III

CSS FOR PRESENTATION

CHAPTER
11

INTRODUCING CASCADING STYLE SHEETS

IN THIS CHAPTER

The benefits and power of CSS

How HTML markup creates a document structure

Writing style rules

Attaching styles to the HTML document

Big concepts: inheritance, specificity, the cascade, rule order, and the box model

You've heard style sheets mentioned quite a bit already, and now we'll finally put them to work and start giving our pages some much-needed style. Cascading Style Sheets (CSS) is the W3C standard for defining the presentation of documents written in HTML, and in fact, any XML language. Presentation, again, refers to the way the document is delivered to the user, whether shown on a computer screen, displayed on a cell phone, printed on paper, or read aloud by a screen reader. With style sheets handling the presentation, HTML can handle the business of defining document structure and meaning, as intended.

CSS is a separate language with its own syntax. This chapter covers CSS terminology and fundamental concepts that will help you get your bearings for the upcoming chapters, where you'll learn how to change text and font styles, add colors and backgrounds, and even do basic page layout. By the end of **Part III**, I aim to give you a solid foundation for further reading on your own and lots of practice.

THE BENEFITS OF CSS

Not that you need further convincing that style sheets are the way to go, but here is a quick rundown of the benefits of using style sheets.

- **Precise type and layout controls.** You can achieve print-like precision using CSS. There is even a set of properties aimed specifically at the printed page (but we won't be covering them in this book).
- **Less work.** You can change the appearance of an entire site by editing one style sheet. This also ensures consistency of formatting throughout the site.

- **More accessible sites.** When all matters of presentation are handled by CSS, you can mark up your content meaningfully, making it more accessible for non-visual or mobile devices.

Come to think of it, there really aren't any disadvantages to using style sheets. There are some lingering hassles from browser inconsistencies, but they can either be avoided or worked around if you know where to look for them.

The Power of CSS

We're not talking about minor visual tweaks here, like changing the color of headlines or adding text indents. When used to its full potential, CSS is a robust and powerful design tool. My eyes were first opened to the possibilities of using CSS for design by the variety and richness of the designs at CSS Zen Garden (*www.csszengarden.com*).

In the misty days of yore (2003), when developers were still hesitant to give up their table-based layouts for CSS, David Shea's CSS Zen Garden site demonstrated exactly what could be accomplished using CSS alone. David posted an HTML document and invited designers to contribute their own style sheets that gave the document a visual design. FIGURE 11–1 shows just a few of my favorites. All of these designs use the *exact same* HTML source document.

Not only that, they don't include a single `img` element (all of the images are in the background of elements). But look at how different each page looks—and how sophisticated. That's all done with style sheets. It is proof of the power in keeping CSS separate from HTML, and presentation separate from structure.

The CSS Zen Garden is no longer being updated and now is considered a historical document of a turning point in the adoption of web standards. Despite its age, I still find it to be a nice one-stop lesson for demonstrating exactly what CSS can do.

Granted, it takes a lot of practice to be able to create CSS layouts like those shown in FIGURE 11–1. Killer graphic design skills help too (unfortunately, you won't get those in this book). I'm showing this to you up front because I want you to be aware of the potential of CSS-based design, particularly because the examples in this beginners' book tend to be simple and straightforward. Take your time learning, but keep your eye on the prize.

HOW STYLE SHEETS WORK

It's as easy as 1-2-3!

1. Start with a document that has been marked up in HTML.
2. Write style rules for how you ' d like certain elements to look.

CSS Zen Dragen
by Matthew Buchanan

By the Pier
by Peter Ong Kelmscott

Organica Creativa
by Eduardo Cesario

Shaolin Yokobue
by Javier Cabrera

FIGURE 11-1. These pages from the CSS Zen Garden use the same HTML source document, but the design is changed with CSS alone (used with permission of CSS Zen Garden and the individual designers).

3. Attach the style rules to the document. When the browser displays the document, it follows your rules for rendering elements (unless the user has applied some mandatory styles, but we'll get to that later).

OK, so there's a bit more to it than that, of course. Let's give each of these steps a little more consideration.

1. Marking Up the Document

You know a lot about marking up content from the previous chapters. For example, you know that it is important to choose elements that accurately describe the meaning of the content. You also heard me say that the markup

EXERCISE 11-1.
A first look

In this chapter, we'll add a few simple styles to a short article. The document, *cooking.html*, and its associated image, *salads.jpg*, are available at *learningwebdesign.com/5e/materials/*.

For now, just open the document in a browser to see how it looks by default (it should look something like FIGURE 11-2). You can also open the document in a text editor to get ready to follow along in the next two exercises.

Cooking with Daniel from Nada Surf

I had the pleasure of spending a crisp, Spring day in Portsmouth, NH cooking and chatting with Daniel Lorca of the band Nada Surf as he prepared a gourmet, sit-down dinner for 28 pals.

When I first invited Nada Surf to be on the show, I was told that Daniel Lorca was the guy I wanted to talk to. Then Daniel emailed his response: "I'm way into it, but i don't want to talk about it, i wanna do it." After years of only having access to touring bands between their sound check and set, I've been doing a lot of *talking* about cooking with rockstars. To actually cook with a band was a dream come true.

Six-hour Salad

Daniel prepared a salad of arugula, smoked tomatoes, tomato jam, and grilled avocado (it's as good as it sounds!). I jokingly called it "6-hour Salad" because that's how long he worked on it. The fresh tomatoes were slowly smoked over woodchips in the grill, and when they were softened, Daniel separated out the seeds which he reduced into a smoky jam. The tomatoes were cut into strips to put on the salads. As the day meandered, the avocados finally went on the grill after dark. I was on flashlight duty while Daniel checked for the perfect grill marks.

I wrote up a streamlined adaptation of his recipe that requires *much* less time and serves 6 people instead of *five*times that amount.

The Main Course

In addition to the smoky grilled salad, Daniel served tarragon cornish hens with a cognac cream sauce loaded with chanterelles and grapes, and wild rice with grilled ramps (wild garlicky leeks). Dinner was served close to midnight, but it was a party so nobody cared.

We left that night (technically, early the next morning) with full bellies, new cooking tips, and nearly 5 hours of footage. I'm considering renaming the show "Cooking with Nada Surf".

FIGURE 11-2. This is what the article looks like without any style sheet instructions. Although we won't be making it beautiful, you will get a feel for how style sheets work.

creates the structure of the document, sometimes called the structural layer, upon which the presentation layer can be applied.

In this and the upcoming chapters, you'll see that having an understanding of your document's structure and the relationships between elements is central to your work as a style sheet author.

In the exercises throughout this chapter you will get a feel for how simple it is to change the look of a document with style sheets. The good news is that I've whipped up a little HTML document for you to play with. You can get acquainted with the page we'll be working with in EXERCISE 11-1.

2. Writing the Rules

A style sheet is made up of one or more style instructions (called style rules) that describe how an element or group of elements should be displayed. The first step in learning CSS is to get familiar with the parts of a rule. As you'll see, they're fairly intuitive to follow. Each rule *selects* an element and *declares* how it should look.

The following example contains two rules. The first makes all the **h1** elements in the document green; the second specifies that the paragraphs should be in a large, sans-serif font. Sans-serif fonts do not have a little slab (a serif) at the ends of strokes and tend to look more sleek and modern.

```
h1 { color: green; }
p  { font-size: large; font-family: sans-serif; }
```

In CSS terminology, the two main sections of a rule are the selector that identifies the element or elements to be affected, and the declaration that provides the rendering instructions. The declaration, in turn, is made up of a property (such as **color**) and its value (**green**), separated by a colon and a space. One or more declarations are placed inside curly brackets, as shown in FIGURE 11-3.

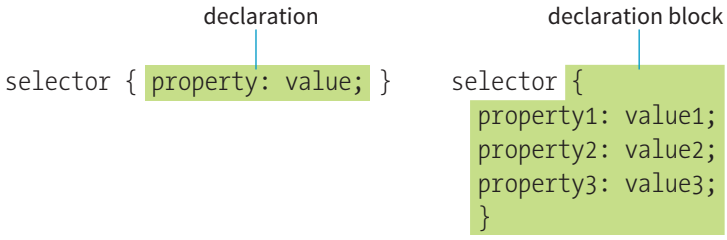


FIGURE 11-3. The parts of a style rule.

Selectors

In the previous small style sheet example, the `h1` and `p` elements are used as selectors. This is called an element type selector, and it is the most basic type of selector. The properties defined for each rule will apply to every `h1` and `p` element in the document, respectively.

Another type of selector is an ID selector, which selects an element based on the value of an element's `id` attribute. It is indicated with the `#` symbol. For example, the selector `#recipe` targets an element with `id="recipe"`.

In upcoming chapters, I'll introduce you to more sophisticated selectors that you can use to target elements, including ways to select groups of elements, and elements that appear in a particular context. See the **"Selectors in this Book"** sidebar for details.

Mastering selectors—that is, choosing the best type of selector and using it strategically—is an important step in mastering CSS.

Selectors in This Book

Instead of throwing the selectors at you all at once, I've spread them out so you can master a few at a time. Here is where you will find them:

Chapter 11:

Chapter 12:

Chapter 13:

Declarations

The declaration is made up of a property/value pair. There can be more than one declaration in a single rule; for example, the rule for the `p` element shown earlier in the code example has both the `font-size` and `font-family` properties. Each declaration must end with a semicolon to keep it separate from the following declaration (see **Note**). If you omit the semicolon, the declaration and the one following it will be ignored. The curly brackets and the declarations they contain are often referred to as the declaration block (FIGURE 11-3).

Because CSS ignores whitespace and line returns within the declaration block, authors typically write each declaration in the block on its own line, as shown in the following example. This makes it easier to find the properties applied to the selector and to tell when the style rule ends.

```
p {
  font-size: large;
  font-family: sans-serif;
}
```

Note that nothing has really changed here—there is still one set of curly brackets, semicolons after each declaration, and so on. The only difference is the insertion of line returns and some character spaces for alignment.

NOTE

Technically, the semicolon is not required after the last declaration in the block, but it is recommended that you get into the habit of always ending declarations with a semicolon. It will make adding declarations to the rule later that much easier.

Properties

The heart of style sheets lies in the collection of standard properties that can be applied to selected elements. The complete CSS specification defines dozens of properties for everything from text indents to how table headers should be read aloud. This book covers the most common and best-supported properties that you can begin using right away.

Values

Values are dependent on the property. Some properties take length measurements, some take color values, and others have a predefined list of keywords. When you use a property, it is important to know which values it accepts; however, in many cases, simple common sense will serve you well. Authoring tools such as Dreamweaver or Visual Studio provide hints of suitable values to choose from. Before we move on, why not get a little practice writing style rules yourself in EXERCISE 11-2?

EXERCISE 11-2. Your first style sheet

Open *cooking.html* in a text editor. In the **head** of the document you will find that I have set up a **style** element for you to type the rules into. The **style** element is used to embed a style sheet in an HTML document. To begin, we'll simply add the small style sheet that we just looked at in this section. Type the following rules into the document, just as you see them here:

```
<style>
h1 {
  color: green;
}
p {
  font-size: large;
  font-family: sans-serif;
}
</style>
```

Save the file, and take a look at it in the browser. You should notice some changes (if your browser already uses a sans-serif font, you may see only a size change). If not, go back and check that you included both the opening and closing curly bracket and semicolons. It's easy to accidentally omit these characters, causing the style sheet not to work.

Now we'll edit the style sheet to see how easy it is to write rules and see the effects of the changes. Here are a few things to try.

IMPORTANT: Remember that you need to save the document after each change in order for the changes to be visible when you reload it in the browser.

- Make the **h1** element "gray" and take a look at it in the browser. Then make it "blue". Finally, make it "orange". (We'll run through the complete list of available color names in **Chapter 13, Colors and Backgrounds**.)
- Add a new rule that makes the **h2** elements orange as well.
- Add a 100-pixel left margin to paragraph (**p**) elements by using this declaration:

```
margin-left: 100px;
```

Remember that you can add this new declaration to the existing rule for **p** elements.

- Add a 100-pixel left margin to the **h2** headings as well.
- Add an orange, 1-pixel border to the bottom of the **h1** element by using this declaration:

```
border-bottom: 1px solid orange;
```

- Move the image to the right margin, and allow text to flow around it with the **float** property. The shorthand **margin** property shown in this rule adds zero pixels of space on the top and bottom of the image and 12 pixels of space on the left and right of the image (the values are mirrored in a manner explained in **Chapter 14, Thinking Inside the Box**):

```
img {
  float: right;
  margin: 0 12px;
}
```

When you are done, the document should look something like the one shown in FIGURE 11-4.

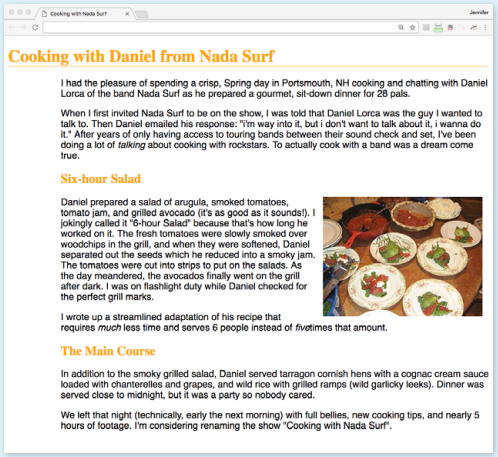


FIGURE 11-4. The article after we add a small style sheet. Not beautiful—just different.

3. Attaching the Styles to the Document

In the previous exercise, we embedded the style sheet right in the document by using the **style** element. That is just one of three ways that style information can be applied to an HTML document. You'll get to try out each of these soon, but it is helpful to have an overview of the methods and terminology up front.

External style sheets

An external style sheet is a separate, text-only document that contains a number of style rules. It must be named with the *.css* suffix. The *.css* document is then linked to (via the **link** element) or imported (via an **@import** rule in a style sheet) into one or more HTML documents. In this way, all the files in a website may share the same style sheet. This is the most powerful and preferred method for attaching style sheets to content. We'll discuss external style sheets more and start using them in the exercises in **Chapter 13**.

Embedded style sheets

This is the type of style sheet we worked with in the exercise. It is placed in a document via the **style** element, and its rules apply only to that document. The **style** element must be placed in the **head** of the document. This example also includes a comment (see the **"Comments in Style Sheets"** sidebar).

```
<head>
   <title>Required document title here</title>
   <style>
     /* style rules go here */
   </style>
</head>
```

Inline styles

You can apply properties and values to a single element by using the **style** *attribute* in the element itself, as shown here:

```
<h1 style="color: red">Introduction</h1>
```

To add multiple properties, just separate them with semicolons, like this:

```
<h1 style="color: red; margin-top: 2em">Introduction</h1>
```

Inline styles apply only to the particular element in which they appear. Inline styles should be avoided, unless it is absolutely necessary to override styles from an embedded or external style sheet. Inline styles are problematic in that they intersperse presentation information into the structural markup. They also make it more difficult to make changes because every **style** attribute must be hunted down in the source.

EXERCISE 11-3 gives you an opportunity to write an inline style and see how it works. We won't be working with inline styles after this point for the reasons listed earlier, so here's your chance.

Comments in Style Sheets

Sometimes it is helpful to leave yourself or your collaborators comments in a style sheet. CSS has its own comment syntax, shown here:

```
/* comment goes here */
```

Content between the **/*** and ***/** will be ignored when the style sheet is parsed, which means you can leave comments anywhere in a style sheet, even within a rule:

```
body {
   font-size: small;
  /* change this later */
}
```

One use for comments is to label sections of the style sheet to make things easier to find later; for example:

```
/* FOOTER STYLES */
```

CSS comments are also useful for temporarily hiding style declarations in the design process. When I am trying out a number of styles, I can quickly switch styles off by enclosing them in **/*** and ***/**, check the design in a browser, then remove the comment characters to make the style appear again. It's much faster than retyping the entire thing.

EXERCISE 11-3.
Applying an inline style

Open the article *cooking.html* in whatever state you last left it in EXERCISE 11-2. If you worked to the end of the exercise, you will have a rule that makes the **h2** elements orange.

Write an inline style that makes the second **h2** gray. We'll do that right in the opening **h2** tag by using the **style** attribute, as shown here:

```
<h2 style="color: gray">The
Main Course</h2>
```

Note that it must be gray-with-an-a (not grey-with-an-e) because that is the way the color is defined in the spec.

Save the file and open it in a browser. Now the second heading is gray, overriding the orange color set in the embedded style sheet. The other **h2** heading is unaffected.

THE BIG CONCEPTS

There are a few big ideas that you need to get your head around to be comfortable with how Cascading Style Sheets behave. I'm going to introduce you to these concepts now so we don't have to slow down for a lecture once we're rolling through the style properties. Each of these ideas will be revisited and illustrated in more detail in the upcoming chapters.

Inheritance

Are your eyes the same color as your parents'? Did you inherit their hair color? Well, just as parents pass down traits to their children, styled HTML elements pass down certain style properties to the elements they contain. Notice in EXERCISE 11-1, when we styled the **p** elements in a large, sans-serif font, the **em** element in the second paragraph became large and sans-serif as well, even though we didn't write a rule for it specifically (FIGURE 11-5). That is because the **em** element inherited the styles from the paragraph it is in. Inheritance provides a mechanism for styling elements that don't have any explicit styles rules of their own.

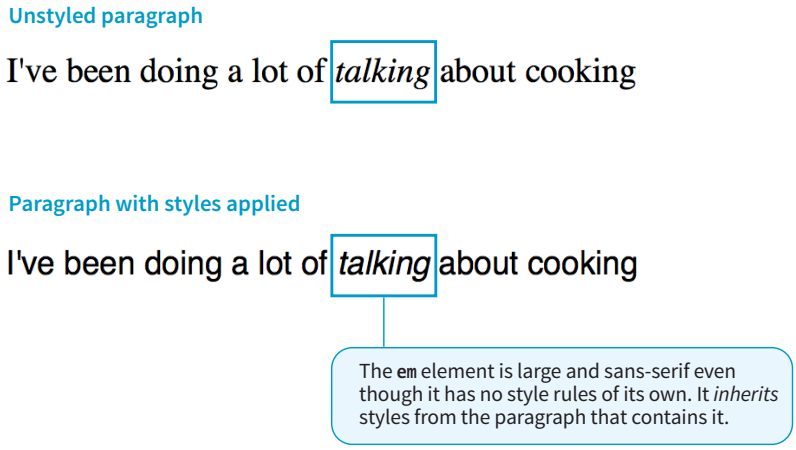


FIGURE 11-5. The **em** element inherits styles that were applied to the paragraph.

Document structure

This is where an understanding of your document's structure becomes important. As I've noted before, HTML documents have an implicit structure, or hierarchy. For example, the sample article we've been playing with has an **html** root element that contains a **head** and a **body**, and the **body** contains heading and paragraph elements. A few of the paragraphs, in turn, contain inline elements such as images (**img**) and emphasized text (**em**). You can visualize the structure as an upside-down tree, branching out from the root, as shown in FIGURE 11-6.

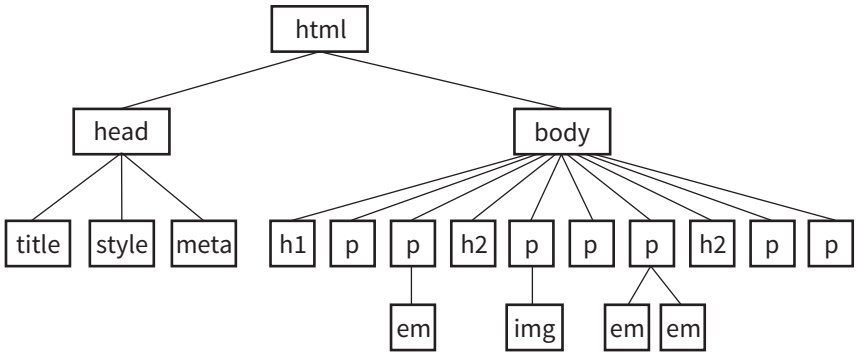


FIGURE 11-6. The document tree structure of the sample document, *cooking.html*.

Parents and children

The document tree becomes a family tree when it comes to referring to the relationship between elements. All the elements contained within a given element are said to be its descendants. For example, the **h1**, **h2**, **p**, **em**, and **img** elements in the document in FIGURE 11-6 are all descendants of the **body** element.

An element that is directly contained within another element (with no intervening hierarchical levels) is said to be the child of that element. Conversely, the containing element is the parent. For example, the **em** element is the child of the **p** element, and the **p** element is its parent.

All of the elements higher than a particular element in the hierarchy are its ancestors. Two elements with the same parent are siblings. We don't refer to "aunts" or "cousins," so the analogy stops there. This may all seem academic, but it will come in handy when you're writing CSS selectors.

