—Creates a pseudo-element that becomes the first child of the matched element. This element is inline by default. Can be used to insert text, images, or other shapes. The `content` property must be specified to make this element appear. Example: `.menu::before`.
- `::first-letter`—Lets you specify styles for only the first text character inside the matched element. Example: `h2::first-letter`.
- `::first-line`—Lets you specify styles for the first line of text inside the matched element.

- `::marker`—Selects the marker of a list item (or any element with `display: list-item`). This is typically a bullet or number, depending on a list type. This allows you to style its color or size, for example. You can also use the `content` property to replace the marker with new content, such as a different unicode character.
- `::placeholder`—Allows you to style the placeholder text shown in an input or textarea.
- `::selection`—Lets you specify styles for any text the user has highlighted with their cursor. This is often used to change the `background-color` of selected text. Only a handful of properties can be used; those include color, background color, cursor, and text decoration.

A.5 Attribute selectors

Attribute selectors can be used to target elements based on their HTML attributes. These have the same specificity as a class selector (0,1,0).

- `[attr]`—Targets elements that have the specified attribute `attr`, regardless of its value. Example: `input[disabled]`.
- `[attr="value"]`—Targets elements that have the specified attribute `attr`, and its value matches the specified string value. Example: `input[type="radio"]`.
- `[attr^="value"]`—"Starts with" attribute selector. Targets by attribute and value, where the value begins with the specified string value. Example: `a[href^= "https"]`.
- `[attr$="value"]`—"Ends with" attribute selector. Targets by attribute and value, where the value ends with the specified string value. Example: `a[href$= ".pdf"]`.
- `[attr*="value"]`—"Contains" attribute selector. Targets by attribute and value, where the attribute value contains the specified string value. Example: `[class*="sprite-"]`.
- `[attr~="value"]`—"Space-separated list" attribute selector. Targets by attribute and value, where the attribute value is a space-separated list of values, one of which matches the specified string value. Example: `a[rel="author"]`.
- `[attr|="value"]`—Targets by attribute and value, where the value either matches the specified string `value`, or begins with it and is

immediately followed by a hyphen (`-`). This is useful for the language attribute, which may or may not specify a language subcode (for example, Mexican Spanish, `es-MX`, or Spanish in general, `es`). Example: `[lang|="es"]`.

Case-insensitive attribute selectors

All of the previous attribute selectors are case-sensitive. You can make any of them case-insensitive by adding an `i` before the closing bracket. Example: `input[value="search" i]`.

welcome

Thank you for purchasing the MEAP for *CSS in Depth, Second Edition*. To get the most benefit from this book, you'll want to have some basic familiarity with HTML and CSS. You don't need a significant amount of experience, but I you should have a general understanding of what the two languages are and how they work together. The book also contains a handful of brief JavaScript snippets you'll need to be able to follow along with as well.

If you happen to have years of experience with CSS, this is a book for you too! A lot of experienced developers still find the language frustrating — or maybe just have a lot to catch up on as CSS is rapidly evolving. These topics are a key focus of *CSS in Depth*.

This book follows a carefully planned curriculum for mastering the most essential aspects of CSS. There are countless introductory-level books to CSS, and many others that are laser focused on a specific topic. My goal is to provide something different, something that teaches you how to think in CSS, how to practically use it on real world websites and apps, and how to connect the pieces among the wide array of topics under CSS.

I'll walk you through a deep look at the often-misunderstood fundamentals of the language. You'll learn how to lay out an entire page numerous ways from beginning to end. I'll provide practical advice for organizing your code and keeping it understandable and maintainable. You'll learn how to focus on the fine details of color, typography, and various visual effects. And I'll show you how to work with motion and transformations of elements of the page. And I'll bring you up-to-date with the latest features of CSS — because there is a lot of new stuff to learn!

Please let me know your thoughts in the liveBook discussion forum as you work your way through the book. Your feedback will be invaluable in improving CSS in Depth.

Thanks again for your interest and for purchasing the MEAP!

—Keith J. Grant

In this book

welcome

Thank you for purchasing the MEAP for *CSS in Depth, Second Edition*. To get the most benefit from this book, you'll want to have some basic familiarity with HTML and CSS. You don't need a significant amount of experience, but I you should have a general understanding of what the two languages are and how they work together. The book also contains a handful of brief JavaScript snippets you'll need to be able to follow along with as well.

If you happen to have years of experience with CSS, this is a book for you too! A lot of experienced developers still find the language frustrating — or maybe just have a lot to catch up on as CSS is rapidly evolving. These topics are a key focus of *CSS in Depth*.

This book follows a carefully planned curriculum for mastering the most essential aspects of CSS. There are countless introductory-level books to CSS, and many others that are laser focused on a specific topic. My goal is to provide something different, something that teaches you how to think in CSS, how to practically use it on real world websites and apps, and how to connect the pieces among the wide array of topics under CSS.

I'll walk you through a deep look at the often-misunderstood fundamentals of the language. You'll learn how to lay out an entire page numerous ways from beginning to end. I'll provide practical advice for organizing your code and keeping it understandable and maintainable. You'll learn how to focus on the fine details of color, typography, and various visual effects. And I'll show you how to work with motion and transformations of elements of the page. And I'll bring you up-to-date with the latest features of CSS — because there is a lot of new stuff to learn!

Please let me know your thoughts in the liveBook discussion forum as you work your way through the book. Your feedback will be invaluable in improving CSS in Depth.

Thanks again for your interest and for purchasing the MEAP!

—Keith J. Grant

In this book