SAMPLE CHAPTER

MANNING

CSS IN DEPTH

Keith J. Grant

Foreword by Chris Coyier

www.itbook.store/books/9781617293450

CSS in Depth

by Keith J. Grant

Chapter 2

Copyright 2018 Manning Publications

www.itbook.store/books/9781617293450

brief contents

PART 1 REVIEWING THE FUNDAMENTALS 1

1 ■ Cascade, specificity, and inheritance 3
2 ■ Working with relative units 28
3 ■ Mastering the box model 55

PART 2 MASTERING LAYOUT 85

4 ■ Making sense of floats 87
5 ■ Flexbox 116
6 ■ Grid layout 144
7 ■ Positioning and stacking contexts 177
8 ■ Responsive design 201

PART 3 CSS AT SCALE 231

9 ■ Modular CSS 233
10 ■ Pattern libraries 253

www.itbook.store/books/9781617293450

PART 4 ADVANCED TOPICS 277

11 ■ Backgrounds, shadows, and blend modes 279
12 ■ Contrast, color, and spacing 300
13 ■ Typography 329
14 ■ Transitions 353
15 ■ Transforms 370
16 ■ Animations 396

www.itbook.store/books/9781617293450

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 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 known as *absolute* units; that is, 5 px always means the same thing. 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.

Developers, even experienced CSS developers, often dislike working with relative units, the notorious em included. The way the value of an em can change makes it seem unpredictable and less clear-cut than 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

www.itbook.store/books/9781617293450

work, and I'll show you how to tame their seemingly unpredictable nature. You can make relative values work for you, and wielded correctly, they'll make your code simpler, more versatile, and easier to work with.

2.1 The power of relative values

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, for example, can 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 struggle for pixel-perfect design

In the web environment, the user can have their browser window set 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. We 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-in. phone screen as well as on a 30-in. monitor.

For a long time, designers mitigated this complexity by focusing on "pixel-perfect" designs. They'd create a tightly defined container, often a centered column around 800 px wide. Then, within these constraints, they'd go about designing more or less like their predecessors did with native applications or print publications.

2.1.2 The end of the pixel-perfect web

As technology improved and manufacturers introduced higher-resolution monitors, the pixel-perfect approach slowly started to break down. In the early 2000s, there was a lot of discussion on whether we developers could safely design for displays 1,024 px wide instead of 800 px wide. Then, we'd have the same conversation all over again for 1,280 px. We had to make judgment calls. Was it better to make our site too wide for older computers or too narrow for new ones?

www.itbook.store/books/9781617293450

When smartphones emerged, developers were forced to stop pretending everyone could have the same experience on their sites. Whether we loved it or hated it, we had to abandon columns of some known number of pixels, and begin thinking about *responsive* design. We could no longer hide from the abstraction that comes with CSS. We had to embrace it.

> *responsive*—In CSS, this 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.

Added abstraction means additional complexity. If I 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've multiple ways to solve a particular problem, you'll need to favor the solution that works more generally under multiple and different circumstances.

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.

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), 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 = 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.

www.itbook.store/books/9781617293450

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.

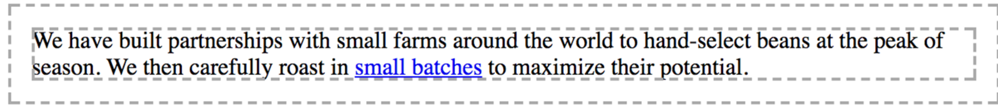

Figure 2.1 Element with 1 em padding (dashed lines added to illustrate padding)

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;        <-- Sets padding on all sides equal to font-size
}
```

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 2 em would produce a computed value of 32 px. 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 will 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.

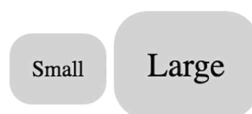

Figure 2.2 Elements with a relatively sized padding and border radius

www.itbook.store/books/9781617293450

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.

Listing 2.2 Appying ems to different elements (HTML)

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

Now, add the styles shown next to your stylesheet. This defines a box using ems. It also defines small and large modifiers, each specifying a different font size.

Listing 2.3 Applying ems applied to different elements (CSS)

```
.box {
  padding: 1em;
  border-radius: 1em;
  background-color: lightgray;
}

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

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

Different font sizes, which will define the elements' em size

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, see figure 2.3. This shows two bits of text, each at a different font size. You'll define these using ems in listing 2.4.