Pass it on

When you write a font-related style rule using the **p** element as a selector, the rule applies to all of the paragraphs in the document as well as the inline text elements they contain. We've seen the evidence of the **em** element inheriting the style properties applied to its parent (**p**) back in FIGURE 11-5. FIGURE 11-7 demonstrates what's happening in terms of the document structure diagram. Note that the **img** element is excluded because font-related properties do not apply to images.

Notice that I've been saying "certain" properties are inherited. It's important to note that some style sheet properties inherit and others do not. In general, properties related to the styling of text—font size, color, style, and the like—are passed down. Properties such as borders, margins, backgrounds, and so on that affect the boxed area around the element tend not to be passed down. This makes sense when you think about it. For example, if you put a border

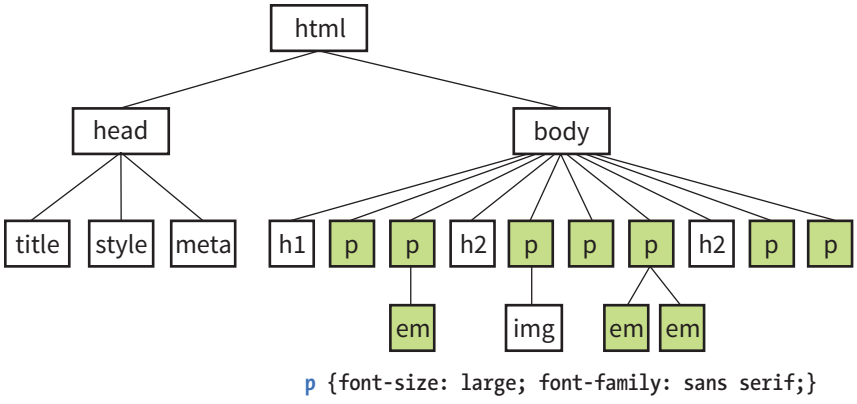


FIGURE 11-7. Certain properties applied to the **p** element are inherited by their children.

around a paragraph, you wouldn't want a border around every inline element (such as **em**, **strong**, or **a**) it contains as well.

You can use inheritance to your advantage when writing style sheets. For example, if you want all text elements to be blue, you could write separate style rules for every element in the document and set the **color** to "blue". A *better* way would be to write a single style rule that applies the **color** property to the **body** element, and let all the elements contained in the **body** inherit that style (FIGURE 11-8).

Any property applied to a specific element overrides the inherited values for that property. Going back to the article example, if we specify that the **em** element should be orange, that would override the inherited blue setting.

CSS TIP

When you learn a new property, it is a good idea to note whether it inherits. Inheritance is noted for every property listing in this book. For the most part, inheritance follows your expectations.

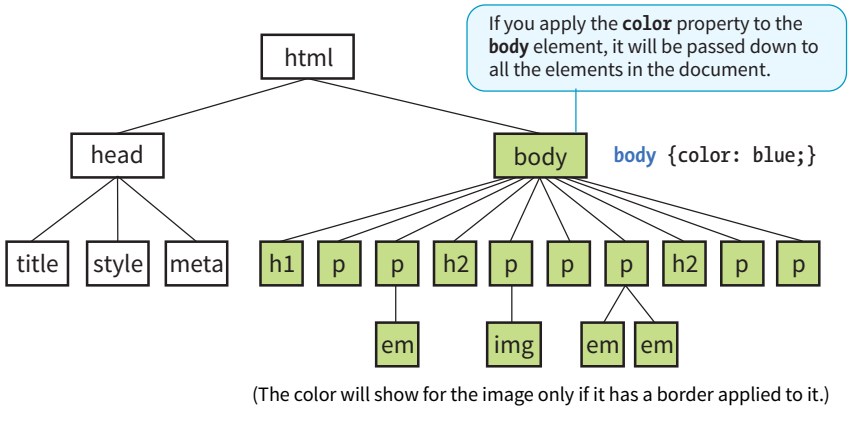


FIGURE 11-8. All the elements in the document inherit certain properties applied to the **body** element.

Conflicting Styles: The Cascade

Ever wonder why they are called "cascading" style sheets? CSS allows you to apply several style sheets to the same document, which means there are bound to be conflicts. For example, what should the browser do if a document's imported style sheet says that `h1` elements should be red, but its embedded style sheet has a rule that makes `h1`s purple? The two style rules with `h1` selectors have equal weight, right?

The folks who wrote the style sheet specification anticipated this problem and devised a hierarchical system that assigns different weights to the various sources of style information. The cascade refers to what happens when several sources of style information vie for control of the elements on a page: style information is passed down ("cascades" down) until it is overridden by a style rule with more weight. Weight is considered based on the *priority* of the style rule source, the *specificity* of the selector, and *rule order*.

The "cascade" refers to what happens when several sources of style information vie for control of the elements on a page.

Priority

If you don't apply any style information to a web page, it renders according to the browser's internal style sheet. We've been calling this the default rendering; the W3C calls it the user agent style sheet. Individual users can apply their own styles as well (the user style sheet, also called the reader style sheet), which override the default styles in their browser. However, if the author of the web page has attached a style sheet (the author style sheet), that overrides both the user and the user agent styles. The sidebar **"Style Rule Hierarchy"** provides an overview of the cascading order from highest to lowest priority.

The only exception is if the user has identified a style as "important," in which case that style will override all competing styles (see the **"Assigning Importance"** sidebar). This permits users to keep settings accommodating a disability such as extra large type for sight impairment.

Style Rule Hierarchy

Style information can come from various origins, listed here from highest priority to lowest. In other words, items higher in the list override items below.

- Any style rule marked `!important` by the reader (user)
- Any style rule marked `!important` by the author
- Style sheets written by the author
- Style sheets created by the reader (user)
- Browser's default style rules ("user agent style sheet")

Specificity

It is possible for conflicts to arise in which an element is getting style instructions from more than one rule. For example, there may be a rule that applies to paragraphs and another rule for a paragraph that has the ID "intro." Which rule should the intro paragraph use?

When two rules in a style sheet conflict, the type of selector is used to determine the winner. The more specific the selector, the more weight it is given to override conflicting declarations. In our example, the selector that includes the ID name (`#intro`) is more specific than a general element selector (like `p`), so that rule would apply to the "intro" paragraph, overriding the rules set for all paragraphs.

When two rules in a single style sheet conflict, the type of selector is used to determine the winner.

It's a little soon to be discussing specificity because we've looked at only two types of selectors. For now, put the term specificity and the concept that some

Assigning Importance

If you want a rule not to be overridden by a subsequent conflicting rule, include the **!important** indicator just after the property value and before the semicolon for that rule. For example, to guarantee paragraph text will be blue, use the following rule:

```
p {color: blue !important;}
```

Even if the browser encounters an inline style later in the document (which should override a document-wide style sheet), like this one:

```
<p style="color: red">
```

that paragraph will still be blue because the rule with the **!important** indicator cannot be overridden by other styles in the author's style sheet.

The only way an **!important** rule may be overridden is by a conflicting rule in a reader (user) style sheet that has also been marked **!important**. This is to ensure that special reader requirements, such as large type or high-contrast text for the visually impaired, are never overridden.

Based on the previous examples, if the reader's style sheet includes this rule

```
p {color: black;}
```

the text would still be blue because all author styles (even those not marked **!important**) take precedence over the reader's styles. However, if the conflicting reader's style is marked **!important**, like this

```
p {color: black !important;}
```

the paragraphs will be black and cannot be overridden by any author-provided style.

Beware that the **!important** indicator is not a get-out-of-jail-free card. Best practices dictate that it should be used sparingly, if at all, and certainly never just to get yourself out of a sticky situation with inheritance and the cascade.

selectors have more "weight," and therefore override others, on your radar. We will revisit specificity in much more detail in **Chapter 12, Formatting Text** when you have more selector types under your belt.

Rule order

The cascade follows a "last one wins" rule. Whichever rule appears last has the last word.

After all the style sheet sources have been sorted by priority, and after all the linked and imported style sheets have been shuffled into place, there are likely to be conflicts in rules with equal weights. When that is the case, the order in which the rules appear is important. The cascade follows a "last one wins" rule. Whichever rule appears last has the last word.

Within a style sheet, if there are conflicts within style rules of identical weight, whichever one comes last in the list "wins." Take these three rules, for example:

```
<style>
  p { color: red; }
  p { color: blue; }
  p { color: green; }
</style>
```

In this scenario, paragraph text will be green because the last rule in the style sheet—that is, the one closest to the content in the document—overrides the earlier ones. Procedurally, the paragraph is assigned a color, then assigned a new one, and finally a third one (green) that gets used. The same thing happens when conflicting styles occur within a single declaration stack:

```
<style>
  p { color: red;
      color: blue;
      color: green; }
</style>
```

The resulting color will be green because the last declaration overrides the previous two. It is easy to accidentally override previous declarations within a rule when you get into compound properties, so this is an important behavior to keep in mind. That is a very simple example. What happens when style sheet rules from different sources come into play?

Let's consider an HTML document that has an embedded style sheet (added with the **style** element) that starts with an **@import** rule for importing an external *.css* file. That same HTML document also has a few inline **style** attributes applied to particular **h1** elements.

STYLE DOCUMENT (*external.css*):

```
…
h1 { color: red }
…
```

HTML DOCUMENT:

```
<!DOCTYPE html>
<html>
<head>
  <title>…</title>
  <style>
    @import url(external.css);  /* set to red first */
    h1 { color: purple;}  /* overridden by purple */
  </style>
</head>
<body>
  <h1 style="color: blue">Heading</h1>  /* blue comes last and wins */
  …
</body>
</html>
```

When the browser parses the file, it gets to the imported style sheet first, which sets **h1**s to red. Then it finds a rule with equal weight in the embedded style sheet that overrides the imported rule, so **h1**s are set to purple. As it continues, it encounters a style rule right in an **h1** that sets its color to blue. Because that rule came last, it's the winner, and that **h1** will be blue. That's the effect we witnessed in EXERCISE 11-3. Note that other **h1**s in this document without inline style rules would be purple, because that was the last **h1** color applied to the whole document.

The Box Model

As long as we're talking about Big CSS Concepts, it is only appropriate to introduce the cornerstone of the CSS visual formatting system: the box model. The easiest way to think of the box model is that browsers see every element on the page (both block and inline) as being contained in a little

Using Rule Order for Fallbacks

Many CSS properties are tried and true and are supported by all browsers; however, there are always useful, new properties emerging that take a while to be implemented by browsers. It is common for just one or two browsers to support a new feature and for others to lag behind or never support it at all. It also takes a long time for some old browsers to completely fade from existence.

Fortunately, there are a number of ways to provide fallbacks (alternative styles using better-supported properties) to non-supporting browsers. The most straightforward method takes advantage of browsers' built-in behavior of ignoring any declaration they don't understand and then using rule order strategically.

In this example, I have added a decorative border image to an element by using the **border-image** property and provided a fallback solid border with the tried-and-true **border** property. Supporting browsers use the image because it is the last rule in the stack. Non-supporting browsers set a solid border but stop there when they get to the **border-image** property they don't understand. They won't crash or throw an error. They just ignore it. The border displays as the fallback solid red line on those browsers, which is fine, but users with supporting browsers will see the decorative border as intended.

```
h1 {
/* fallback first */
  border: 25px solid #eee;
/* newer technique */
border-image: url(fancyframe.
png) 55 fill / 55px / 25px;
}
```

You'll see this method of providing fallbacks by putting newer properties last throughout this book.

A Quick History of CSS

The first official version of CSS (the **CSS Level 1 Recommendation**, a.k.a **CSS1**) was released in 1996, and included properties for adding font, color, and spacing instructions to page elements. Unfortunately, lack of browser support prevented the widespread adoption of CSS for several years.

CSS Level 2 (CSS2), released in 1998, most notably added properties for positioning that allowed CSS to be used for page layout. It also introduced styles for other media types (such as print and handheld) and more sophisticated methods for selecting elements. **CSS Level 2, Revision 1 (CSS2.1)** made some minor adjustments to CSS2 and became a Recommendation in 2011.

CSS Level 3 (CSS3) is different from prior versions in that it is divided into individual modules, each addressing a feature such as animation, multiple column layouts, or borders. While some modules are being standardized, others remain experimental. In that way, browser developers can begin implementing (and we can begin using!) one feature at a time instead of waiting for an entire specification to be "ready."

Now that each CSS module is on its own track, modules have their own Level numbers. No more big, all-encompassing CSS versions. Newly introduced modules, such as the Grid Layout Module, start out at Level 1. Modules that have been around a while may have already reached Level 4.

You won't believe how many individual specifications are in the works! For an overview of the specifications in their various states of "doneness," see the W3C's CSS current work page at *www.w3.org/Style/CSS/current-work*.

rectangular box. You can apply properties such as borders, margins, padding, and backgrounds to these boxes, and even reposition them on the page.

We're going to go into a lot more detail about the box model in **Chapter 14**, but having a general feel for it will benefit you even as we discuss text and backgrounds in the following two chapters.

To see the elements roughly the way the browser sees them, I've written style rules that add borders around every content element in our sample article:

```
h1 { border: 1px solid blue; }
h2 { border: 1px solid blue; }
p { border: 1px solid blue; }
em { border: 1px solid blue; }
img { border: 1px solid blue; }
```

FIGURE 11-9 shows the results. The borders reveal the shape of each block element box. There are boxes around the inline elements (`em` and `img`) as well. If you look at the headings, you will see that block element boxes expand to fill the available width of the browser window, which is the nature of block elements in the normal document flow. Inline boxes encompass just the characters or image they contain.

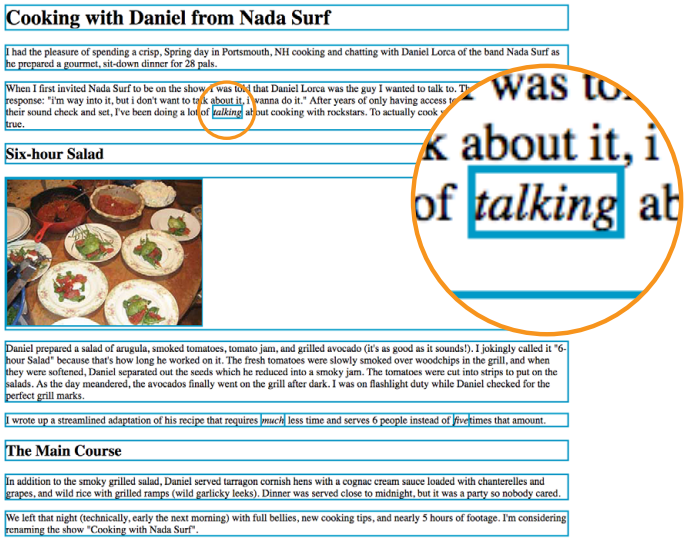


FIGURE 11-9. Rules around all the elements reveal their element boxes.

Grouped Selectors

Hey! This is a good opportunity to show you a handy style rule shortcut. If you ever need to apply the same style property to a number of elements, you can group the selectors into one rule by separating them with commas. This

one rule has the same effect as the five rules listed previously. Grouping them makes future edits more efficient and results in a smaller file size:

```
h1, h2, p, em, img { border: 1px solid blue; }
```

Now you have two selector types in your toolbox: a simple element selector and grouped selectors.

CSS UNITS OF MEASUREMENT

This chapter lays the groundwork for upcoming lessons, so it's a good time to get familiar with the units of measurement used in CSS. You'll be using them to set font size, the width and height of elements, margins, indents, and so on. The complete list is provided in the sidebar **"CSS Units."**

Some will look familiar (like inches and millimeters), but there are some units that bear more explanation: absolute units, rem, em, and vw/vh. Knowing how to use CSS units effectively is another one of those core CSS skills.

Pop Quiz

Can you guess why I didn't just add the **border** property to the **body** element and let it inherit to all the elements in the grouped selector?

Answer:

Because **border** is one of the properties that are not inherited.

CSS Units

CSS3 provides a variety of units of measurement. They fall into two broad categories: absolute and relative.

Absolute units

Absolute units have predefined meanings or real-world equivalents. With the exception of pixels, they are not appropriate for web pages that appear on screens.

- **px** pixel, defined as equal to 1/96 of an inch in CSS3.
- **in** inches.
- **mm** millimeters.
- **cm** centimeters.
- **q** ¼ millimeter.
- **pt** points (1/72 inch). Points are a unit commonly used in print design.
- **pc** picas (1 pica = 12 points or 1/6 inch). Points are a unit commonly used in print design.

Relative units

Relative units are based on the size of something else, such as the default text size or the size of the parent element.

- **em** a unit of measurement equal to the current font size.
- **ex** x-height, approximately the height of a lowercase "x" in the font.
- **rem** root em, equal to the em size of the root element (**html**).
- **ch** zero width, equal to the width of a zero (0) in the current font and size.
- **vw** viewport width unit, equal to 1/100 of the current viewport (browser window) width.
- **vh** viewport height unit, equal to 1/100 of the current viewport height.
- **vmin** viewport minimum unit, equal to the value of **vw** or **vh**, whichever is smaller.
- **vmax** viewport maximum unit, equal to the value of **vw** or **vh**, whichever is larger.

NOTES

- Although not a "unit," percentages are another common measurement value for web page elements. Percentages are calculated relative to another value, such as the value of a property applied to the current element or its parent or ancestor. The spec always says what a percentage value for a property is calculated on.

  When used for page layouts, percentage values ensure that page elements stay proportional.
- Child elements do not inherit the relative values of their parent, but rather the resulting *calculated* value.
- IE9 supports **vm** instead of **vmin**. IE and Edge (all versions as of 2017) do not support **vmax**.

Absolute Units

Absolute units have predefined meanings or real-world equivalents. They are always the same size, regardless of the context in which they appear.

The most popular absolute unit for web design is the pixel, which CSS3 defines as 1/96 inch. Pixels are right at home on a pixel-based screen and offer precise control over the size of the text and elements on the page. For a while there, pixels were all we used. Then we realized they are too rigid for pages that need to adapt to a wide variety of screen sizes and user preferences. Relative measurements like rem, em, and % are more appropriate to the fluid nature of the medium.

As long as we are kicking `px` to the curb, all of the absolute units—such as `pt`, `pc`, `in`, `mm`, and `cm`—are out because they are irrelevant on screens, although they may be useful for print style sheets. That narrows down your unit choices a bit.

That said, pixels do still have their place in web design for elements that truly should stay the same size regardless of context. Border widths are appropriate in pixels, as are images that have inherent pixel dimensions.

Relative Units

As I just established, relative units are the way to go for most web measurements, and there are a few options: rem, em, and `vw`/`vh`.

Rem Fallbacks for Old IE Browsers

The drawback to rems is that IE8 and earlier do not support them at all, and you need to provide a fallback declaration with the equivalent measurement in pixels. There are production tools that can convert all your rem units to pixels automatically, which are discussed in **Chapter 20, Modern Development Tools**.

The rem unit

CSS3 introduced a relative measurement called a rem (for root em) that is based on the font size of the root (`html`) element, whatever that happens to be. In modern browsers, the default root font size is 16 pixels; therefore, a rem is equivalent to a 16-pixel unit (unless you set it explicitly to another value). An element sized to 10rem would measure 160 pixels.

For the most part, you can use rem units like an absolute measurement in style rules; however, because it is relative, if the base font size changes, so does the size of a rem. If a user changes the base font size to 24 pixels for easier reading from a distance, or if the page is displayed on a device that has a default font size of 24 pixels, that 10rem element becomes 240 pixels. That seems dodgy, but rest assured that it is a feature, not a bug. There are many instances in which you want a layout element to expand should the text size increase. It keeps the page proportional with the font size, which can help maintain optimum line lengths.

The em unit

An em is a relative unit of measurement that, in traditional typography, is based on the width of the capital letter M (thus the name "em"). In the CSS

specification, an em is calculated as the distance between baselines when the font is set without any extra space between the lines (also known as leading). For text with a font size of 16 pixels, an em measures 16 pixels; for 12-pixel text, an em equals 12 pixels; and so on, as shown in FIGURE 11-10.

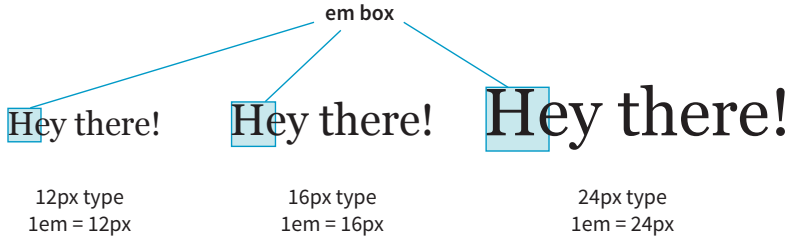


FIGURE 11-10. An em is based on the size of the text.

Once the dimension of an em for a text element is calculated by the browser, it can be used for all sorts of other measurements, such as indents, margins, the width of the element on the page, and so on. Basing measurements on text size helps keep everything in proportion should the text be resized.

The trick to working with ems is to remember they are always relevant to the current font size of the element. To borrow an example from Eric Meyer and Estelle Weyl's *CSS: The Definitive Guide* (O'Reilly), if you set a 2em left margin on an `h1`, `h2`, and `p`, those elements will not line up nicely because the em units are based on their respective element's sizes (FIGURE 11-11).

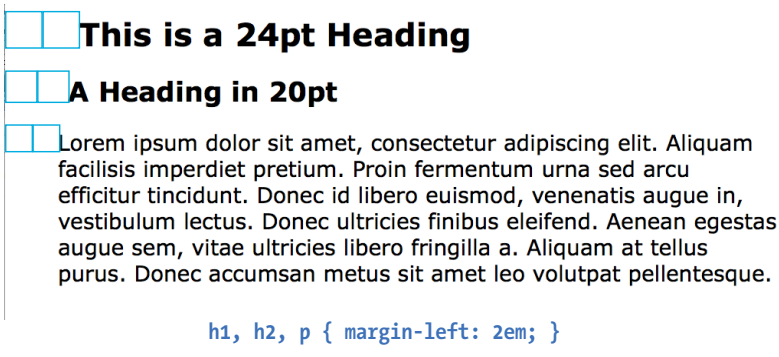


FIGURE 11-11. Em measurements are always relevant to the element's font size. An em for one element may not be the same for another.

Viewport percentage lengths (vw/vh)

The viewport width (`vw`) and viewport height (`vh`) units are relative to the size of the viewport (browser window). A `vw` is equal to 1/100 the width of the viewport. Similarly, a `vh` is equal to 1/100 the height of the viewport. Viewport-based units are useful for making images and text elements stay the full width or height of the viewport:

NOTE

Don't confuse the em unit of measurement with the `em` HTML element used to indicate emphasized text. They are totally different things.

```
header {
  width: 100vw;
  height: 100vh; }
```

It's also easy to specify a unit to be a specific percentage of the window size, such as 50%:

```
img {
  width: 50vw;
  height: 50vh; }
```

BROWSER SUPPORT NOTE

IE9 supports **vm** *instead of* **vmin**. *IE and Edge (all versions as of 2017) do not support* **vmax**.

Related are the **vmin** unit (equal to the value of **vw** or **vh**, whichever is smaller) and **vmax** (equal to the value of **vw** or **vh**, whichever is larger).

That should give you a good introduction to the units you'll be using in your style sheets. I recommend reading the full CSS Values and Units Module (*www.w3.org/TR/css3-values/*) to deepen your knowledge and make the values listed for properties in this book easier to understand. In addition to length units, it includes text-based values (such as keywords, text strings, and URLs), numbers and percentage values, colors, and more.

DEVELOPER TOOLS RIGHT IN YOUR BROWSER

Because of the cascade, a single page element may have styles applied from a number of sources. This can make it tricky to debug a page when styles aren't displaying the way you think they should. Fortunately, every major browser comes with developer tools that can help you sort things out.

I've opened the simple *cooking.html* document that we've been working on in the Chrome browser, then selected View → Developer → Developer Tools from the menu. The Developer Tools panel opens at the bottom of the document, as you can see in FIGURE 11-12. You can also make it its own separate window by clicking the windows icon in the top left.

In the Elements tab on the left, I can see the HTML source for the document. The content is initially hidden so you can see the structure of the document more clearly, but clicking the arrows opens each section. When I click the element in the source (like the second **p** element shown in the figure), that element is also highlighted in the browser window view.

In the Styles tab on the right, I can see all of the styles that are being applied to the selected element. In the example, I see the **font-size**, **font-family**, and **margin-left** properties from the **style** element in the document. If there were external CSS documents, they'd be listed too. I can also see the "User Agent Style Sheet," which is the browser's default styles. In this case, the browser style sheet adds the margin space around the paragraph. Chrome also provides a box model diagram for the selected element that shows the content dimensions, padding, border, and margins that are applied. This is a great tool for troubleshooting unexpected spacing in layouts.

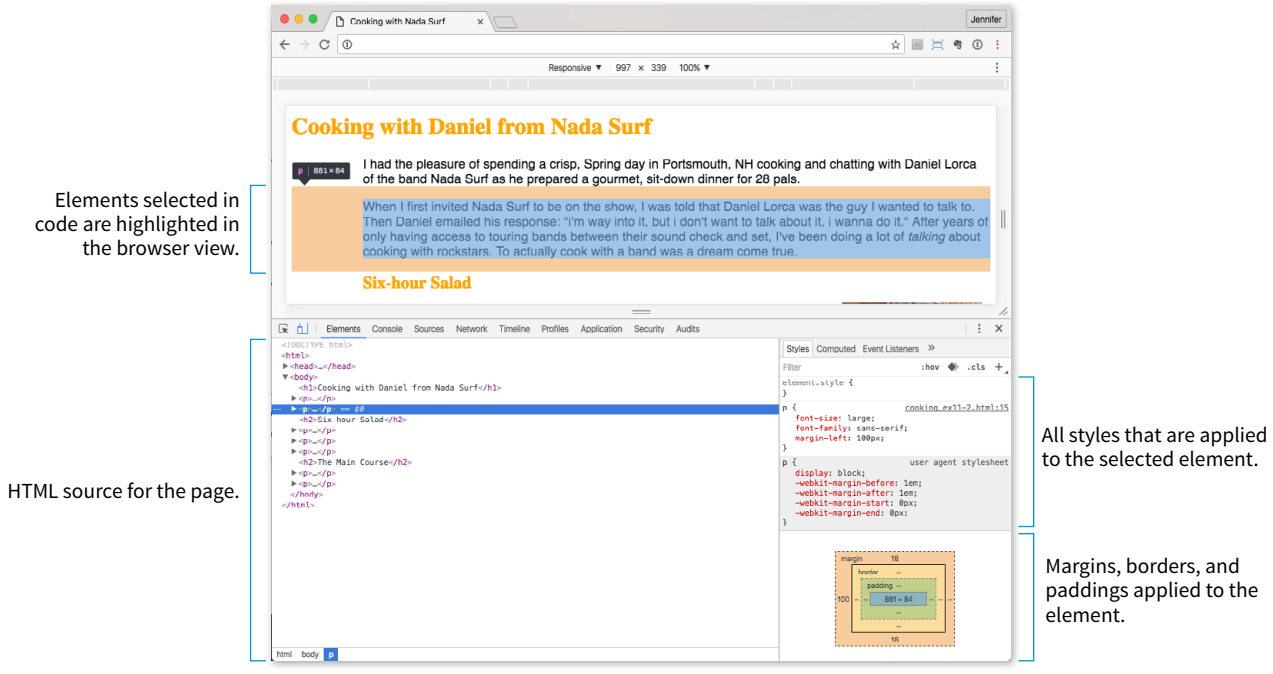


FIGURE 11-12. The Chrome browser with the Developer Tools panel open.

The *cool* thing is that when you edit the style rules in the panel, the changes are reflected in the browser view of the page in real time! If I select the **h1** element and change the color from orange to green, it turns green in the window. It's a great way to experiment with or troubleshoot a design; however, the changes are not being made to the document itself. It's just a preview, so you'll have to duplicate the changes in your source.

You can inspect *any* page on the web in this way, play around with turning styles off and on, and even add some of your own. Nothing you do has any effect on the actual site, so it is just for your education and amusement.

The element and style inspectors are just the tip of the iceberg of what browser developer tools can do. You can also tweak and debug JavaScript, check performance, view the document in various device simulations, and much more. The good news is that all major browsers now have built-in tools with similar features. As a web developer, you'll find they are your best friend.

- Chrome DevTools (View → Developer → Developer Tools)
  developer.chrome.com/devtools
- Firefox (Tools → Web Developer)
  developer.mozilla.org/en-US/docs/Tools
- Microsoft Edge (open with F12 key)
  developer.microsoft.com/en-us/microsoft-edge/platform/documentation/f12-devtools-guide/

- Safari (Develop → Show Web Inspector)
  developer.apple.com/library/content/documentation/AppleApplications/Conceptual/Safari_Developer_Guide/Introduction/Introduction.html)
- Opera (View → Developer Tools → Opera Dragonfly)
  www.opera.com/dragonfly/
- Internet Explorer 9+ (open with F12 key)
  msdn.microsoft.com/en-us/library/gg589512(v=vs.85).aspx

MOVING FORWARD WITH CSS

This chapter covered all the fundamentals of Cascading Style Sheets, including rule syntax, ways to apply styles to a document, and the central concepts of inheritance, the cascade (including priority, specificity, and rule order), and the box model. Style sheets should no longer be a mystery, and from this point on, we'll merely be building on this foundation by adding properties and selectors to your arsenal and expanding on the concepts introduced here.

CSS is a vast topic, well beyond the scope of this book. Bookstores and the web are loaded with information about style sheets for all skill levels. I've compiled a list of the resources I've found the most useful during my learning process. I've also provided a list of popular tools that assist in writing style sheets.

Books

There is no shortage of good books on CSS out there, but these are the ones that taught me, and I feel good recommending them.

- *CSS: The Definitive Guide, 4th Edition* by Eric A. Meyer and Estelle Weyl (O'Reilly)
- *CSS Cookbook* by Christopher Schmitt (O'Reilly)

Online Resources

The sites listed here are good starting points for online exploration of style sheets.

CSS-Tricks *(css-tricks.com)*

The is the blog of CSS guru Chris Coyier. Chris *loves* CSS and enthusiastically shares his research and tinkering on his site.

World Wide Web Consortium *(www.w3.org/TR/CSS/)*

The World Wide Web Consortium oversees the development of web technologies, including CSS. This page is a "snapshot" of the CSS specifications. See also *www.w3.org/Style/CSS/current-work*.

MDN Web Docs *(developer.mozilla.org)*

The CSS pages at MDN include detailed reference pages, step-by-step tutorials, and demos. It's a great hub for researching any web technology.

A List Apart *(www.alistapart.com/topics/code/css/)*

This online magazine features some of the best thinking and writing on cutting-edge, standards-based web design. It was founded in 1998 by Jeffrey Zeldman and Brian Platz.

TEST YOURSELF

Here are a few questions to test your knowledge of the CSS basics. Answers are provided in **Appendix A**.

1. Identify the various parts of this style rule:

```
blockquote { line-height: 1.5; }
```

selector: _____ value: _____

property: _____ declaration: _____

2. What color will paragraphs be when this embedded style sheet is applied to a document? Why?

```
<style type="text/css">
   p { color: purple; }
   p { color: green; }
   p { color: gray; }
</style>
```

3. Rewrite each of these CSS examples. Some of them are completely incorrect, and some could just be written more efficiently.

a.
```
p {font-family: sans-serif;}
p {font-size: 1em;}
p {line-height: 1.2em;}
```

b.
```
blockquote {
font-size: 1em
line-height: 150%
color: gray }
```

c.
```
body
{background-color: black;}
{color: #666;}
{margin-left: 12em;}
{margin-right: 12em;}
```

d.
```
p {color: white;}
blockquote {color: white;}
li {color: white;}
```

e.
```
<strong style="red">Act now!</strong>
```

4. Circle all the elements that you would expect to appear in red when the following style rule is applied to a document with the structure diagrammed in FIGURE 11-13.

```
div#intro { color: red;}
```

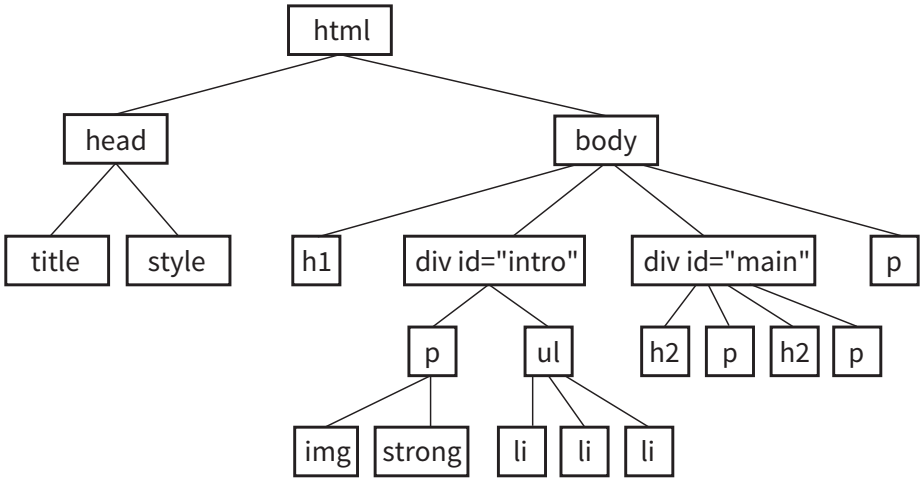


FIGURE 11-13. The document structure of a sample document.

CHAPTER 12

FORMATTING TEXT

Now that you've gotten your feet wet formatting text, are you ready to jump into the deep end? By the end of this chapter, you'll pick up over 40 additional CSS properties used to manipulate the appearance of text. Along the way, you'll also learn how to use more powerful selectors for targeting elements in a particular context and with a specific `id` or `class` name.

The nature of the web makes specifying type tricky, if not downright frustrating, particularly if you have experience designing for print or even formatting text in a word processing program. There is no way to know for sure whether the font you specify will be available or how large or small the type will appear when it hits your users' browsers. We'll address the best practices for dealing with these challenges as we go along.

Throughout this chapter, we'll be sprucing up a Black Goose Bistro online menu similar to the one we marked up back in **Chapter 5, Marking Up Text.** I encourage you to work along with the exercises to get a feel for how the properties work. FIGURE 12-1 shows how the menu looks before and after we're done. It's not a masterpiece, because we're just scratching the surface of CSS here, but at least the text has more personality.

IN THIS CHAPTER

BASIC FONT PROPERTIES

When I design a text document (for print or the web), one of the first things I do is specify a font. In CSS, fonts are specified using a set of font-related properties for typeface, size, weight, font style, and special characters. There are also shortcut properties that let you specify multiple font attributes in a single rule.

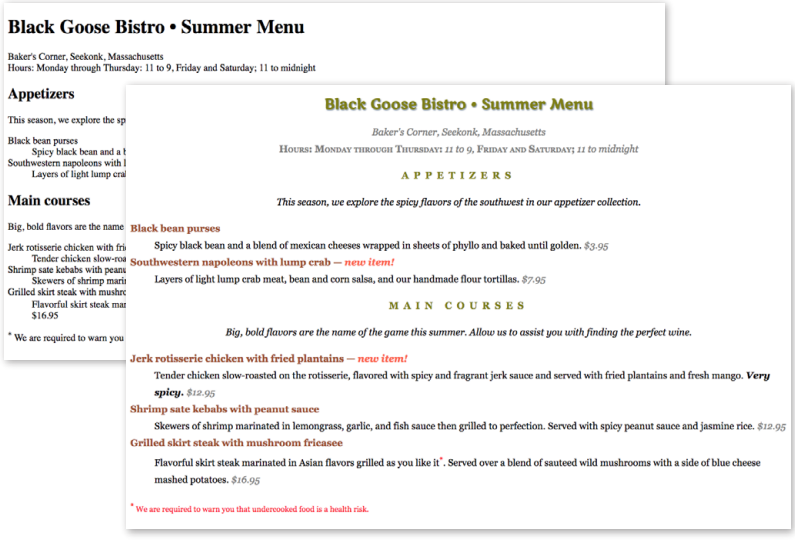


FIGURE 12-1. Before and after views of the Black Goose Bistro menu that we'll be working on in this chapter.

A Word About Property Listings

Each CSS property listing in this book is accompanied by information on how it behaves and how to use it. Property listings include:

Values:
These are the accepted values for the property. Predefined keyword values appear in code font (for example, **`small`**, **`italic`**, or **`small-caps`**) and must be typed in exactly as shown.

Default:
This is the value that will be used for the property by default (its initial value)—that is, if no other value is specified. Note that the default browser style sheet values may vary from the defaults defined in CSS.

Applies to:
Some properties apply only to certain types of elements.

Inherits:
This indicates whether the property is passed down to the element's descendants.

CSS-wide keywords

All CSS properties accept the three CSS-wide keywords: **`initial`**, **`inherit`**, and **`unset`**. Because they are shared by all properties, they are not listed with the values for individual property listings.

- The **`initial`** keyword explicitly sets the property to its default (initial) value.
- The **`inherit`** keyword allows you to explicitly force an element to inherit a style property from its parent. This may come in handy to override other styles applied to that element and to guarantee that the element always matches its parent.
- Finally, **`unset`** erases declared values occurring earlier in the cascade, setting the property to either **`inherit`** or **`initial`**, depending on whether it inherits or not.

Specifying the Font Name

Choosing a typeface, or font family as it is called in CSS, for your text is a good place to start. Let's begin with the **font-family** property and its values.

font-family

Values: *one or more font or generic font family names, separated by commas*

Default: depends on the browser

Applies to: all elements

Inherits: yes

Use the **font-family** property to specify a font or list of fonts (known as a font stack) by name, as shown in these examples:

```
body { font-family: Arial; }
var { font-family: Courier, monospace; }
p { font-family: "Duru Sans", Verdana, sans-serif; }
```

Here are some important syntax requirements:

- All font names, with the exception of generic font families, must be capitalized. For example, use **Arial** instead of **arial**.
- Use commas to separate multiple font names, as shown in the second and third examples.
- Notice that font names that contain a character space (such as Duru Sans in the third example) must appear within quotation marks.

You might be asking, "Why specify more than one font?" That's a good question, and it brings us to one of the challenges of specifying fonts for the web.

Font limitations

Browsers are limited to displaying fonts they have access to. Traditionally, that meant the fonts that were already installed on the user's hard drive. In 2010, however, there was a boom in browser support for embedded web fonts using the CSS **@font-face** rule, so it became possible for designers to provide their own fonts. See the sidebar **"Say Hello to Web Fonts"** for more information.

But back to our **font-family** rule. Even when you specify that the font should be Futura in a style rule, if the browser can't find it (for example, if that font is not installed on the user's computer or the provided web font fails to load), the browser uses its default font instead.

Fortunately, CSS allows us to provide a list of back-up fonts (that font stack we saw earlier) should our first choice not be available. If the first specified font is not found, the browser tries the next one, and down through the list until it finds one that works. In the third **font-family** rule shown in the previous code example, if the browser does not find Duru Sans, it will use Verdana, and if Verdana is not available, it will substitute some other sans-serif font.

■ AT A GLANCE

Font Properties

The CSS2.1 font-related properties are universally supported:

```
font-family
font-size
font-weight
font-style
font-variant
font
```

The CSS Font Module Level 3 adds these properties for more sophisticated font handling, although browser support is inconsistent as of this writing:

```
font-stretch
font-variant-ligatures
font-variant-position
font-variant-caps
font-variant-numeric
font-variant-alternates
font-variant-east-asian
font-size-adjust
font-kerning
font-feature-settings
font-language-override
```

Say Hello to Web Fonts

The ability to provide your own font for use on a web page has been around since 1998, but it was never feasible because of browser inconsistencies. Fortunately, that story has changed, and now web fonts are a perfectly viable option. The web has never looked better!

There is a lot to say about web fonts, so this sidebar is merely an introduction, starting with the challenges.

Web font formats

There have been two main hurdles to including fonts with web pages. First, there is the problem that different browsers support different font formats. Most fonts come in OpenType (OTF) or TrueType (TTF) format, but older versions of Internet Explorer accept only its proprietary Embedded Open Type (EOT).

The good news is that there is a new standard for packaging fonts for delivery to web pages that all browser vendors, even IE, are implementing. The new format, WOFF/WOFF2 (for Web Open Font Format versions 1 and 2), is a container that packages font files for web delivery. Now that IE9 supports WOFF, one day it may be all we need. As of this writing, however, it is still a best practice to provide the same font in a number of different formats (more on that in just a moment).

The other issue with providing fonts on web pages is that the font companies, or foundries, are concerned (a polite way to say "freaked out") that their fonts will be sitting vulnerably on servers and available for download. Fonts cost a lot to create and are very valuable. Most come with licenses that cover very specific uses by a limited number of machines, and "free to download for whatever" is usually not included.

So, to link to a web font, you need to use the font legally and provide it in a way that all browsers support. There are two general approaches to providing fonts: host them yourself or use a web font service. Let's look at both options.

Host your own

In the "host your own" option, you find the font you want, put it on your server in all the required formats, and link it to your web page by using the CSS3 `@font-face` rule. It is worth noting that each font file corresponds to a single weight or variant of a typeface. So if you want to use regular, bold, and italic versions, you have to host three different font files and reference each in your CSS.

Step 1: Find a font. This can be a bit of a challenge because the End User License Agreement (EULA) for virtually all commercial fonts does not cover web usage. Be sure to purchase the additional web license if it is available. However, thanks to demand, some foundries are opening fonts up for web use, and there are a growing number of open source fonts that you can use for free. The service Fontspring (*fontspring.com*), by Ethan Dunham, is a great place to purchase fonts that have a web license that you can use on your site or your own computer. The site Font Squirrel (*fontsquirrel.com*), also by Ethan Dunham, is a great source for open source fonts that can be used for commercial purposes for free.