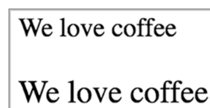

Figure 2.3 Two different font sizes using ems

www.itbook.store/books/9781617293450

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.

Listing 2.4 Relative `font-size` markup

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

The slogan inherits its font size from <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 2.5 Applying ems to `font-size`

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

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

Calculates to 1.2 times the element's inherited font size

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

www.itbook.store/books/9781617293450

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 Element with 1.2 em font and 1.2 em padding

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 2.6 Applying ems to `font-size` and `padding`

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

.slogan {
  font-size: 1.2em;
  padding: 1.2em;
  background-color: #ccc;
}
```

- `font-size: 1.2em;` — **Evaluates to 19.2 px**
- `padding: 1.2em;` — **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.

- Top level
  - Second level
    - Third level
      - Fourth level
        - Fifth level

Figure 2.5 Nested lists with shrinking text

www.itbook.store/books/9781617293450

Shrinking text occurs when you nest lists several levels deep and apply an em-based font size to each level. Listings 2.7 and 2.8 provide an example of this by setting the font size of unordered lists to .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 2.7 Applying ems to a list

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

ul {
  font-size: .8em;
}
```

Listing 2.8 Nested lists

```
<ul>
  <li>Top level
    <ul>
      <li>Second level
        <ul>
          <li>Third level
            <ul>
              <li>Fourth level
                <ul>
                  <li>Fifth level</li>
                </ul>
              </li>
            </ul>
          </li>
        </ul>
      </li>
    </ul>
  </li>
</ul>
```

This list is nested inside the first one, inheriting its font size . . .

. . . and this one is nested inside of that, inheriting the second list's font size . . .

. . . and so on

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 we 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.

- Top level
  - Second level
    - Third level
      - Fourth level
        - Fifth level

Figure 2.6 Nested lists with corrected text

www.itbook.store/books/9781617293450

One way you can accomplish this is with the code in listing 2.9. This sets the font size of the first list to .8 em as before (listing 2.7). 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 2.9 Correcting the shrinking text

```
ul {
  font-size: .8em;
}

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

Lists within lists should have the same font size as their parent.

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 2.10 Specifying font size using rems

```
:root {
  font-size: 1em;
}

ul {
  font-size: .8rem;
}
```

The :root pseudo-class is equivalent to the HTML type selector.

Uses the browser's default size (16 px)

www.itbook.store/books/9781617293450

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 .8 rem, which calculates to 12.8 px. Because this is relative to the root, the font size will remain constant, even if you nest lists.

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 (–), 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. Not only is it harder to find where to set this (usually in the browser's settings page), but changes at this level remain permanent, until the user returns and 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.

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 for most other measures, especially paddings, margins, and border radius (though I favor the use of percentages for container widths when necessary).

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, but again, 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 ems for most other properties.

2.3 Stop thinking in pixels

One pattern, or rather, antipattern, that has been common for the past several years is to reset the font size at the page's root to .625 em or 62.5%.

www.itbook.store/books/9781617293450

Listing 2.11 Antipattern: globally resetting `font-size` to 10 px

```
html {
  font-size: .625em;
}
```

I don't recommend this. 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.

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 most text, so you'll have to override it throughout the page. You'll find yourself setting paragraphs to 1.4 rem and asides to 1.4 rem and `nav` links to 1.4 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 1.4 rem into your code, but in your mind, you're still thinking "14 pixels." On a 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 onscreen, change it. This takes some trial and error, but in reality, so does working with pixels. (In chapter 13, 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 will involve 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 reiterate 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.

www.itbook.store/books/9781617293450

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.

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

Listing 2.12 Setting the true default font size

```
:root {                          <-- Or use the HTML selector
  font-size: 0.875em;            <-- 14/16 (desired px / inherited px) equals .875
}
```

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.

SINGLE-ORIGIN

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.

Figure 2.7 Panel with relative units and an inherited font size

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

Listing 2.13 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.

www.itbook.store/books/9781617293450

Listing 2.14 Panel with relative units