Step 2: Save it in multiple formats. As of this writing, providing multiple formats (EOT, WOFF, TTF, SVG) is a reality. The recommended source for the various formats is the font vendor where you purchased the font, as they will be the best quality and

approved under the EULA. If you have an open source font (one that is free from licensing restrictions) and you need alternative formats, there is a service that will take your font and make everything you need for you—the "@font-face Generator" from Font Squirrel (*www.fontsquirrel.com/fontface/generator*). Go to that page and upload your font, and it gives back the font in TTF, EOT, WOFF, WOFF2, and SVG, as well as the CSS code you need to make it work.

Step 3: Upload to the server. Developers typically keep their font files in the same directory as the CSS files, but that's just a matter of preference. If you download a package from Font Squirrel, be sure to keep the pieces together as you found them.

Step 4: Write the code. Link the font to your site by using the **`@font-face`** rule in your *.css* document. The "at-rule" gives the font a **`font-family`** name that you can then reference later in your style sheet. It also lists the locations of the font files in their various formats. This cross-browser code example was developed by Ethan Dunham (yep, him again!) to address a bug in IE. I recommend reading the full article at *blog.fontspring.com/2011/02/further-hardening-of-the-bulletproof-syntax/*. See also Paul Irish's updated version at *paulirish.com/2009/bulletproof-font-face-implementation-syntax/*.

```
@font-face {
  font-family: 'MyWebFont';
  src: url('webfont.eot'); /* IE9 Compat Modes */
  src: url('webfont.eot?#iefix') format('embedded-
opentype'), /* IE6-IE8 */
  url('webfont.woff') format('woff'),
    /* Modern Browsers */
  url('webfont.ttf')  format('truetype'),
    /* Safari, Android, iOS */
  url('webfont.svg#svgFontName') format('svg');
    /* Legacy iOS */
  }
```

Then you just refer to the established font name in your font rules, like so:

```
p {font-family: MyWebFont; }
```

Use a font embedding service

If that seems like a lot of work, you may want to sign up with one of the font embedding services that do all the heavy lifting for you. For a fee, you get access to high-quality fonts, and the service handles font licensing and font protection for the foundries. They also generally provide an interface and tools that make embedding a font as easy as copy and paste.

The services have a variety of fee structures. Some charge monthly fees; some charge by the font. Some have a surcharge for bandwidth as well. There are generally tiered plans that range from free to hundreds of dollars per month.

Here are some font embedding services that are popular as of this writing, but it's worth doing a web search to see what's currently offered.

Google Web Fonts (www.google.com/webfonts)

Google Web Fonts is a free service that provides access to hundreds of open source fonts that are free for commercial use. All you have to do is choose a font, and then copy and paste the code they generate for you. If you don't have a font budget and you aren't too particular about fonts, this is a wonderful way to go. We'll use it in the first exercise in this chapter.

Typekit, from Adobe (www.typekit.com)

Typekit was the first web font service and is now part of Adobe. Their service uses JavaScript to link the fonts to your site in a way that improves performance and quality in all browsers. I also recommend their blog for excellent articles on how type works (see *blog.typekit.com/category/type-rendering/*).

Fonts.com (fonts.com)

Fonts.com boasts the largest font collection from the biggest font foundries. If you need a particular font, they are likely to have it.

Other services include Cloud Typography by Hoefler & Co. (*www.typography.com/cloud/welcome/*), Typotheque (*www.typotheque.com/webfonts*), and Fonts Live (*www.fontslive.com*). They differ in the number of fonts they offer and their fee structures, so you may want to shop around. Fontstand (*fontstand.com/*) allows you to rent fonts on a monthly basis, which, depending on your use, could work out to be a fraction of the cost of buying the font outright.

Summing up web fonts

Which method you use to add fonts to your site is up to your discretion. If you like total control, hosting your own font (legally, of course) may be a good way to go. If you need a very particular, well-known font because your client's brand depends on it, you will probably find it on one of the web font services for a price. If you want to experiment with web fonts and are happy to choose from what's freely available, then Google Web Fonts is for you.

You now have a good foundation in including web fonts on your web pages. The landscape is likely to change quickly over the next few years, so be sure to do your own research when you are ready to get started.

NOTE

Generic font family names do not need to be capitalized in the style rule.

Generic font families

That last option, "some other sans-serif font," bears more discussion. "Sans-serif" is just one of five generic font families that you can specify with the `font-family` property. When you specify a generic font family, the browser chooses an available font from that stylistic category. FIGURE 12-2 shows examples from each family.

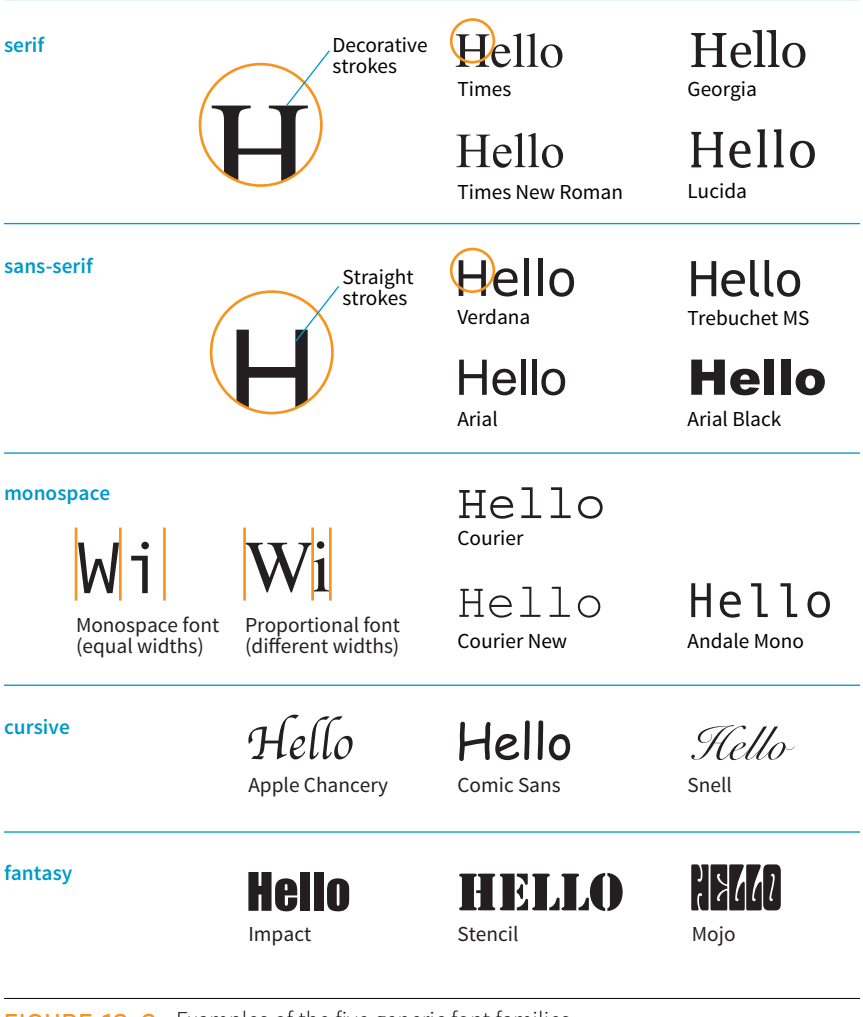


FIGURE 12-2. Examples of the five generic font families.

`serif`

Examples: Times, Times New Roman, Georgia

Serif typefaces have decorative slab-like appendages (serifs) on the ends of certain letter strokes.

`sans-serif`

Examples: Arial, Arial Black, Verdana, Trebuchet MS, Helvetica, Geneva

Sans-serif typefaces have straight letter strokes that do not end in serifs.

`monospace`

Examples: Courier, Courier New, and Andale Mono

In monospace (also called constant width) typefaces, all characters take up the same amount of space on a line. For example, a capital W will be no wider than a lowercase i. Compare this to proportional typefaces (such as the one you're reading now) that allot different widths to different characters.

`cursive`

Examples: Apple Chancery, Zapf-Chancery, and Comic Sans

Cursive fonts emulate a script or handwritten appearance.

`fantasy`

Examples: Impact, Western, or other decorative font

Fantasy fonts are purely decorative and would be appropriate for headlines and other display type.

Font stack strategies

The best practice for specifying fonts for web pages is to start with your first choice, provide some similar alternatives, and then end with a generic font family that at least gets users in the right stylistic ballpark. For example, if you want an upright, sans-serif font, you might start with a web font if you are providing one (Oswald), list a few that are more common (Univers, Tahoma, Geneva), and finish with the generic sans-serif. There is no limit to the number of fonts you can include, but many designers strive to keep it under 10.

```
font-family: Oswald, Univers, Tahoma, Geneva, sans-serif;
```

A good font stack should include stylistically related fonts that are known to be installed on most computers. Sticking with fonts that come with the Windows, macOS, and Linux operating systems, as well as fonts that get installed with popular software packages such as Microsoft Office and Adobe Creative Suite, gives you a solid list of "web-safe" fonts to choose from. A good place to look for stylistically related web-safe fonts is CSS Font Stack (*www.cssfontstack.com*). There are many articles on font stack strategies that are just a Google search away. I recommend Michael Tuck's "8 Definitive Font Stacks" (*www.sitepoint.com/eight-definitive-font-stacks*), which is an oldie but goodie.

So, as you see, specifying fonts for the web is more like merely suggesting them. You don't have absolute control over which font your users will see. You might get your first choice; you might get the generic fallback. It's one of those web design quirks you learn to live with.

Now seems like a good time to get started formatting the Black Goose Bistro menu. We'll add new style rules one at a time as we learn new properties, staring with EXERCISE 12-1.

EXERCISE 12-1. Formatting a menu

In this exercise, we'll change the fonts for the body and main heading of the Black Goose Bistro menu document, *menu.html*, which is available at *learningwebdesign.com/5e/materials*. Open the document in a text editor. You can also open it in a browser to see its "before" state. It should look something like FIGURE 12-1. Hang on to this document, because this exercise will continue as we pick up additional font properties.

I've included an embedded font in this exercise to show you how easy it is to do with a service like Google Web Fonts.

1. Use an embedded style sheet for this exercise. Start by adding a **style** element in the **head** of the document, like this:

```
<head>
  <title>Black Goose Bistro</title>
  <style>

  </style>
</head>
```

2. I would like the main text to appear in Verdana or some other sans-serif font. Instead of writing a rule for every element in the document, we will write one rule for the **body** element that will be inherited by all the elements it contains. Add this rule to the embedded style sheet:

```
<style>
  body {font-family: Verdana, sans-serif;}
</style>
```

3. I want a fancy font for the "Black Goose Bistro, Summer Menu" headline, so I chose a free display font called Marko One from Google Web Fonts (*www.google.com/webfonts*). Google gave me the code for linking the font file on their server to my HTML file (it's actually a link to an external style sheet). It must be placed in the **head** of the document, so copy it exactly as it appears, but keep it on one line. Put it after the **title** and before the **style** element.

```
<head>
<title>Black Goose Bistro</title>
<link href="http://fonts.googleapis.com/→
css?family=Marko+One" rel="stylesheet">
<style>
…
```

4. Now write a rule that applies it to the **h1** element. Notice I've specified Georgia or another serif font as fallbacks:

```
<style>
  body {font-family: Verdana, sans-serif;}
  h1 {font-family: "Marko One", Georgia, serif;}
</style>
```

5. Save the document and reload the page in the browser. It should look like FIGURE 12-3. Note that you'll need to have an internet connection and a current browser to view the Marko One headline font. We'll work on the text size in the next exercise.

Black Goose Bistro • Summer Menu

Baker's Corner, Seekonk, Massachusetts
Hours: Monday through Thursday: 11 to 9, Friday and Saturday; 11 to midnight

Appetizers

This season, we explore the spicy flavors of the southwest in our appetizer collection.

Black bean purses
Spicy black bean and a blend of mexican cheeses wrapped in sheets of phyllo and baked until golden. $3.95
Southwestern napoleons with lump crab — **new item!**
Layers of light lump crab meat, bean and corn salsa, and our handmade flour tortillas. $7.95

Main courses

Big, bold flavors are the name of the game this summer. Allow us to assist you with finding the perfect wine.

Jerk rotisserie chicken with fried plantains — **new item!**
Tender chicken slow-roasted on the rotisserie, flavored with spicy and fragrant jerk sauce and served with fried plantains and fresh mango. **Very spicy.** $12.95
Shrimp sate kebabs with peanut sauce
Skewers of shrimp marinated in lemongrass, garlic, and fish sauce then grilled to perfection. Served with spicy peanut sauce and jasmine rice. $12.95
Grilled skirt steak with mushroom fricasee
Flavorful skirt steak marinated in asian flavors grilled as you like it*. Served over a blend of sauteed wild mushrooms with a side of blue cheese mashed potatoes. $16.95

* We are required to warn you that undercooked food is a health risk.

FIGURE 12-3. The menu after we change only the font family.

Specifying Font Size

Use the aptly named **font-size** property to specify the size of the text.

font-size

Values: *length unit* | *percentage* | xx-small | x-small | small | medium | large | x-large | xx-large | smaller | larger

Default: medium

Applies to: all elements

Inherits: yes

You can specify text size in several ways:

- Using one of the CSS length units, as shown here:

  ```
  h1 { font-size: 1.5em; }
  ```

  When specifying a number of units, be sure the unit abbreviation immediately follows the number, with no extra character space in between (see the sidebar **"Providing Measurement Values"**).

  CSS length units are discussed in **Chapter 11, Introducing Cascading Style Sheets**. See also the **"CSS Units Cheat Sheet"** sidebar.

- As a percentage value, sized up or down from the element's inherited font size:

  ```
  h1 { font-size: 150%; }
  ```

- Using one of the absolute keywords (**xx-small**, **x-small**, **small**, **medium**, **large**, **x-large**, **xx-large**). On most current browsers, **medium** corresponds to the default font size.

  ```
  h1 { font-size: x-large; }
  ```

- Using a relative keyword (**larger** or **smaller**) to nudge the text larger or smaller than the surrounding text:

  ```
  strong { font-size: larger; }
  ```

I'm going to cut to the chase and tell you that, despite all these options, the preferred values for **font-size** in contemporary web design are the relative length units **em** and **rem**, as well as percentage values. You can specify font size in pixels (**px**), but in general, they do not provide the flexibility required in web page design. All of the other absolute units (**pt**, **pc**, **in**, etc.) are out too, unless you are creating a style sheet specifically for print.

I'll explain the keyword-based **font-size** values in a moment, but let's start our discussion with the best practice using relative values.

Providing Measurement Values

When you're providing measurement values, the unit must immediately follow the number, like this:

```
margin: 2em;
```

Adding a space before the unit will cause the property not to work:

```
INCORRECT: margin: 2 em;
```

It is acceptable to omit the unit of measurement for zero values:

```
margin: 0;
```

AT A GLANCE

CSS Units Cheat Sheet

As a quick reference, here are the CSS length units again:

Relative units

em	ex	rem	ch
vw	vh	vmin	vmax

Absolute units

px	in	mm	cm
q	pt	pc	

The preferred font-size values are em, rem, and %.

Sizing text with relative values

The best practice for setting the font size of web page elements is to do it in a way that respects the user's preference. Relative sizing values %, rem, and em allow you to use the default font size as the basis for proportional sizing of other text elements. It's usually not important that the headlines are exactly 24 pixels; it is important that they are 1.5 times larger than the main text so they stand out. If the user changes their preferences to make their default font size larger, the headlines appear larger, too.

To maintain the browser's default size, set the **font-size** of the root element to 100% (see **Note**):

```
html {
  font-size: 100%;
}
```

That sets the basis for relative sizing. Because the default font size for all modern browsers is 16 pixels, we'll assume our base size is 16 pixels going forward (we'll also keep in mind that it could be different).

NOTE

It is also common practice to set the **body** *to 100%, but setting it on the* **html** *element is a more flexible approach.*

Rem values

The rem unit, which stands for "root em," is always relative to the size of the root (**html**) element. If the root size is 16 pixels, then a rem equals 16 pixels. What's nice about rem units is, because they are always relative to the same element, they are the same size wherever you use them throughout the document. In that way, they work like an absolute unit. However, should the root size be something other than 16 pixels, elements specified in rem values will resize accordingly and proportionally. It's the best of both worlds.

BROWSER SUPPORT NOTE

Note that rem units are not supported in Internet Explorer 8 and earlier. If for some reason you need to support old browsers, you'll need to provide a fallback declaration set in pixels. There are also tools that change all your rem units to pixels automatically, as discussed in **Chapter 20, Modern Web Development Tools**.

Here is that same heading sized with rem values:

```
h1 { font-size: 1.5rem; }  /* 1.5 x 16 = 24 */
```

Em measurements

Em units are based on the font size of the current element. When you specify **font-size** in ems, it will be relative to the inherited size for that element. Once the em is calculated for an element, it can be used for other measurements as well, such as margins, padding, element widths, and any other setting you want to always be relative to the size of the font.

Here I've used em units to specify the size of an **h1** that has inherited the default 16-pixel font size from the root:

```
h1 { font-size: 1.5em; }  /* 1.5 x 16 = 24 */
```

There are a few snags to working with ems. One is that because of rounding errors, there is some inconsistency in how browsers and platforms render text set in ems.

The other tricky aspect to using ems is that they are based on the *inherited* size of the element, which means that their size is based on the context in which they are applied.

The **h1** in the previous example was based on an inherited size of 16 pixels. But if this **h1** had appeared in an **article** element that had its font size set to 14 pixels, it would inherit the 14-pixel size, and its resulting size would be just 21 pixels (1.5 × 14 = 21). FIGURE 12-4 shows the results.

THE MARKUP

```
<h1>Headline in Body</h1>
<p>Pellentesque ligula leo,…</p>
<article>
  <h1>Headline in Article</h1>
  <p>Vivamus …</p>
</article>
```

THE STYLES

```
h1 {
  font-size: 1.5em;    /* sets all h1s to 1.5em */
}
article {
  font-size: .875em    /* 14 pixels based on 16px default */
}
```

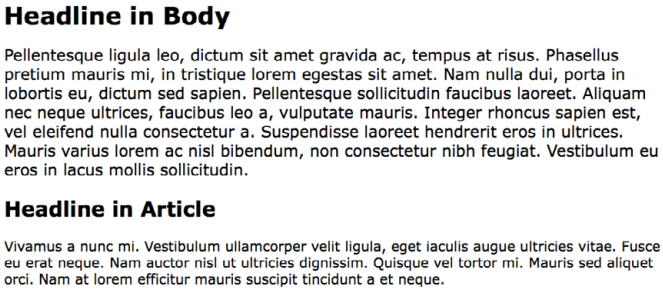


FIGURE 12-4. All **h1** elements are sized at 1.5em, but they are different sizes because of the context in which they appear.

From this example, you can see that an element set in ems might appear at different sizes in different parts of the document. If you wanted the **h1** in the article to be 24 pixels as well, you could calculate the em value by dividing the target size by its context: 24 / 14 = 1.71428571 em. (No need to round that figure down…the browser knows what to do with it.)

If you have elements nested several layers deep, the size increase or decrease compounds, which can create problems. With many layers of nesting, text may end up being way too small. When working with ems, pay close attention and write style rules in a way that takes the context into account.

This compounding nature of the em is what has driven the popularity of the predictable rem unit.

NOTE

Ethan Marcotte introduced the **target ÷ context = result** *formula in his book* ***Responsive Web Design*** *(A Book Apart). It is useful for converting pixel values into percentages and ems.*

Percentage values

To calculate % and em values, use this formula: target size ÷ size of context = result.

We saw a percentage value (100%) used to preserve the default font size, but you can use percentage values for any element. They are pretty straightforward.

In this example, the **h1** inherits the default 16px size from the **html** element, and applying the 150% value multiplies that *inherited* value, resulting in an **h1** that is 24 pixels:

```
h1 { font-size: 150%; }  /* 150% of 16 = 24 */
```

Working with keywords

An alternative way to specify **font-size** is by using one of the predefined absolute keywords: **xx-small**, **x-small**, **small**, **medium**, **large**, **x-large**, and **xx-large**. The keywords do not correspond to particular measurements, but rather are scaled consistently in relation to one another. The default size is **medium** in current browsers. FIGURE 12-5 shows how each of the absolute keywords renders in a browser when the default text is set at 16 pixels. I've included samples in Verdana and Times to show that, even with the same base size, there is a big difference in legibility at sizes **small** and below. Verdana was designed to be legible on screens at small font sizes; Times was designed for print so is less legible in that context.