```
.panel {
  padding: 1em;                  Uses ems for padding
  border-radius: 0.5em;          and border radius
  border: 1px solid #999;        <-- Uses 1 px for a thin border
}

.panel > h2 {
  margin-top: 0;                 <-- Removes extra space from the panel
  font-size: 0.8rem;                 top; more on this in chapter 3
  font-weight: bold;             Styles the heading font
  text-transform: uppercase;     using rems for font size
}
```

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 targets an `h2` that's a child element of a `.panel` element. See appendix A for a complete reference of selectors and combinators.

In listing 2.13, 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. This'll make the panel render at different sizes based on the size of the user's screen (shown in figure 2.8).

> *media query*—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. See listing 2.15 for an example; I'll cover this in greater depth in chapter 8.

SINGLE-ORIGIN

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.

SINGLE-ORIGIN

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.

SINGLE-ORIGIN

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.

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

www.itbook.store/books/9781617293450

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

Listing 2.15 Responsive base `font-size`

```
:root {
  font-size: 0.75em;
}

@media (min-width: 800px) {
  :root {
    font-size: 0.875em;
  }
}

@media (min-width: 1200px) {
  :root {
    font-size: 1em;
  }
}
```

- Applies to all screens, but is overridden for larger screens
- Applies only to screens 800 px and wider, overriding the original value
- Applies only to screens 1,200 px and larger, overriding both values

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 (12 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 14 px and 16 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 viewport size. 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 your boss or your client decides the fonts on the site you built are too small or too large, you can change them globally by only touching 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">`.

www.itbook.store/books/9781617293450

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.

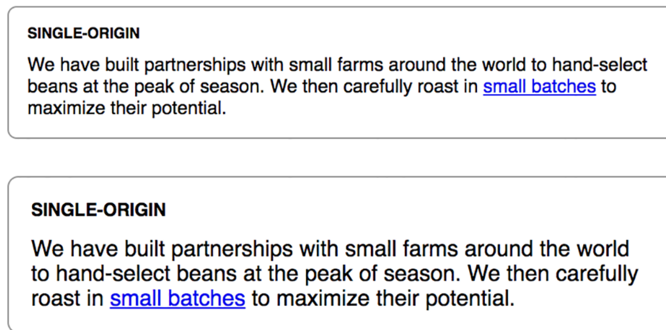

Figure 2.9 A normal panel and a large panel defined on the same page

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 next. Update your stylesheet to match.

Listing 2.16 Creating a larger version of the panel

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

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

Annotations: `font-size: 1rem;` — **Establishes a predictable font size for the component**; `font-size: 0.8em;` — **Uses ems to make other fonts relative to the established parent font size**

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 line of CSS. All you have to do is override the parent element's 1 rem 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.

www.itbook.store/books/9781617293450

Listing 2.17 Scaling the entire panel with one declaration

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

Compound selector targets elements with both panel and large classes

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 only take this one declaration to resize it, as long as everything inside is defined using ems.

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.

> *viewport*—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.

If you're not familiar with viewport-relative units, here is a brief explanation.

- *vh*—1/100th of the viewport height
- *vw*—1/100th of the viewport width
- *vmin*—1/100th of the smaller dimension, height or width (IE9 supports vm instead of vmin)
- *vmax*—1/100th of the larger dimension, height or width (not supported in IE or, at the time of writing, Edge)

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.

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.

www.itbook.store/books/9781617293450

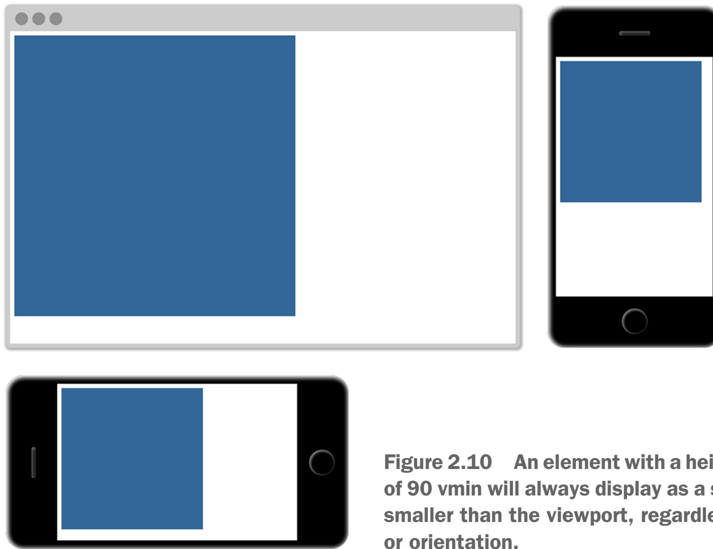

Figure 2.10 An element with a height and width of 90 vmin will always display as a square a little smaller than the viewport, regardless of its size or orientation.

Listing 2.18 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.

NOTE Viewport-relative units are a newer feature for most browsers, so there are a few odd bugs when you use them in more exotic combinations with other styles. See "Known Issues" at http://caniuse.com/#feat=viewport-units for a list.

CSS3

Several of the unit types in this chapter weren't in earlier versions of CSS (rems and viewport-relative units, in particular). They were added amid a series of changes to the language, which is often called CSS3.

In the late 1990s and early 2000s, after initial work on the CSS specification, little changed for a long time. The W3C (World Wide Web Consortium) published the CSS

www.itbook.store/books/9781617293450

2 specification in May 1998. Shortly thereafter, work began on version 2.1 to correct issues and bugs in version 2. Work on CSS 2.1 continued for many years, with few significant additions to the language. It was not finalized as a Proposed Recommendation until April 2011. By this point, browsers had already implemented most of the CSS 2.1 changes, and were well on their way to adding newer features under the moniker CSS3.

The "3" is an informal version number; there's no CSS3 specification. Instead, the specification was broken up into individual modules, each independently versioned. The specification for backgrounds and borders is now separate from the one for box models, 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 and others, such as a flexbox, are at level 1.

As these changes were introduced, we saw an explosion of new features rolling out in browsers from 2009 through 2013. Notable additions at this time included rems and viewport-relative units, as well as new selectors, media queries, web fonts, rounded borders, animations, transitions, transformations, and different ways to specify colors. And, new features are steadily emerging each year.

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. 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 at as such because, as far as the web is concerned, it's all CSS.

2.4.1 Using vw 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: 2vw` 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 7.5 px on an iPhone 6. What would be nice is this scaling effect, but with the extremes a little less severe. You can achieve this with CSS's `calc()` function.

2.4.2 Using calc() for font size

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

www.itbook.store/books/9781617293450

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 vw units. Remove the previous base font size (and the related media queries) from your stylesheet. Add this in its place.

Listing 2.19 Using `calc()` to define `font-size` in ems and vh units

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

Now, open the page and slowly resize your browser. You'll see the font scale smoothly as you do. The 0.5 em here operates as a sort of minimum font size, and the 1 vw adds a responsive scalar. This'll give you a base font size that scales from 11.75 px on an iPhone 6 up to 20 px in a 1,200 px browser window. You can adjust these values to your liking.

You've now 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.

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 only be used 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 stylesheet.

Listing 2.20 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>
```

www.itbook.store/books/9781617293450

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 2.11).

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.

Figure 2.11 Unitless line height is recalculated for each descendant element.

Add listing 2.21 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 2.21 Line height with a unitless number

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

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

Descendant elements inherit the unitless value. (annotation for `line-height: 1.2;`)

If instead you specify the line height using a unit, you may encounter unexpected results, like that shown in figure 2.12. The lines of text overlap one another. Listing 2.22 shows the CSS that generated the overlap.

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.

Figure 2.12 Overlapping lines due to an inherited `line-height`

Listing 2.22 Line height with units results in unexpected output

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

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

Descendant elements inherit the calculated value (19.2 px). (annotation for `line-height: 1.2em;`)

Evaluates to 32 px (annotation for `font-size: 2em;`)

www.itbook.store/books/9781617293450

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.

> *length*—The formal name for a CSS value that denotes a distance measurement. It's a number followed by a unit, such as 5 px. Length comes in two flavors: absolute and relative. Percentages are similar to lengths, but strictly speaking, they're not considered lengths.