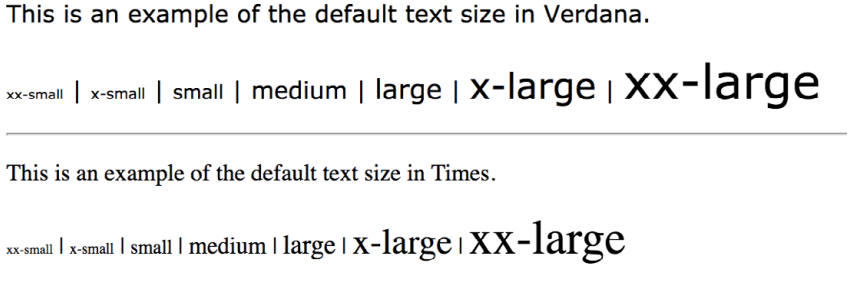


FIGURE 12-5. Text sized with absolute keywords.

The relative keywords, **larger** and **smaller**, are used to shift the size of text relative to the size of the parent element text. The exact amount of the size change is determined by each browser and is out of your control. Despite that limitation, it is an easy way to nudge type a bit larger or smaller if the exact proportions are not critical.

You can apply your new CSS font knowledge in EXERCISE 12-2.

EXERCISE 12-2. Setting font size

Let's refine the size of some of the text elements to give the online menu a more sophisticated appearance. Open *menu.html* in a text editor and follow the steps. You can save the document at any point and take a peek in the browser to see the results of your work. You should also feel free to try out other size values along the way.

1. There are many approaches to sizing text on web pages. In this example, start by putting a stake in the ground and setting the **font-size** of the **body** element to 100%, thus clearing the way for em measurements thereafter:

```
body {
  font-family: Verdana, sans-serif;
  font-size: 100%;
}
```

2. The browser default of 16 pixels is a fine size for the main page text, but I would like to improve the appearance of the heading levels. I'd like the main heading to be 24 pixels, or one and a half times larger than the body text [target (24) ÷ context (16) = 1.5]. I'll add a new rule that sets the size of the **h1** to 1.5em. I could have used 150% to achieve the same thing.

```
h1 {
  font-size: 1.5em;
}
```

3. Now make the **h2**s the same size as the body text so they blend in with the page better:

```
h2 {
  font-size: 1em;
}
```

FIGURE 12-6 shows the result of our font-sizing efforts.

FIGURE 12-6. The online menu after a few minor font-size changes to the headings.

Font Weight (Boldness)

After font families and size, the remaining font properties are straightforward. For example, if you want a text element to appear in bold, use the **font-weight** property to adjust the boldness of type.

font-weight

Values: normal | bold | bolder | lighter | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900

Default: normal

Applies to: all elements

Inherits: yes

As you can see, the **font-weight** property has many predefined values, including descriptive terms (**normal**, **bold**, **bolder**, and **lighter**) and nine numeric values (**100** to **900**) for targeting various weights of a font if they are available.

Because most fonts commonly used on the web have only two weights, normal (or Roman) and bold, the only font weight value you will use in most cases is **bold**. You may also use **normal** to make text that would otherwise appear in bold (such as strong text or headlines) appear at a normal weight.

The numeric chart may come in handy when using web fonts with a large range of weights (I've seen a few Google web fonts that require numeric size values). If multiple weights are not available, numeric settings of 600 and higher generally result in bold text, as shown in FIGURE 12-7 (although even that can vary by browser).

If a separate bold face is not available, the browser may "synthesize" a bold font by beefing up the available normal face (see **Note**).

NOTE

The CSS Fonts Module Level 3 introduced the **font-synthesis** *property, which allows authors to turn off (with a value of* **none***) or allow synthesized bold fonts (value of* **weight***); however, it is still considered experimental at this time.*

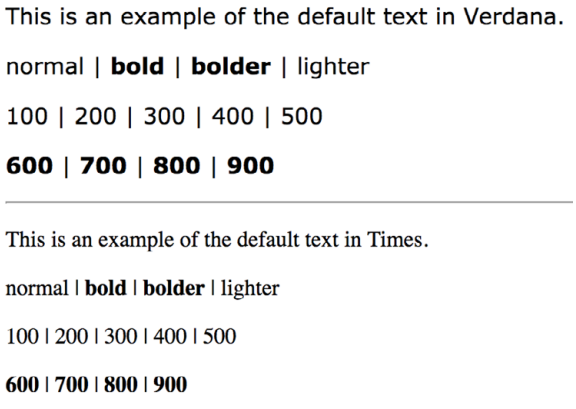


FIGURE 12-7. The effect (and lack thereof!) of **font-weight** values.

Font Style (Italics)

The **font-style** property affects the posture of the text—that is, whether the letter shapes are vertical (**normal**) or slanted (**italic** and **oblique**).

font-style

Values: normal | italic | oblique

Default: normal

Applies to: all elements

Inherits: yes

Use the **font-style** property to make text **italic**. Another common use is to make text that is italicized in the browser's default styles (such as emphasized text) display as **normal**. There is an **oblique** value that specifies a slanted version of the font; however, browsers generally display **oblique** exactly the same as **italic**.

Try out weight and style in EXERCISE 12-3.

EXERCISE 12-3. Making text bold and italic

Back to the menu. I've decided that I'd like all of the menu item names to be in bold text. What I'm not going to do is wrap each one in **<b>** tags…that would be so 1996! I'm also not going to mark them up as **strong** elements…that is not semantically accurate. Instead, the right thing to do is simply apply a style to the semantically correct **dt** (definition term) elements to make them all bold at once. Add this rule to the end of the style sheet, save the file, and try it out in the browser:

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

Now that all the menu item names are bold, some of the text I've marked as **strong** isn't standing out very well, so I think I'll make them italic for further emphasis. To do this, simply apply the **font-style** property to the **strong** element:

```
strong { font-style: italic;}
```

Once again, save and reload. It should look like the detail shown in FIGURE 12-8.

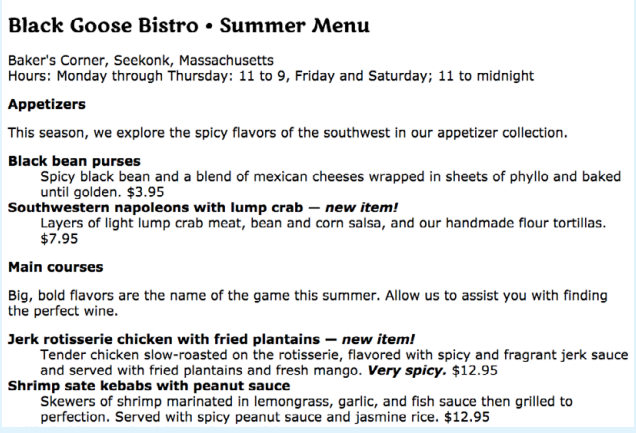


FIGURE 12-8. Applying the **font-weight** and **font-style** properties.

Font Variant in CSS2.1 (Small Caps)

font-variant

Values: normal | small-caps

Default: normal

Applies to: all elements

Inherits: yes

Some typefaces come in a "small caps" variant. This is a separate font design that uses small uppercase-style letters in place of lowercase letters. Small caps characters are designed to match the size and density of lowercase text so they blend in.

Small caps should be used for strings of three or more capital letters appearing in the flow of text, such as acronyms and abbreviations, that may look jarring as full-sized capitals. Compare NASA and USA in the standard font to NASA and USA in small caps. Small caps are also recommended for times, like 1AM or 2017AD.

When the **font-variant** property was introduced in CSS2.1, it was a one-trick pony that allowed designers to specify a small-caps font for text elements. CSS3 has greatly expanded the role of **font-variant**, as I will cover in the upcoming section **"Advanced Typography with CSS3."** For now, we'll look at only the CSS2.1 version of **font-variant**.

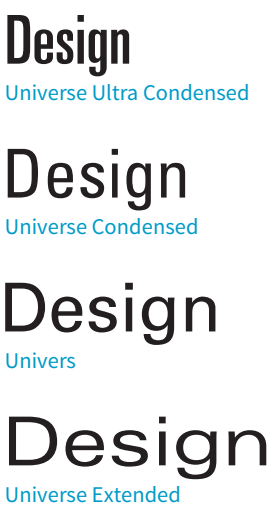


FIGURE 12-9. Examples of condensed, normal, and extended versions of the Universe typeface.

In most cases, browsers simulate small caps by scaling down uppercase letters in the current font. To typography sticklers, this is less than ideal and results in inconsistent stroke weights, but you may find it an acceptable option for adding variety to small amounts of text. You will see an example of small caps when we use the **font-variant** property in EXERCISE 12-5.

Font Stretch (Condensed and Extended)

font-stretch

Values:	normal | ultra-condensed | extra-condensed | condensed | semi-condensed | semi-expanded | expanded | extra-expanded | ultra-expanded
Default:	normal
Applies to:	all elements
Inherits:	yes

The CSS3 **font-stretch** property tells the browser to select a normal, condensed, or extended font in the font family (FIGURE 12-9). If the browser cannot find a matching font, it will *not* try to synthesize the width by stretching or squeezing text; it may just substitute a font of a different width. Browser support is just beginning to kick in for this property. As of this writing, it works on IE11+, Edge, Firefox, Chrome 48+, Opera, and Android 52+, but it is not yet supported on Safari or iOS Safari; however, that may change.

WARNING

Be careful when using shorthand properties like **font**. **Any omitted property resets to its default value.** *On the flip side, the shorthands are a good way to get a blank slate if you need one.*

The Shortcut font Property

Specifying multiple font properties for each text element can get repetitive and lengthy, so the creators of CSS provided the shorthand **font** property, which compiles all the font-related properties into one rule.

font

Values:	*font-style font-weight font-variant font-stretch font-size/line-height font-family* | caption | icon | menu | message-box | small-caption | status-bar
Default:	depends on default value for each property listed
Applies to:	all elements
Inherits:	yes

The value of the **font** property is a list of values for all the font properties we just looked at, separated by character spaces. It is important to note that only the CSS2.1 version of **font-variant** (**small-caps**) can be used in the **font** shortcut (which is one reason I kept it separate). In this property, the order of the values is important:

```
{ font: style weight stretch variant size/line-height font-family; }
```

CROSS-BROWSER SUPPORT TIP

If you include values for the newer **font-stretch** property in the **font** shorthand, first list a version that omits stretch for browsers that don't support it. You will end up with two declarations like this:

```
h3 {
  font: bold 1.25em Helvetica;
  font: bold extended 1.25em
Helvetica;
}
```

At minimum, the **font** property *must* include a **font-size** value and a **font-family** value, in that order. Omitting one or putting them in the wrong order causes the entire rule to be invalid. This is an example of a minimal font property value:

```
p { font: 1em sans-serif; }
```

Once you've met the size and family requirements, the other values are optional and may appear in any order *prior* to the **font-size**. When style, weight, stretch, or variant is omitted, its value is set to **normal**. That makes it easy to accidentally override a previous setting with the shorthand property, so be careful when you use it.

There is one value in there, **line-height**, that we have not seen yet. As it sounds, it adjusts the height of the text line and is used to add space between lines of text. It appears just after **font-size**, separated by a slash, as shown in these examples. The **line-height** property is covered in more detail later in this chapter.

```
h3 { font: oblique bold small-caps 1.5em/1.8em Verdana, sans-serif; }
h2 { font: bold 1.75em/2 sans-serif; }
```

In EXERCISE 12-4, we'll use the shorthand **font** property to make some changes to the **h1** headings in the bistro menu.

System font keywords

The **font** property also has a number of keyword values (**caption**, **icon**, **menu**, **message-box**, **small-caption**, and **status-bar**) that represent system fonts, the fonts used by operating systems for things like labels for icons and menu items. These may be useful when you're designing a web application so that it matches the environment the user is working on. These are considered shorthand values because they encapsulate the font, size, style, and weight of the font used for each purpose with only one keyword.

Like the shorthand **font** property, EXERCISE 12-4 is short and sweet.

ADVANCED TYPOGRAPHY WITH CSS3

Now you have a good basic toolkit for formatting fonts with CSS. If you want to get fancy, you should read up on all the properties in the CSS Fonts Module Level 3, which give you far more control over character selection and position. I'm going to keep my descriptions brief because of space restraints and the fact that many of these features are still experimental or have very limited browser support. But if nice typography is your thing, I urge you to do more research, starting with the specification at *www.w3.org/TR/css-fonts-3*.

EXERCISE 12-4.
Using the shorthand font property

One last tweak to the menu, and then we'll take a brief break. To save space, we can replace all the font properties we've specified for the **h1** element with one declaration with the shorthand **font** property:

```
h1 {
  font: bold 1.5em "Marko One",
Georgia, serif;
}
```

You might find it redundant that I included the bold font weight value in this rule. After all, the **h1** element was already bold by default, right? The thing about shorthand properties is that if you omit a value, it is reset to the default value for that *property*, not the browser's default value.

In this case, the default **font-weight** value within a **font** declaration is **normal**. Because our style sheet overrides the browser's default bold heading style, the **h1** would appear in normal-weight text if we don't explicitly make it bold in the **font** property. Shorthand properties can be tricky that way…pay attention so you don't leave something out and override a default or inherited value you were counting on.

You can save this and look at it in the browser. If you've done your job right, it should look exactly the same as in the previous step.

Font Variant in CSS3

The collection of **font-variant-** prefixed properties in CSS3 aims to give designers and developers access to special characters (glyphs) in fonts that can make the typography on a page more sophisticated.

As I mentioned earlier, the CSS3 Font Module greatly expanded the definition of **font-variant**. Now it can serve as a shorthand property for a number of **font-variant-** prefixed properties. These properties are still considered experimental, although browser support is starting to pick up. Still, it's interesting to see how font control in web design is evolving, so let's take a look.

NOTE

With the exception of **font-variant-position**, *which has a specific purpose, the other* **font-variant** *properties are great opportunities to practice progressive enhancement. They are nice to have but OK to lose.*

font-variant-ligatures

A ligature is a glyph that combines two or more characters into one symbol. One common example is the combination of a lowercase f and i, where the dot on the i becomes part of the f (fi). Ligatures can smooth out the appearance of known awkward letter pairings, and ligature glyphs are included in many fonts. The **font-variant-ligatures** property provides a way to control the use of ligatures on web pages. This one is better supported than the others, and already works in IE10+, Chrome 34+, as well as Safari and Opera (with the **-webkit-** prefix). I would expect browser support to steadily improve.

NOTE

The **font-variant-ligatures** *property has a long list of values, which you can find at www.w3.org/TR/css-fonts-3/#propdef-font-variant-ligatures.*

font-variant-caps

Allows the selection of small-cap glyphs (**small-caps**) from the font's character set rather than simulating them in the browser. The **all-small-caps** value uses small caps for upper- and lowercase letters. **unicase** uses small caps for uppercase only, and lowercase letters in the word stay the same. **titling-caps** is used for all-caps titles but is designed to be less strong. Other options are **petite-caps** and **all-petite-caps**.

font-variant-position

Selects superscript (**super**) or subscript (**sub**) glyphs from the font's character set when they are available. Otherwise, the browser creates superscript or subscript text for the **sup** and **sub** elements by shrinking the character and moving it above or below the baseline.

font-variant-numeric

Allows the selection of various number character styles if they are available. For example, you can pick numerals that are proportional or line up in columns as for a spreadsheet (**proportional-numbers**/**tabular-numbers**) opt for old-style numerals (**old-style-nums**) where some characters dip

below the baseline, and specify whether fractions should be on a diagonal or stacked (`diagonal-fractions`/`stacked-fractions`). It also allows you to make ordinal numbers look like 2nd instead of 2nd (`ordinal`) and gives you a way to use zeros with slashes through them as is preferred in some contexts (`slashed-zero`).

`font-variant-alternates`

Fonts sometimes offer more than one glyph for a particular character—for example, a few swash designs for the letter S, or an old-fashioned s that looks more like an f. `font-variant-alternates` provides a way to specify swashes and other alternative characters. Many of its values are font-specific and must be defined first with the `@font-features-values` at-rule. I'll leave a deeper explanation to the spec.

`font-variant-east-asian`

Allows selection of particular Asian glyphs.

Finally, the old `font-variant` property that has been around since the beginning of CSS has been upgraded to be a shorthand property for all of the properties listed here. You can use it today with the original `small-caps` value, and it will be perfectly valid. Once these properties gain traction, it will be able to do a whole lot more.

Other CSS3 Properties

It's time to finish up our review of the font properties in the Fonts Module Level 3. I'll give you a general idea of what is available (or will be, after browser support catches up) and you can dig deeper in the spec on your own:

`font-size-adjust`

The size text *looks* on the page often has more to do with the height of the lowercase x (its x-height) than the specified size of the text. For example, 10-point type with relatively large x-height is likely easier to read than 10-point type with dainty little lowercase letters. The `font-size-adjust` property allows the browser to adjust the size of a fallback font until its x-height matches the x-height of the first-choice font. This can ensure better legibility even when a fallback font needs to be used.

`font-kerning`

Kerning is the space between character glyphs. Fonts typically contain metadata about which letter pairs need to be cozied up together to make the spacing in a word look consistent. The `font-kerning` property allows the font's kerning information to be applied (`normal`), turned off (`none`), or left to the browser's discretion (`auto`).

`font-feature-settings`

This property gives authors the ability to control advanced typographic features in OpenType fonts that are not widely used, such as swashes, small

caps, ligatures, automatic fractions, and more. Those features should look familiar, as many of them can be controlled with various `font-variant` properties. In fact, the spec recommends you use `font-variant` whenever possible and reserve `font-feature-settings` for edge cases. As of this writing, however, the `font-feature-settings` property has better browser support, so for the time being it may be a better option. Just be aware that it cascades poorly, meaning it is easy to undo a setting when you use it later to set something else. CSS-Tricks provides a good overview by Robin Rendle (*css-tricks.com/almanac/properties/f/font-feature-settings*).

`font-language-override`

This experimental property controls the use of language-specific glyphs.

We've finally made our way through the various ways to control fonts in CSS (it took a while!), but that is just one aspect of text presentation. Changing the color of text is another common design choice.

> **FURTHER READING**
>
> For a nice overview of OpenType features and why they are worthwhile, read "Caring about OpenType Features" by Tim Brown at Adobe Typekit (*practice.typekit.com/lesson/caring-about-opentype-features/*).

CHANGING TEXT COLOR

You got a glimpse of how to change text color in **Chapter 11, Introducing Cascading Style Sheets**, and to be honest, there's not a lot more to say about it here. You change the color of text with the `color` property.

`color`

Values:	*color value (name or numeric)*
Default:	depends on the browser and user's preferences
Applies to:	all elements
Inherits:	yes

Using the `color` property is very straightforward. The value of the `color` property can be a predefined color name (see the **"Color Names"** sidebar) or a numeric value describing a specific RGB color. Here are a few examples, all of which make the `h1` elements in a document gray:

```
h1 { color: gray; }
h1 { color: #666666; }
h1 { color: #666; }
h1 { color: rgb(102,102,102); }
```

Don't worry about the numeric values for now; I just wanted you to see what they look like. RGB color is discussed in detail in **Chapter 13, Colors and Backgrounds,** so in this chapter, we'll just stick with color names for demonstration purposes.

Color is inherited, so you can change the color of all the text in a document by applying the **color** property to the **body** element, as shown here:

```
body { color: fuchsia; }
```

OK, so you probably wouldn't want all your text to be fuchsia, but you get the idea.

For the sake of accuracy, I want to point out that the **color** property is not strictly a text-related property. In fact, according to the CSS specification, it is used to change the foreground (as opposed to the background) color of an element. The foreground of an element consists of both the text it contains as well as its border. So, when you apply a color to an element (including image elements), know that color will be used for the border as well, unless there is a specific **border-color** property that overrides it. We'll talk more about borders and border color in **Chapter 14, Thinking Inside the Box.**

Before we add color to the online menu, I want to take a little side trip and introduce you to a few more types of selectors that will give us more flexibility in targeting elements in the document for styling.

■ AT A GLANCE

Color Names

CSS2.1 defines 17 standard color names:

black	white	purple
lime	navy	aqua
silver	maroon	fuchsia
olive	blue	orange
gray	red	green
yellow	teal	

The updated CSS Color Module Level 3 allows names from a larger set of 140 color names to be specified in style sheets. You can see samples of each in FIGURE 13-2 and at *learningwebdesign.com/colornames.html*.

A FEW MORE SELECTOR TYPES

So far, we've been using element names as selectors. In the last chapter, you saw how to group selectors together in a comma-separated list so you can apply properties to several elements at once. Here are examples of the selectors you already know:

Element selector `p { color: navy; }`

Grouped selectors `p, ul, td, th { color: navy; }`

The disadvantage of selecting elements this way, of course, is that the property (in this case, navy blue text) is applied to every paragraph and other listed elements in the document. Sometimes you want to apply a rule to a particular paragraph or paragraphs. In this section, we'll look at three selector types that allow us to do just that: descendant selectors, ID selectors, and class selectors.