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)

In 2015, a long-awaited CSS specification titled *Custom Properties for Cascading Variables* was published as a Candidate Recommendation. This specification introduced the concept of variables to the language, which enabled a new level of dynamic, context-based styles. You can declare a variable and assign it a value; then you can reference this value throughout your stylesheet. You can use this to reduce repetition in your stylesheet, as well as some other beneficial applications as you'll see shortly.

At the time of writing, support for custom properties has rolled out in all major browsers except IE. For up-to-date support information on lesser-known browsers, check "Can I Use" at http://caniuse.com/#feat=css-variables.

> **NOTE** If you happen to use a CSS preprocessor that supports its own variables, such as Sass (syntactically awesome stylesheets) or Less, you may be tempted to disregard CSS variables. Don't. The new 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. Listing 2.23 is an example of a variable declaration. Start a fresh page and stylesheet, and add this CSS.

www.itbook.store/books/9781617293450

Listing 2.23 Defining a custom property

```
: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 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 2.13.

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.

Figure 2.13 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 2.24 Using a custom property

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

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

Sets the font family for paragraphs to Helvetica, Arial, sans-serif

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 2.25 Using custom properties for colors

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

Defines a blue brand-color variable

www.itbook.store/books/9781617293450

```
p {
  font-family: var(--main-font);
  color: var(--brand-color);
}
```

The `var()` function accepts a 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 2.26 Providing fallback values

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

p {
  font-family: var(--main-font, sans-serif);   <- Specifies a fallback value of sans-serif
  color: var(--secondary-color, blue);          <- 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 2.14.

This panel is similar to the one you saw earlier (figure 2.7). The HTML for this is shown in listing 2.27. It has two instances of the panel: one inside the body and one inside a dark section. Update your HTML to match this.

www.itbook.store/books/9781617293450

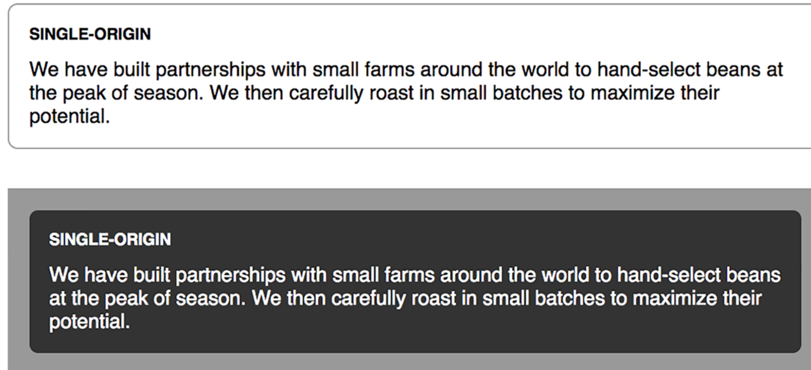

Figure 2.14 Custom properties produce different colored panels based on local variable values.

Listing 2.27 Two panels in different contexts on the page

```
<body>
  <div class="panel">                          <- A regular panel
    <h2>Single-origin</h2>                        on the page
    <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">                            The second panel inside
    <div class="panel">                           a dark container
      <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>
```

Let's 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.

www.itbook.store/books/9781617293450

Listing 2.28 Using variables to define the panel colors

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

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

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

Defines background and text color variables as white and black, respectively

Uses the variables in the panel's styles

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 2.29 Styling the dark container

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

Puts a margin between the dark container and the preceding panel

Applies a dark gray background to the dark container

Redefines the --main-bg and --main-color variables within the scope of the container

Reload the page, and the second panel will have a dark background and white text. This is because when the panel uses these variables, they'll 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.

www.itbook.store/books/9781617293450

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.

2.6.2 Changing custom properties with JavaScript

Custom properties can also be accessed and manipulated live in the browser using JavaScript. Because this isn't a book on JavaScript, I'll show you enough to get acquainted with the concept. I'll leave it to you to integrate this into your JavaScript projects.

The following listing shows how to access a property on an element. It adds a script to the page, which logs the value of the root element's `--main-bg` property.

Listing 2.30 Accessing a custom property in JavaScript