Descendant Selectors

A descendant selector targets elements that are contained within (and therefore are descendants of) another element. It is an example of a contextual selector because it selects the element based on its context or relation to another element. The sidebar **"Other Contextual Selectors"** lists some more.

A character space between element names means that the second element must be contained within the first.

Descendant selectors are indicated in a list separated by a character space. This example targets emphasized text (**em**) elements, but *only* when they appear in list items (**li**). Emphasized text in paragraphs and other elements would be unaffected (FIGURE 12-10).

```
li em { color: olive; }
```

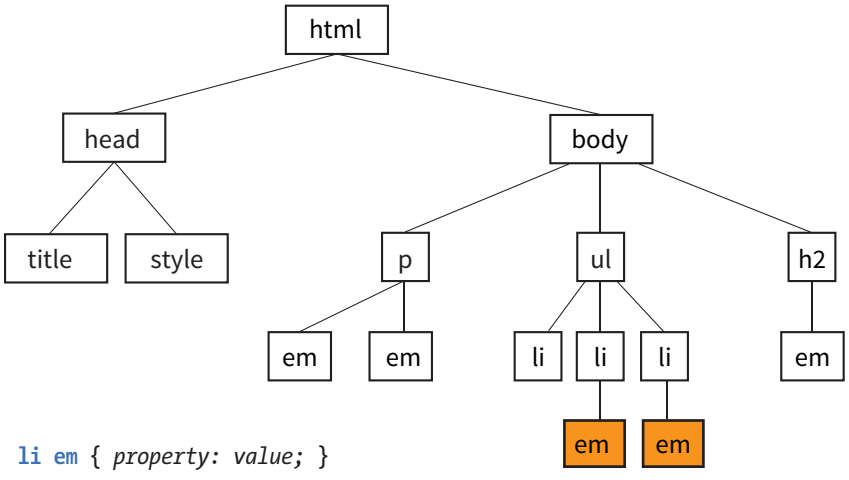


FIGURE 12-10. Only **em** elements within **li** elements are selected. The other **em** elements are unaffected.

Here's another example that shows how contextual selectors can be grouped in a comma-separated list, just as we saw earlier. This rule targets **em** elements, but only when they appear in **h1**, **h2**, and **h3** headings:

```
h1 em, h2 em, h3 em { color: red; }
```

It is also possible to nest descendant selectors several layers deep. This example targets **em** elements that appear in anchors (**a**) in ordered lists (**ol**):

```
ol a em { font-variant: small-caps; }
```

ID Selectors

The # symbol identifies an ID selector.

Back in **Chapter 5, Marking Up Text**, we learned about the **id** attribute, which gives an element a unique identifying name (its id reference). The **id** attribute can be used with any element, and it is commonly used to give meaning to the generic **div** and **span** elements. ID selectors allow you to target elements by their **id** values. The symbol that identifies ID selectors is the octothorpe (**#**), also known as a hash or pound symbol.

Here is an example of a list item with an **id** reference:

```
<li id="sleestak">Sleestak T-shirt</li>
```

Now you can write a style rule just for that list item using an ID selector, like so (notice the **#** preceding the **id** reference):

```
li#sleestak { color: olive; }
```

Because **id** values must be unique in the document, it is acceptable to omit the element name. The following rule is equivalent to the last one:

```
#sleestak { color: olive; }
```

You can also use an ID selector as part of a contextual selector. In this example, a style is applied only to **a** elements that appear within the element identified as "resources." In this way, you can treat links in the element named "resources" differently than all the other links on the page without any additional markup.

```
#resources a { text-decoration: none; }
```

You should be beginning to see the power of selectors and how they can be used strategically along with well-planned semantic markup.

Other Contextual Selectors

Descendant selectors are one of four types of contextual selectors (called combinators in the Selectors specifications Level 3 and Level 4). The other three are child selectors, next-sibling selectors, and subsequent-sibling selectors.

Child selector

A child selector is similar to a descendant selector, but it targets only the direct children of a given element. There may be no other hierarchical levels in between. They are indicated with the greater-than symbol (**>**). The following rule affects emphasized text, but only when it is directly contained in a **p** element. An **em** element inside a link (**a**) within the paragraph would not be affected.

```
p > em {font-weight: bold;}
```

Next-sibling selector

A next-sibling selector targets an element that comes directly after another element with the same parent. It is indicated with a plus (**+**) sign. This rule gives special treatment to paragraphs that follow an **h1**. Other paragraphs are unaffected.

```
h1 + p {font-style: italic;}
```

Subsequent-sibling selectors

A subsequent-sibling selector selects an element that shares a parent with the specified element and occurs after it in the source order. They do not need to follow one another directly. This type of selector is new in CSS3 and is not supported by Internet Explorer 8 and earlier. The following rule selects any **h2** that both shares a parent element (such as a **section** or **article**) with an **h1** and appears after it in the document.

```
h1 ~ h2 {font-weight: normal;}
```

Class Selectors

The period (.) symbol indicates a class selector.

One last selector type, and then we can get back to text style properties. The other element identifier you learned about in **Chapter 5** is the `class` identifier, used to classify elements into a conceptual group. Unlike the `id` attribute, multiple elements may share a `class` name. Not only that, but an element may belong to more than one class.

You can target elements belonging to the same class with—you guessed it—a class selector. Class names are indicated with a period (`.`) at the beginning of the selector. For example, to select all paragraphs with `class="special"`, use this selector (the period indicates the following word is a class selector):

```
p.special { color: orange; }
```

To apply a property to *all* elements of the same class, omit the element name in the selector (be sure to leave the period; it's the character that indicates a class). This example targets all paragraphs and any other element that has been marked up with `class="special"`:

```
.special { color: orange; }
```

Specificity 101

In **Chapter 11**, I introduced you to the term specificity, which refers to the fact that more specific selectors have more weight when it comes to handling style rule conflicts. Now that you know a few more selectors, it is a good time to revisit this very important concept.

This list of selector types from most to least specific should serve you well in most scenarios:

- **Inline styles** with the `style` attribute are more specific than (and will override...)
- **ID selectors**, which are more specific than (and will override...)
- **Class selectors**, which are more specific than (and will override...)
- **Individual element selectors**

The full story is a little more complicated, but here it is in a nutshell. To calculate specificity, start by drawing three boxes:

```
[ ]  [ ]  [ ]
```

Now count up the number of IDs in the selector, and put that number in the first box. Next count up the number of classes and pseudo-classes in the selector, and put that number in the second box. Third, count up the element names, and put that number in the third box.

Specificity is compared box by box. The first box that is not a tie determines which selector wins. Here is a simple example of two conflicting rules for the `h1` element:

```
h1 { color: red;}                [0] [0] [1]
h1.special { color: lime; }      [0] [1] [1]
```

The second one has a class selector and the first one doesn't; therefore, the second one is more specific and has more weight.

How about something more complicated?

```
article#main aside.sidebar:hover > h1:first-of-type  [1] [3] [3]
.x.x.x.x.x.x.x.x  a:link         [0] [8] [1]
```

The second selector targets a link in an element with a string of class names (represented by ".x"). But the first selector has an ID (**#main**) and is therefore more specific.

You may need to do this full specificity calculation, but in most cases you'll have a feel for which selector is more specific by following previously listed general guidelines.

You can use specificity strategically to keep your style sheets simple and your markup minimal. For example, it is possible to set a style for an element (**p**, in this example), and then override when necessary by using more specific selectors.

```
p { line-height: 1.2em; }              [0] [0] [1]
blockquote p { line-height: 1em; }     [0] [0] [2]
p.intro { line-height: 2em; }          [0] [1] [1]
```

In these examples, **p** elements that appear within a **blockquote** have a smaller line height than ordinary paragraphs. However, all paragraphs with a **class** of "intro" will have a 2em line height, even if it appears within a **blockquote**, because class selectors are more specific.

Understanding the concepts of inheritance and specificity is critical to mastering CSS, and there is a lot more to be said about specificity. The **"More About Specificity"** sidebar provides useful references.

Now, back to the menu. Fortunately, our Black Goose Bistro page has been marked up thoroughly and semantically, so we have a lot of options for selecting specific elements. Give these new selectors a try in EXERCISE 12-5.

More About Specificity

The specificity overview in this chapter is enough to get you started, but when you get more experienced and your style sheets become more complicated, you may find that you need a more thorough understanding of the inner workings.

For the technical explanation of exactly how specificity is calculated, see the CSS Selectors Module Level 4 specification at *www.w3.org/TR/selectors4/#specificity*.

Eric Meyer provides a thorough, yet more digestible, description of this system in his book *Selectors, Specificity, and the Cascade: Applying CSS to Documents* (O'Reilly). This material is also included in his book co-authored with Estelle Weyl, *CSS: The Definitive Guide, 4e* (O'Reilly).

If you are looking for help online, I recommend the *Smashing Magazine* article "CSS Specificity: Things You Should Know" (*coding.smashingmagazine.com/2007/07/27/css-specificity-things-you-should-know/*) by Vitaly Friedman. It's over a decade old, but the concepts hold true.

As for most web design topics, the MDN Web Docs site provides a comprehensive explanation: *developer.mozilla.org/en-US/docs/Web/CSS/Specificity*.

The Universal Selector

The universal element selector (*) matches any element , like a wildcard in programming languages. The style rule

```
* { border: 1px solid gray; }
```

puts a 1-pixel gray border around every element in the document. It is also useful as a contextual selector, as shown in this example that selects all elements in an "intro" section:

```
#intro * { color: gray; }
```

Be aware that every element will be selected with the universal selector, including some that you might not be expecting to style. For example, some styles might mess up your form controls, so if your page contains form inputs, the safest bet is to avoid the universal selector.

EXERCISE 12-5. Using selectors

This time, we'll add a few more style rules using descendant, ID, and class selectors combined with the **font** and **color** properties we've learned about so far.

1. I'd like to add some attention-getting color to the "new item!" elements next to certain menu item names. They are marked up as **strong**, so we can apply the **color** property to the **strong** element. Add this rule to the embedded style sheet, save the file, and reload it in the browser:

```
strong {
  font-style: italic;
  color: tomato;
}
```

   That worked, but now the **strong** element "Very spicy" in the description is "tomato" red too, and that's not what I want. The solution is to use a contextual selector that targets only the **strong** elements that appear in **dt** elements. Remove the **color** declaration you just wrote from the **strong** rule, and create a new rule that targets only the **strong** elements within definition list terms:

```
dt strong { color: tomato; }
```

2. Look at the document source, and you will see that the content has been divided into three unique **div**s: **info**, **appetizers**, and **entrees**. We can use these to our advantage when it comes to styling. For now, let's do something simple and apply a teal color to the text in the **div** with the ID "info". Because color inherits, we need to apply the property only to the **div** and it will be passed down to the **h1** and **p**:

```
#info { color: teal; }
```

3. Now let's get a little fancier and make the paragraph inside the "info" section italic in a way that doesn't affect the other paragraphs on the page. Again, a contextual selector is the answer. This rule selects only paragraphs contained within the **info** section of the document:

```
#info p { font-style: italic; }
```

4. I want to give special treatment to all of the prices on the menu. Fortunately, they have all been marked up with **span** elements:

```
<span class="price">$3.95</span>
```

   So now all we have to do is write a rule using a class selector to change the font to Georgia or some serif font, make the prices italic, and gray them back:

```
.price {
  font-family: Georgia, serif;
  font-style: italic;
  color: gray;
}
```

5. Similarly, in the "info" **div**, I can change the appearance of the spans that have been marked up as belonging to the "label" class to make the labels stand out:

```
.label {
  font-weight: bold;
  font-variant: small-caps;
  font-style: normal;
}
```

6. Finally, there is a warning at the bottom of the page that I want to make small and red. It has been given the class "warning," so I can use that as a selector to target just that paragraph for styling. While I'm at it, I'm going to apply the same style to the **sup** element (the footnote asterisk) earlier on the page so they match. Note that I've used a grouped selector, so I don't need to write a separate rule.

```
p.warning, sup {
  font-size: small;
  color: red;
}
```

FIGURE 12-11 shows the results of all these changes. We now have some touches of color and special typography treatments.

Black Goose Bistro • Summer Menu

Baker's Corner, Seekonk, Massachusetts
HOURS: MONDAY THROUGH THURSDAY: *11 to 9*, FRIDAY AND SATURDAY; *11 to midnight*

Appetizers

This season, we explore the spicy flavors of the southwest in our appetizer collection.

Black bean purses
Spicy black bean and a blend of mexican cheeses wrapped in sheets of phyllo and baked until golden. *$3.95*

Southwestern napoleons with lump crab — *new item!*
Layers of light lump crab meat, bean and corn salsa, and our handmade flour tortillas. *$7.95*

Main courses

Big, bold flavors are the name of the game this summer. Allow us to assist you with finding the perfect wine.

Jerk rotisserie chicken with fried plantains — *new item!*
Tender chicken slow-roasted on the rotisserie, flavored with spicy and fragrant jerk sauce and served with fried plantains and fresh mango. ***Very spicy.*** *$12.95*

Shrimp sate kebabs with peanut sauce
Skewers of shrimp marinated in lemongrass, garlic, and fish sauce then grilled to perfection. Served with spicy peanut sauce and jasmine rice. *$12.95*

Grilled skirt steak with mushroom fricasee
Flavorful skirt steak marinated in asian flavors grilled as you like it*. Served over a blend of sauteed wild mushrooms with a side of blue cheese mashed potatoes. *$16.95*

* We are required to warn you that undercooked food is a health risk.

FIGURE 12-11. The current state of the bistro menu.

TEXT LINE ADJUSTMENTS

The next batch of text properties has to do with the treatment of whole lines of text rather than the shapes of characters. They allow web authors to format web text with indents, extra space between lines (leading), and different horizontal alignments, similar to print.

Line Height

line-height

Values: *number* | *length measurement* | *percentage* | `normal`

Default: `normal`

Applies to: all elements

Inherits: yes

The **`line-height`** property defines the minimum distance from baseline to baseline in text. We saw it earlier as part of the shorthand **`font`** property. The **`line-height`** property is said to specify a "minimum" distance because if you put a tall image or large characters on a line, the height of that line expands to accommodate it.

A baseline is the imaginary line upon which the bottoms of characters sit. Setting a line height in CSS is similar to adding leading in traditional typesetting; however, instead of space being added between lines, the extra space is split above and below the text. The result is that **`line-height`** defines the height of a line-box in which the text line is vertically centered (FIGURE 12-12).

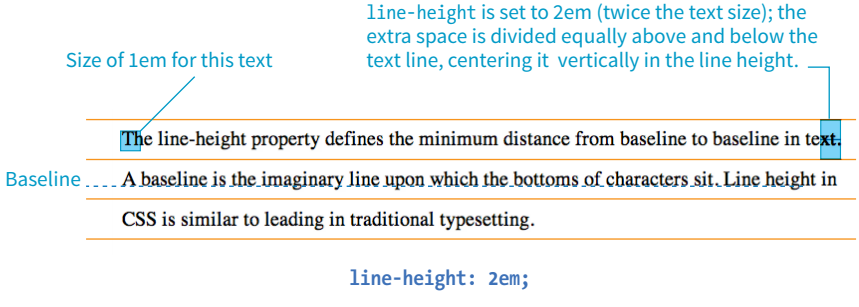


FIGURE 12-12. Text lines are centered vertically in the line height.

These examples show three different ways to make the line height twice the height of the font size:

```
p { line-height: 2; }
p { line-height: 2em; }
p { line-height: 200%; }
```

When a number is specified alone, as shown in the first example, it acts as a scaling factor that is multiplied by the current font size to calculate the **line-height** value.

Line heights can also be specified in one of the CSS length units. Ems and percentage values are based on the current font size of the element. In the three examples, if the font size is 16 pixels, the calculated line height would be 32 pixels (see FIGURE 12-12).

The difference between using a scaling factor (number value) and a relative value (em or %) is how they inherit. If you set the line height with a scaling factor for a whole document on the **body** element, its descendants inherit the multiplier. If the scaling factor is set to 2 for the **body**, a 24-pixel headline will end up with a line height of 48 pixels.

If you set the **line-height** on the **body** element using ems or percentages, its descendants inherit the *calculated* size based on the body's font size. For example, if the line height is set to **1em** for the **body** element (calculated at 16 pixels), a 24-pixel headline inherits the calculated 16-pixel line height, not the 1em value. This is likely not the effect you are after, making number values a more intuitive option.

Indents

The **text-indent** property indents the first line of text by a specified amount.

text-indent

Values: *length measurement* | *percentage*

Default: 0

Applies to: block containers

Inherits: yes

You can specify a length measurement or a percentage value for **text-indent**. The results are shown in FIGURE 12-13. Here are a few examples:

```
p#1 { text-indent: 2em; }
p#2 { text-indent: 25%; }
p#3 { text-indent: -35px; }
```

Percentage values are calculated based on the width of the *parent* element, and they are passed down to their descendant elements as percentage values (not calculated values). So if a **div** has a **text-indent** of 10%, so will all of its descendants.

In the third example, notice that a negative value was specified, and that's just fine. It will cause the first line of text to hang out to the left of the left text edge (also called a hanging indent).

NOTE

The **text-indent** *property indents just the first line of a block. If you want space along the whole side of the text block, use one of the* **margin** *or* **padding** *properties to add it.*

Designers may be accustomed to specifying indents and margins in tandem, but to be consistent with how CSS handles them, margins will be discussed as part of the box model in **Chapter 14**.

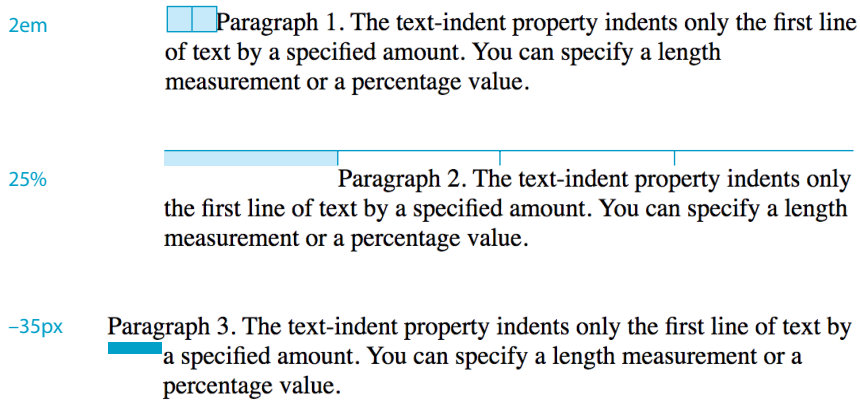


FIGURE 12-13. Examples of the **text-indent** property.

DESIGN TIP

If you use a hanging indent, be sure that there is also a left padding applied to the element. Otherwise, the hanging text may disappear off the left edge of the browser window.

Horizontal Text Alignment

You can align text for web pages just as you would in a word processing or desktop publishing program with the **text-align** property.

text-align

Values: left | right | center | justify | start | end

Default: start

Applies to: block containers

Inherits: yes

This is a fairly straightforward property to use. The results of the various CSS2.1 **text-align** values are shown in FIGURE 12-14.

text-align: left	Aligns text on the left margin
text-align: right	Aligns text on the right margin
text-align: center	Centers the text in the text block
text-align: justify	Aligns text on both right and left margins

The CSS Text Module Level 3 added the **start** and **end** values, which specify the side of the line box the text should align to (see **Note**). This accommodates languages that are written vertically and right to left. For left-to-right reading languages, **start** corresponds to **left**.

Good news—only five more text properties to go! Then we'll be ready to try a few of them in the Black Goose Bistro menu.

NOTE

*The CSS Text Module Level 3 also defines two new properties related to text alignment—***text-align-last** *(for aligning the last line of text) and* **text-justify** *(for more fine-tuned control over how space is inserted in justified text).*

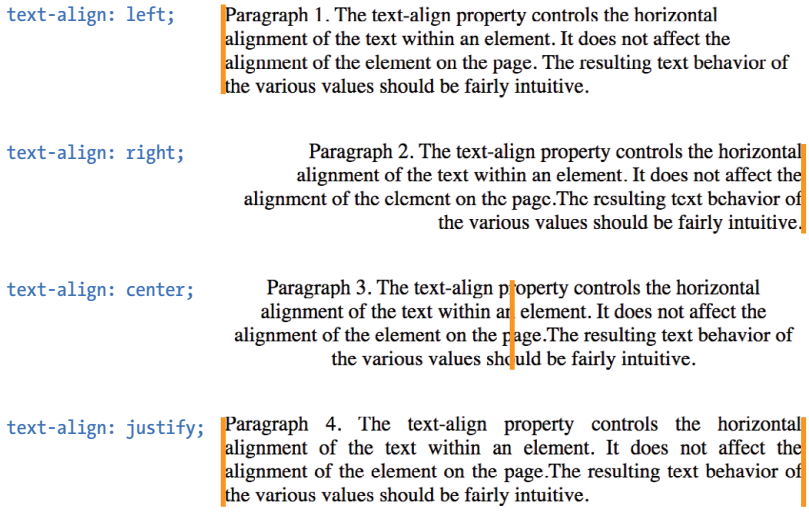


FIGURE 12-14. Examples of CSS2.1 **text-align** values.