```
<script type="text/javascript">
  var rootElement = document.documentElement;
  var styles = getComputedStyle(rootElement);
  var mainColor = styles.getPropertyValue('--main-bg');
  console.log(String(mainColor).trim());
</script>
```

Gets the styles object for an element

Gets the --main-bg value from the styles object

Ensures mainColor is a String and trims whitespace; logs "#fff"

Because you can specify new values for custom properties on the fly, you can use JavaScript to set a new value for `--main-bg` dynamically. If you set it to a light blue, it'll appear as shown in figure 2.15.

SINGLE-ORIGIN

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.

Figure 2.15 JavaScript can set the panel's background by changing the `--main-bg` variable.

The code in the next listing sets a new value to `--main-bg` on the root element. Add this at the end of the `<script>` tag.

Listing 2.31 Setting a custom property in JavaScript

```
var rootElement = document.documentElement;
rootElement.style.setProperty('--main-bg', '#cdf');
```

Sets --main-bg to a light blue on the root element

www.itbook.store/books/9781617293450

If you run this script, any elements inheriting the `--main-bg` property will update to use this new value. On your page, this changes the background of the first panel to light blue. The second panel remains unchanged, as it's still inheriting the property from the dark container.

With this technique, you can use JavaScript to re-theme your site, live in the browser. Or, you could highlight certain parts of the page or make any number of other on-the-fly changes. Using only a few lines of JavaScript, you can make changes that'll affect a large number of elements on the page.

2.6.3 Experimenting with custom properties

Custom properties are a whole new area of CSS that developers are just beginning to explore. Because browser support has been limited, it hasn't yet seen much "prime-time" use. I'm sure that over time, you'll see best practices and novel uses emerge. This is something to keep your eye on. Experiment with custom properties and see what you can come up with.

Be aware that any declaration using `var()` will be ignored by old browsers that don't understand it. Provide a fallback behavior for those browsers when possible:

```
color: black;
color: var(--main-color);
```

This will not always be possible, however, given the dynamic nature of custom properties. Keep an eye on browser support at http://caniuse.com.

Summary

- Embrace the use of relative units, allowing the page's structure to determine the meaning of your styles.
- Favor the use of rems for font size, but selectively use ems for simple scaling of components on the page.
- You can make your entire page scale responsively without any media queries.
- Use unitless values when specifying line height.
- You can start getting familiar with one of CSS's newest features, custom properties.

www.itbook.store/books/9781617293450

WEB DEVELOPMENT/HTML

CSS IN DEPTH

Keith J. Grant

Some websites really pop. They look great, they're visually consistent, and they feel interactive and responsive. You can bet their developers knew CSS in depth. CSS specifies everything from the structural layout of page elements to their individual look and feel. True masters know the patterns of CSS development, the techniques to implement them, and the subtle touches that result in beautiful typography, fluid transitions, and balanced graphics. Join them!

CSS in Depth exposes you to a world of CSS techniques that range from clever to mind-blowing. This instantly useful book is packed with creative examples and powerful best practices that will sharpen your technical skills and inspire your sense of design. You'll gain new insights into familiar features like floats and units, and experiment with emerging ideas like responsive design and pattern libraries. Bottom line: this book will make you a better web designer and your apps will look fantastic!

What's Inside

- Avoid common CSS pitfalls
- Master misunderstood concepts
- Use flexbox and grid layout
- Responsive designs for any device
- Code for reuse and maintainability

Written for web developers who know the basics of CSS and HTML.

Keith J. Grant is a senior web developer who builds and maintains web applications and websites, including The New York Stock Exchange site.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/css-in-depth

"Become better at writing code that lasts and is understandable and performant."
—From the Foreword by Chris Coyier
Cofounder of CodePen

"From zero to hero in CSS!"
—Pierfrancesco D'Orsogna
GamePix

"The bible of the most up-to-date CSS."
—Phily Austria
Faraday Future

"A well-written, concise book. I enjoyed every minute of reading it."
—Tanya Wilke, Sanlam

"A clear and complete guide to CSS."
—Giancarlo Massari, Unic

ISBN-13: 978-1-61729-345-0
ISBN-10: 1-61729-345-8

$44.99 / Can $59.99 [INCLUDING eBOOK]

www.itbook.store/books/9781617293450