NOTE

The CSS3 Text Module includes enhancements to **text-decoration**, *including* **text-decoration-line**, **text-decoration-color**, **text-decoration-style**, **text-decoration-skip**, *and* **text-underline-position**. *No version of IE or Edge supports these properties, but with the exception of* **-skip**, *they are supported in other modern browsers. See CanIUse.com for specifics.*

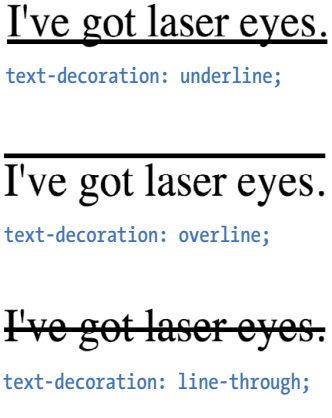


FIGURE 12-15. Examples of **text-decoration** values.

UNDERLINES AND OTHER "DECORATIONS"

If you want to put a line under, over, or through text, or if you'd like to turn of the underline under links, then **text-decoration** is the property for you.

text-decoration

Values:	none | underline | overline | line-through | blink
Default:	none
Applies to:	all elements
Inherits:	no, but since lines are drawn across child elements, they may look like they are "decorated" too

The values for **text-decoration** are intuitive and are shown in FIGURE 12-15.

underline	Underlines the element
overline	Draws a line over the text
line-through	Draws a line through the text

The most popular use of the **text-decoration** property is turning off the underlines that appear automatically under linked text, as shown here:

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

There are a few cautionary words to be said regarding **text-decoration**:

- First, if you get rid of the underlines under links, be sure there are other cues to compensate, such as color and weight.

- On the flip side, because underlines are such a strong visual cue to "click here," underlining text that is *not* a link may be misleading and frustrating. Consider whether italics may be an acceptable alternative.
- Finally, there is no reason to make your text blink. Browser makers agree and therefore have dropped support for blinking text. IE never supported it in the first place.

CHANGING CAPITALIZATION

I remember when desktop publishing programs introduced a feature that let me change the capitalization of text on the fly (OK, I'm dating myself here). This made it easy to see how my headlines might look in all capital letters without needing to retype them. CSS includes this feature as well with the **text-transform** property.

text-transform

Values: none | capitalize | lowercase | uppercase | full-width

Default: none

Applies to: all elements

Inherits: yes

When you apply the **text-transform** property to a text element, it changes its capitalization when it renders without changing the way it is typed in the source. The values are as follows (FIGURE 12-16):

none	As it is typed in the source
capitalize	Capitalizes the first letter of each word
lowercase	Makes all letters lowercase
uppercase	Makes all letters uppercase
full-width	Chooses a "full-width" version of a character if one exists (not well supported)

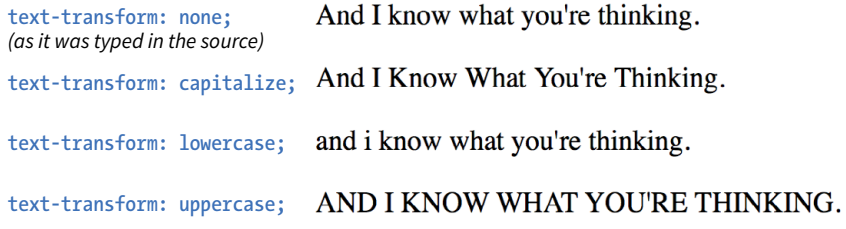


FIGURE 12-16. The **text-transform** property changes the capitalization of characters when they are displayed, regardless of how they are typed in the source.

SPACED OUT

The next two text properties are used to insert space between letters (**letter-spacing**) or words (**word-spacing**) when the text is displayed.

letter-spacing

Values: *length measurement* | normal

Default: normal

Applies to: all elements

Inherits: yes

word-spacing

Values: *length measurement* | normal

Default: normal

Applies to: all elements

Inherits: yes

The **letter-spacing** and **word-spacing** properties do what they say: add space between the letters of the text or words in a line, respectively.

FIGURE 12-17 shows the results of letter spacing and word spacing applied to the simple paragraph shown here:

```
<p>Black Goose Bistro Summer Menu</p>
```

B l a c k G o o s e B i s t r o S u m m e r M e n u

```
p { letter-spacing: 8px; }
```

Black Goose Bistro Summer Menu

```
p { word-spacing: 1.5em; }
```

FIGURE 12-17. **letter-spacing** (top) and **word-spacing** (bottom).

It is worth noting that when you specify em measurements, the calculated size is passed down to child elements, even if they have a smaller font size than the parent.

In EXERCISE 12-6 later in this chapter, we'll make one last trip back to the Black Goose Bistro menu and use the **letter-spacing** property on **h2**s.

TEXT SHADOW

The **text-shadow** property adds a "shadow" below your text that makes it seem to hover or pop out above the page. Since flat-color design has become the fashion, drop shadows have gone out of style, but they can still be a useful visual tool, particularly when your text is in front of a patterned or photographic background.

Text shadows are drawn behind the text but in front of the background and border if there is one. Text shadows are supported by all current browsers. Internet Explorer versions 9 and earlier lack support.

text-shadow

Values: *'horizontal offset' 'vertical offset' 'blur radius' 'color'* | none

Default: none

Applies to: all elements

Inherits: yes

The value for the **text-shadow** property is two or three measurements (a horizontal offset, vertical offset, and an optional blur radius) and a color. FIGURE 12-18 shows an example of a minimal text shadow declaration.

```
h1 {
  color: darkgreen;
  text-shadow: .2em .2em silver;
}
```

```
h1 {
  color: darkgreen;
  text-shadow: -.3em -.3em silver;
}
```

text-shadow: .2em .2em silver;

text-shadow: -.3em -.3em silver;

FIGURE 12-18. A minimal text drop shadow.

The first value is a horizontal offset that positions the shadow to the right of the text (a negative value pulls the shadow to the *left* of the text). The second measurement is a vertical offset that moves the shadow down by the specified amount (a negative value moves the shadow *up*). The declaration ends with the color specification (silver). If the color is omitted, the text color will be used.

That should give you an idea for how the first two measurements work, but that sharp shadow doesn't look very...well...shadowy. What it needs is a blur radius measurement. Zero (0) is no blur, and the blur gets softer with higher values (FIGURE 12-19). Usually, you just have to fiddle with values until you get the effect you want.

text-shadow: .2em .2em .1em silver;

text-shadow: .2em .2em .3em silver;

FIGURE 12-19. Adding a blur radius to a text drop shadow.

It is possible to apply several text shadows to the same element. If you vary the position and blur amounts, you can give the text the appearance of multiple light sources.

So go have some fun with text shadows, but be careful not to overdo it. Not only can drop shadows make text difficult to read, but adding a shadow to everything can slow down page performance (scrolling, mouse interactions, etc.) as well, which is particularly problematic for mobile browsers without much processing power. In addition, be careful that your text doesn't require a shadow in order to be visible. Folks with non-supporting browsers won't see a thing. My advice is to use drop shadows as an enhancement in a way that isn't critical if they don't appear.

EXERCISE 12-6 gives you a chance to try out more text formatting properties to put a little polish on the Black Goose Bistro menu.

The Other Text Properties

In the interest of saving space and keeping this an introductory-level book, I haven't given these properties the full treatment, but they are worth mentioning. Each is labeled with the CSS Level in which it was introduced.

For even more text-related properties in development, see the following CSS Text Modules:

- CSS Text Module Level 3: *www.w3.org/TR/css-text-3*
- CSS Text Decoration Module Level 3: *www.w3.org/TR/css-text-decor-3*
- CSS Text Module Level 4 (still in Working Draft and considered experimental): *www.w3.org/TR/css-text-4*

`white-space` (*CSS2*) Specifies how whitespace in the element source is handled in layout. For example, the **`pre`** value preserves the character spaces and returns found in the source, similar to the **`pre`** HTML element.

`vertical-align` (*CSS2*) Specifies the vertical alignment of an inline element's baseline relative to the baseline of the surrounding text. It is also used to set the vertical alignment of content in a table cell (**`td`**).

`word-break` *and* **`line-break`** (*CSS3*) Affects how text wrapping is calculated within words and lines, respectively, in various languages, including East Asian (Chinese, Japanese, Korean).

`text-justify` (*CSS3*) Specifies the manner in which space is to be added within and between words when the **`text-align`** property on the element is set to **`justify`**.

`text-align-last` (*CSS3*) Specifies how the last line of a block of text should be justified when the **`text-align`** property on the element is set to **`justify`**. For example, it is often preferable to have the last line left-justified for justified text to avoid awkwardly spaced words.

`tab-size` (*CSS3*) Specifies the length of the tab character (Unicode point 0009) in number of characters or a length measurement.

`hyphens` (*CSS3*) Provides control over how text is hyphenated. **`manual`** means hyphenation happens only when there is a hyphen added in the source. **`auto`** gives control to the browser, and **`none`** turns off hyphenation completely.

`overflow-wrap` (*CSS3*) Specifies whether browsers are allowed to break words to fit text in its bounding box.

`hanging-punctuation` (*CSS3*) Determines whether the punctuation mark may be outside the element's line box at the start or end of a line. Hanging punctuation can make margins appear more tidy.

The following properties are in the spec, but should not be used. Use the **`dir`** HTML attribute instead.

`direction` (*CSS3*) Specifies the direction in which the text reads: left to right (**`ltr`**) or right to left (**`rtl`**).

`unicode-bidi` (*CSS2*) Related to bidirectional features of Unicode. The Recommendation states that it allows the author to generate levels of embedding within the Unicode embedding algorithm. If you have no idea what this means, don't worry. Neither do I.

EXERCISE 12-6. Finishing touches

Let's add a few finishing touches to the online menu, *menu.html*. It might be useful to save the file and look at it in the browser after each step to see the effect of your edits and to make sure you're on track. The finished style sheet is provided in the *materials* folder for this chapter.

1. First, I have a few global changes to the **body** element in mind. I've had a change of heart about the **font-family**. I think that a serif font such as Georgia would be more sophisticated and appropriate for a bistro menu. Let's also use the **line-height** property to open up the text lines and make them easier to read. Make these updates to the **body** style rule, as shown:

```
body {
  font-family: Georgia, serif;
  font-size: small;
  line-height: 1.75em;
}
```

2. I also want to redesign the "info" section of the document. Remove the teal color setting by deleting that whole rule. Once that is done, make the **h1** olive green and the paragraph in the header gray. Add color declarations to the existing rules:

```
~~#info { color: teal; }~~ /* delete */
h1 {
  font: bold 1.5em "Marko One", Georgia, serif;
  color: olive;}
#info p {
  font-style: italic;
  color: gray;}
```

3. Next, to imitate a fancy restaurant menu, I'm going to center a few key elements on the page with the **text-align** property. Write a rule with a grouped selector to center the headings and the "info" section:

```
h1, h2, #info {
  text-align: center;}
```

4. I want to make the "Appetizer" and "Main Courses" **h2** headings more eye-catching. Instead of large, bold type, I'm going to use all uppercase letters, extra letter spacing, and color to call attention to the headings. Here's the new rule for **h2** elements that includes all of these changes:

```
h2 {
  font-size: 1em;
  text-transform: uppercase;
  letter-spacing: .5em;
  color: olive;}
```

5. We're really close now; just a few more tweaks to those paragraphs right after the **h2** headings. Let's center those too and make them italic:

```
h2 + p {
  text-align: center;
  font-style: italic;}
```

   Note that I've used a next-sibling selector (**h2 + p**) to select any paragraph that follows an **h2**.

6. Next, add a softer color to the menu item names (in **dt** elements). I've chosen "sienna," one of the names from the CSS3 color module. Note that the **strong** elements in those **dt** elements stay "tomato" red because the color applied to the **strong** elements overrides the color inherited by their parents.

```
dt {
  font-weight: bold;
  color: sienna;}
```

7. Finally, for kicks, add a drop shadow under the **h1** heading. You can play around with the values a little to see how it works. I find it to look a little clunky against a white background, but when you have a patterned background image, sometimes a drop shadow provides the little punch you need to make the text stand out. Notice how small the shadow values are—a little goes a long way!

```
h1 {
  font: bold 1.5em "Marko One", Georgia, serif;
  color: olive;
  text-shadow: .05em .05em .1em lightslategray;}
```

And we're done! FIGURE 12-20 shows how the menu looks now—an improvement over the unstyled version, and we used only text and color properties to do it. Notice that we didn't touch a single character of the document markup in the process. That's the beauty of keeping style separate from structure.

Black Goose Bistro • Summer Menu

Baker's Corner, Seekonk, Massachusetts

HOURS: MONDAY THROUGH THURSDAY: *11 to 9*, FRIDAY AND SATURDAY: *11 to midnight*

APPETIZERS

This season, we explore the spicy flavors of the southwest in our appetizer collection.

Black bean purses
Spicy black bean and a blend of mexican cheeses wrapped in sheets of phyllo and baked until golden. *$3.95*

Southwestern napoleons with lump crab — *new item!*
Layers of light lump crab meat, bean and corn salsa, and our handmade flour tortillas. *$7.95*

MAIN COURSES

Big, bold flavors are the name of the game this summer. Allow us to assist you with finding the perfect wine.

Jerk rotisserie chicken with fried plantains — *new item!*
Tender chicken slow-roasted on the rotisserie, flavored with spicy and fragrant jerk sauce and served with fried plantains and fresh mango. ***Very spicy.*** *$12.95*

Shrimp sate kebabs with peanut sauce
Skewers of shrimp marinated in lemongrass, garlic, and fish sauce then grilled to perfection. Served with spicy peanut sauce and jasmine rice. *$12.95*

Grilled skirt steak with mushroom fricasee
Flavorful skirt steak marinated in asian flavors grilled as you like it*. Served over a blend of sauteed wild mushrooms with a side of blue cheese mashed potatoes. *$16.95*

* We are required to warn you that undercooked food is a health risk.

FIGURE 12-20. The formatted Black Goose Bistro menu.

CHANGING LIST BULLETS AND NUMBERS

Before we close out this chapter on text properties, I want to show you a few tweaks you can make to bulleted and numbered lists. As you know, browsers automatically insert bullets before unordered list items, and numbers before items in ordered lists (the list markers). For the most part, the rendering of these markers is determined by the browser. However, CSS provides a few properties that allow authors to choose the type and position of the marker, or turn them off entirely.

Choosing a Marker

Apply the **list-style-type** property to the **ul**, **ol**, or **li** element select the type of marker that appears before each list item (see **Note**).

NOTE

This section documents the CSS2.1 **list-style** *types that are well supported on current browsers. CSS3 extends the marker functionality shown here, including a method for authors to define their own list styles, allowing for numbering in many languages (www.w3.org/TR/css3-lists/).*

list-style-type

Values: none | disc | circle | square | decimal | decimal-leading-zero | lower-alpha | upper-alpha | lower-latin | upper-latin | lower-roman | upper-roman | lower-greek

Default: disc

Applies to: ul, ol, and li (or elements whose display value is list-item)

Inherits: yes

More often than not, developers use the **list-style-type** property with its value set to **none** to remove bullets or numbers altogether. This is handy when you're using list markup as the foundation for a horizontal navigation menu or the entries in a web form. You can keep the semantics but get rid of the pesky markers.

NOTE

CSS3 introduces the **@counter-style** *rule, which provides* **box**, **check**, **diamond**, *and* **dash** *marker types as well as the ability to specify your own markers when a predefined one won't do. See the spec for details.*

The **disc**, **circle**, and **square** values generate bullet shapes just as browsers have been doing since the beginning of the web itself (FIGURE 12-21). Unfortunately, there is no way to change the appearance (size, color, etc.) of generated bullets, so you're stuck with the browser's default rendering.

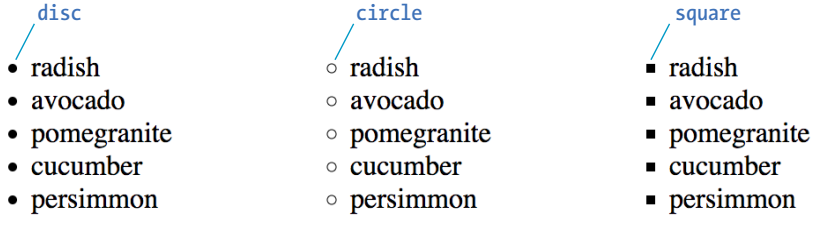


FIGURE 12-21. The **list-style-type** values **disc**, **circle**, and **square**.

The remaining keywords (TABLE 12-1) specify various numbering and lettering styles for use with ordered lists.

TABLE 12-1. Lettering and numbering system (CSS2.1)

Keyword	System
decimal	1, 2, 3, 4, 5...
decimal-leading-zero	01, 02, 03, 04, 05...
lower-alpha	a, b, c, d, e...
upper-alpha	A, B, C, D, E...
lower-latin	a, b, c, d, e... (same as lower-alpha)
upper-latin	A, B, C, D, E... (same as upper-alpha)
lower-roman	i, ii, iii, iv, v...
upper-roman	I, II, III, IV, V...
lower-greek	α, β, γ, δ, ε...

List Item Display Role

You may have noticed that the list style properties apply to "elements whose display value is **list-item**." The CSS2.1 specification allows any element to perform like a list item by setting its **display** property to **list-item**. This property can be applied to any HTML element or elements in other XML languages. For example, you could automatically bullet or number a series of paragraphs by setting the **display** property of paragraph (**p**) elements to **list-item**, as shown in this example:

```
p.lettered {
 display: list-item;
 list-style-type: upper-alpha;
}
```

Marker Position

By default, the marker hangs outside the content area for the list item, displaying as a hanging indent. The **list-style-position** property allows you to pull the bullet inside the content area so it runs into the list content.

list-style-position

Values: inside | outside | hanging

Default: outside

Applies to: **ul**, **ol**, and **li** (or elements whose display value is **list-item**)

Inherits: yes

I've applied a light green background color to the list items in FIGURE 12-22 to reveal the boundaries of their content area boxes.

You can see that when the position is set to **outside** (top), the markers fall outside the content area.When it is set to **inside** (bottom), the markers are tucked into the content area.

```
li {background-color: #F99;}
ul#outside {list-style-position: outside;}
ul#inside {list-style-position: inside;}
```

CSS3 adds the **hanging** value for **list-style-position**. it is similar to **inside**, but the markers appear outside and abutting the left edge of the shaded area.

outside

- **Radish.** Praesent in lacinia risus. Morbi urna ipsum, efficitur id erat pellentesque, tincidunt commodo sem. Phasellus est velit, porttitor vel dignissim vitae, commodo ut urna.
- **Avocado.** Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Curabitur lacinia accumsan est, ut malesuada lorem consectetur eu.
- **Pomegranite.** Nam euismod a ligula ac bibendum. Aenean ac justo eget lorem dapibus aliquet. Vestibulum vitae luctus orci, id tincidunt nunc. In a mauris odio. Duis convallis enim nunc.

inside

- **Radish.** Praesent in lacinia risus. Morbi urna ipsum, efficitur id erat pellentesque, tincidunt commodo sem. Phasellus est velit, porttitor vel dignissim vitae, commodo ut urna.
- **Avocado.** Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Curabitur lacinia accumsan est, ut malesuada lorem consectetur eu.
- **Pomegranite.** Nam euismod a ligula ac bibendum. Aenean ac justo eget lorem dapibus aliquet. Vestibulum vitae luctus orci, id tincidunt nunc. In a mauris odio. Duis convallis enim nunc.

FIGURE 12-22. The **list-style-position** property.

Make Your Own Bullets

You can also use your own image as a bullet by using the **list-style-image** property.

list-style-image

Values: url(*location*) | none

Default: none

Applies to: **ul**, **ol**, and **li** (or elements whose display value is **list-item**)

Inherits: yes

The value of the **list-style-image** property is the URL of the image you want to use as a marker. The **list-style-type** is set to **disc** as a backup in case the image does not display or the property isn't supported by the browser or other user agent. The result is shown in FIGURE 12-23.

```
ul {
  list-style-type: disc;
  list-style-image: url(/images/rainbow.gif);
  list-style-position: outside;
}
```

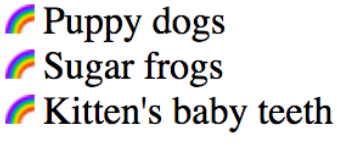


FIGURE 12-23. Using an image as a marker.

CSS TIP

There is a **list-style** shorthand property that combines the values for type, position, and image, in any order. For example:

```
ul {
  list-style: url(/images/rainbow.gif) disc outside;
}
```

As for all shorthands, be careful not to override list style properties set earlier in the style sheet.

Wow! Whatta chapter! We started by looking at properties for specifying fonts and character shapes followed by a review of all the text-level settings and effects. You also got to use descendent, ID, and class selectors and looked a little more closely at specificity. We topped it off with the properties available for adding some style to lists. I don't expect you to have all of these properties committed to memory (although many will become second nature the more you practice), but let's see how you do on the following questions.

TEST YOURSELF

It's time to see how well you understand the font properties and selectors introduced in this chapter. Check **Appendix A** for the answers if you get stuck.

1. Match the style property with the text samples in FIGURE 12-24.

   a. _____ `{font-size: 1.5em;}`

   b. _____ `{text-transform: capitalize;}`

   c. _____ `{text-align: right;}`

   d. _____ `{font-family: Verdana; font-size: 1.5em;}`

   e. _____ `{letter-spacing: 3px;}`

   f. _____ `{font: bold italic 1.2em Verdana;}`

   g. _____ `{text-transform: uppercase;}`

   h. _____ `{text-indent: 2em;}`

   i. _____ `{font-variant: small-caps;}`

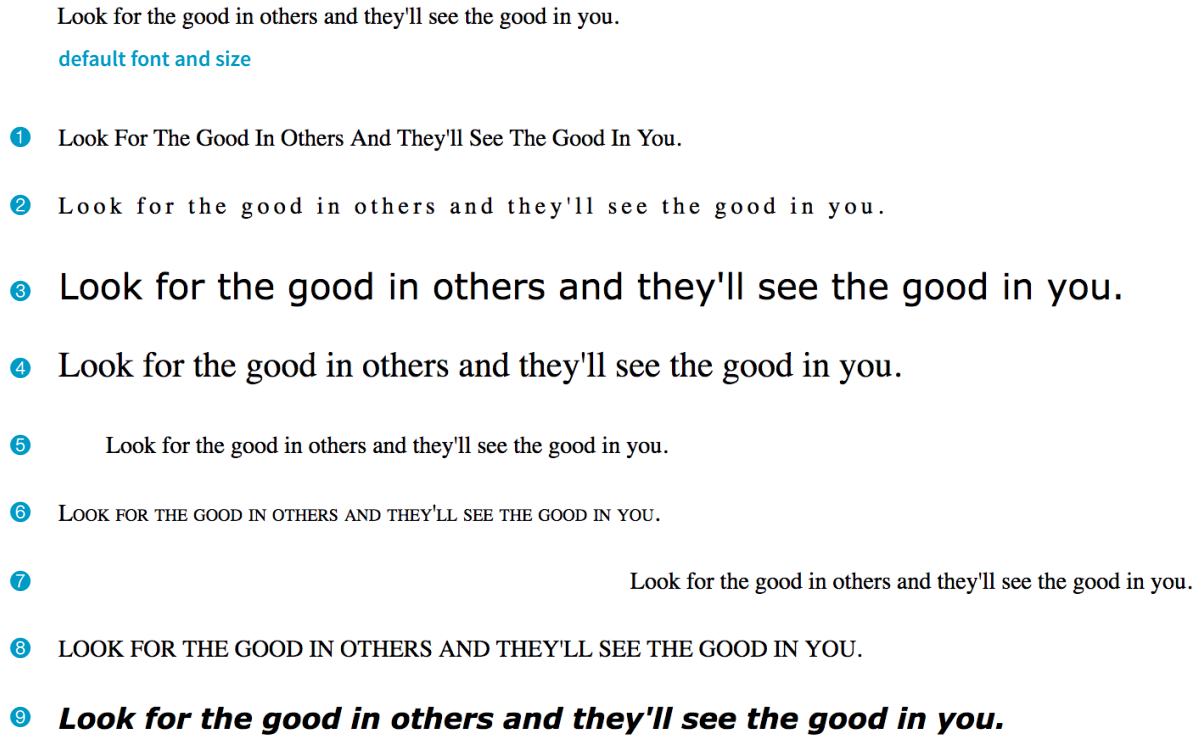


FIGURE 12-24. Styled text samples.

2. Here is a chance to get a little practice writing selectors. Using the diagram shown in FIGURE 12-25, write style rules that make each of the elements described here red (`color: red;`). Write the selector as efficiently as possible.

   a. All text elements in the document

   b. `h2` elements

   c. `h1` elements and all paragraphs

   d. Elements belonging to the class `special`

   e. All elements in the "intro" section

   f. `strong` elements in the "main" section

   g. Extra credit: just the paragraph that appears after an `h2`

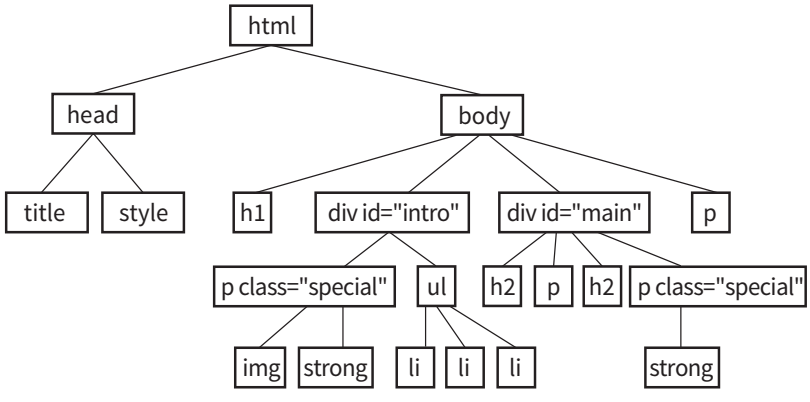


FIGURE 12-25. Sample document structure.

CSS REVIEW: FONT AND TEXT PROPERTIES

In this chapter, we covered the properties used to format text elements. Here is a summary in alphabetical order.

Property	Description
color	Specifies the foreground color (text and borders) for an element
direction	Indicates whether the text reads left-to-right or right-to-left
font	A shorthand property that combines font properties
font-family	Specifies a typeface or generic font family
font-feature-settings	Allows access to lesser-used OpenType features
font-kerning	Controls how browsers implement kerning data (space between characters)
font-language-override	Controls use of language-specific glyphs
font-size	Specifies the size of the font
font-size-adjust	Matches the x-height of a fallback font with the specified font
font-stretch	Selects a condensed, normal, or extended font
font-style	Specifies italic or oblique fonts
font-synthesis	Controls whether a browser may simulate bold or italic fonts
font-variant	Specifies a small-caps font
font-variant-alternates	Selects alternate versions of character glyphs
font-variant-caps	Selects small caps and similar alternates when available
font-variant-east-asian	Selects alternate glyphs in Chinese, Japanese, and Korean
font-variant-ligatures	Selects ligatures for certain letter pairs when available
font-variant-numeric	Selects alternate number glyphs
font-variant-position	Selects subscript or superscript character glyphs
font-weight	Specifies the boldness of the font
hanging-punctuation	Indicates whether the punctuation may hang outside the content box
hyphens	Controls how text is hyphenated
letter-spacing	Inserts space between letters
line-break	Describes rules for breaking lines
line-height	Indicates the distance between baselines of neighboring text lines

Property	Description
`list-style-image`	Specifies an image to be used as a list marker
`list-style-position`	Puts a list marker inside or outside the content area
`list-style-type`	Selects the marker type for list items
`overflow-wrap`	Specifies whether the browser can break lines within words to prevent overflow
`tab-size`	Specifies the length of a tab character
`text-align`	Indicates the horizontal alignment of text
`text-align-last`	Specifies how the last line in justified text is aligned
`text-decoration`	Specifies underlines, overlines, and lines through
`text-indent`	Specifies the amount of indentation of the first line in a block
`text-justify`	Denotes how space is distributed in justified text
`text-shadow`	Adds a drop shadow under the text
`text-transform`	Changes the capitalization of text when it displays
`unicode-bidi`	Works with Unicode bidirectional algorithms
`vertical-align`	Adjusts the vertical position of inline elements relative to the baseline
`white-space`	Specifies how whitespace in the source is displayed
`word-break`	Specifies whether to break lines within words
`word-spacing`	Inserts space between words
`word-wrap`	Indicates whether the browser can break lines within words to prevent overflow (same as **`overflow-wrap`**)

CHAPTER 13

COLORS AND BACKGROUNDS

PLUS MORE SELECTORS AND EXTERNAL STYLE SHEETS

If you had seen the web back in 1993, you would have found it to be a dreary affair by today's standards—every background was gray, and all the text was black. Then came Netscape Navigator and, with it, a handful of HTML attributes that allowed rudimentary (but welcome) control over font colors and backgrounds. For years, we made do. But thankfully, we now have style sheet properties that have laid those unmentionable presentational attributes to rest.

We're going to cover a *lot* of ground in this chapter. Of course, I'll introduce you to all of the properties for specifying colors and backgrounds. This chapter also rounds out your collection of selector types and shows you how to create an external style sheet. Our first order of business, however, is to explore the options for specifying color in CSS, including a primer on the nature of color on computer monitors.

IN THIS CHAPTER

SPECIFYING COLOR VALUES

There are two main ways to specify colors in style sheets—with a predefined color name, as we have been doing so far:

```
color: red;        color: olive;        color: blue;
```

Or, more commonly, with a numeric value that describes a particular RGB color (the color model on computer monitors). You may have seen color values that look like these:

```
color: #FF0000;    color: #808000;        color: #00F;
```

We'll get to all the ins and outs of RGB color in a moment, but first, a short and sweet section on the standard color names.

Color Names

The most intuitive way to specify a color is to call it by name. Unfortunately, you can't make up just any color name and expect it to work. It has to be one of the color keywords predefined in the CSS Recommendation. CSS1 and CSS2 adopted the 16 standard color names originally introduced in HTML 4.01. CSS2.1 tossed in **orange** for a total of 17 (FIGURE 13-1).

FUN FACT

The extended color names, also known as the X11 color names, were originally provided with the X Window System for Unix.

CSS3 adds support for the extended set of 140 (rather fanciful) color names. Now we can specify names like **burlywood**, **peachpuff**, **oldlace**, and my long-time favorite, **papayawhip**! The extended colors are shown in FIGURE 13-2, but if you want a more accurate view, point your browser at *learningwebdesign.com/colornames.html*. CSS3 also added the **transparent** keyword, which can be used with any property that has a color value.

Color names are easy to use—just drop one into place as the value for any color-related property:

```
color: silver;
background-color: gray;
border-bottom-color: teal;
```

FIGURE 13-1. The 17 standard color names in CSS2.1. (Note that "gray" must be spelled with an "a.")

FIGURE 13-2. The 140 extended color names in CSS3. Bear in mind that these look quite different on a screen.

RGB Color Values

Names are easy, but as you can see, they are limited. By far, the most common way to specify a color is by its RGB value. It also gives you millions of colors to choose from.

For those who are not familiar with how computers deal with color, I'll start with the basics before jumping into the CSS syntax.

A word about RGB color

Computers create the colors you see on a monitor by combining three colors of light: red, green, and blue. This is known as the RGB color model. You can provide recipes (of sorts) for colors by telling the computer how much of each color to mix in. The amount of light in each color "channel" is typically described on a scale from 0 (none) to 255 (full blast), although it can also be provided as a percent. The closer the three values get to 255 (100%), the closer the resulting color gets to white (FIGURE 13-3). Wondering why the scale is from 0 to 255? See the **"Why 255?"** sidebar.

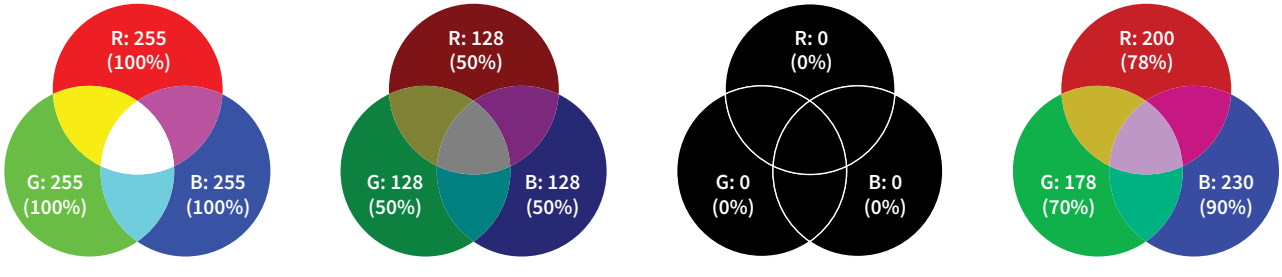
Why 255?

In true RGB color, 8 bits of information are devoted to each color channel. Because 8 bits can describe 256 shades ($2^8 = 256$), colors are measured on a scale from 0 to 255.

Any color you see on your monitor can be described by a series of three numbers: a red value, a green value, and a blue value. This is one of the ways that image editors such as Adobe Photoshop keep track of the colors for every pixel in an image. With the RGB color system, a pleasant lavender can be described as R:200, G:178, B:230.

Taken together, 255 colors in each channel can define around 16.7 million color combinations. This color space of millions of colors is known as Truecolor. There are different ways to encode those colors (that is, convert them to bytes for computers), and the web uses an encoding called sRGB. So, if you see an option for saving images as sRGB in a graphics program, click Yes.

The RGB Color Model



RGB: 255, 255, 255
white

RGB: 128, 128, 128
gray

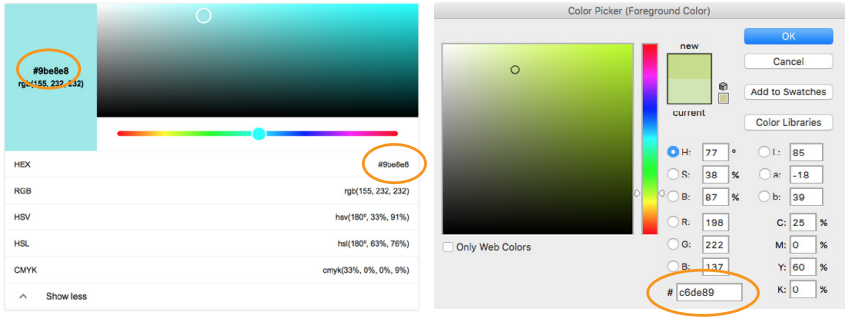
RGB: 0, 0, 0
black

RGB: 200, 178, 230
pleasant lavender

FIGURE 13-3. Computers create colors on a monitor by mixing different amounts of red, green, and blue light (thus, RGB). The color in the middle of each diagram shows what happens when the three color channels are combined. The more light there is in each channel (i.e., the higher the number value), the closer the combination is to white.

Picking a color

There are a number of ways to pick a color and find its RGB color values. One quick and easy option is to go to Google.com and search "color picker," and *voilà*—a full-featured color picker (FIGURE 13-4, left)! If you tend to keep an image-editing program such as Adobe Photoshop open and handy, you can use its built-in color picker (FIGURE 13-4, right).



Google color picker

Photoshop color picker

FIGURE 13-4. Color pickers such as the one at Google.com (search "color picker") and in Photoshop.

Both the Google and image editor color pickers show how the selected color would be expressed in a variety of color models (to reveal the values in Google, click "Show color values" below the picker). RGB is the most common for web design, so we're focusing our attention on that one. HSL (Hue Saturation Lightness or Luminosity) is another option for specifying color in style sheets, and we'll take a look at it in a moment (see **Note**). CMYK (Cyan Magenta Yellow blacK) is used primarily for print media, so you won't use it except perhaps to translate print colors to their screen equivalents.

When you select a color from the spectrum in the color picker, the red, green, and blue values are listed, as pointed out in FIGURE 13-4. And look next to the # symbol—those are the same three values, converted to hexadecimal equivalents so they are ready to go in a style sheet. I'll explain the six-digit hex values in a moment.

The Web Palette

You may come across the terms web palette or web-safe colors in web production tools like Dreamweaver or Photoshop. The web got its start in the days when computer monitors typically could display only 256 colors at a time. The web palette was a collection of 216 colors that could be displayed on both Windows and Macintosh operating systems without dithering, and thus they were "safe" for the web. That era is long behind us, as is the need to restrict our color choices to the web palette.

NOTE

HSL is not the same as HSB (Hue Saturation Brightness), another color model provided in Photoshop and other image editors.

Writing RGB values in style sheets

CSS allows RGB color values to be specified in a number of formats. Going back to that pleasant lavender, we could add it to a style sheet by listing each value on a scale from 0 to 255:

```
color: rgb(200, 178, 230);
```

You can also list them as percentage values, although that is less common:

```
color: rgb(78%, 70%, 90%);
```

■ AT A GLANCE

Specifying RGB Values

There are four formats for providing RGB values in CSS:

```
rgb(255, 255, 255)
rgb(100%, 100%, 100%)
#FFFFFF
#FFF
```

All of these examples specify white.

Or, you can provide the six-digit hexadecimal version that we saw in the color pickers. These six digits represent the same three RGB values, except they have been converted into hexadecimal (or hex for short) equivalents. Note that hex RGB values are preceded by the **#** symbol and do not require the **rgb()** notation shown in the previous examples. They may be upper- or lowercase, but it is recommended that you be consistent:

```
color: #C8B2E6;
```

There is one last shorthand way to specify hex color values. If your value happens to be made up of three pairs of double digits or letters, such as

```
color: #FFCC00; or color: #993366;
```

you can condense each pair down to one digit or letter. It's easier to type and to read, and it slightly reduces the size of your file. These examples are equivalent to the ones just listed:

```
color: #FC0;  or  color: #936;
```

About hexadecimal values

It's time to clarify what's going on with that six-digit string of characters. What you're looking at is actually a series of three two-digit numbers, one each for red, green, and blue. But instead of decimal (base-10, the system we're used to), these values are written in hexadecimal, or base-16. FIGURE 13-5 shows the structure of the hex RGB value.

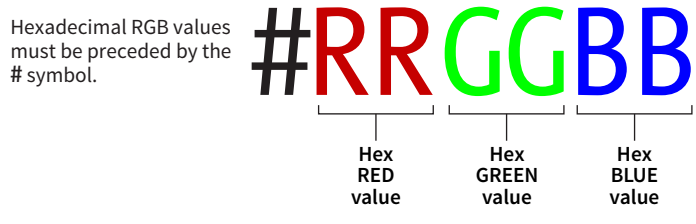


FIGURE 13-5. Hexadecimal RGB values are made up of three two-digit numbers, one for red, one for green, and one for blue.

■ TIP

Handy Hex Values

White = #FFFFFF or #FFF
(the equivalent of 255,255,255)

Black = #000000 or #000
(the equivalent of 0,0,0)

The hexadecimal numbering system uses 16 digits: 0–9 and A–F (for representing the quantities 10–15). FIGURE 13-6 shows how this works. The hex system is used widely in computing because it reduces the space it takes to store certain information. For example, the RGB values are reduced from three to two digits once they're converted to hexadecimal.

Now that most graphics and web development software provides easy access to hexadecimal color values (as we saw in FIGURE 13-4), there isn't much need to translate RGB values to hex yourself, as we needed to do back in the old days. Should you need to, there are plenty of decimal-to-hexadecimal converters online.

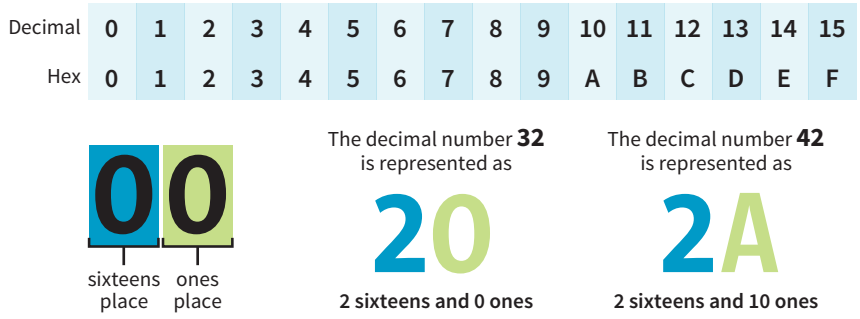


FIGURE 13-6. The hexadecimal numbering system is base-16.

RGBa Color

RGBa color allows you to specify a color and make it as transparent or as opaque as you like. The "a" in "RGBa" stands for alpha, which is an additional channel that controls the level of transparency on a scale from 0 (fully transparent) to 1 (fully opaque). Here's how it looks written in a style rule:

```
color: rgba(0, 0, 0, .5);
```

The first three values in the parentheses are regular old RGB values, in this case creating the color black. The fourth value, .5, is the transparency level. So this color is black with 50% transparency. That allows other colors or background patterns to show through slightly (FIGURE 13-7).

FIGURE 13-7. Headings with various levels of transparency using RGBa values.

BROWSER SUPPORT NOTE

Internet Explorer versions 8 and earlier do not support RGBa color, so if a significant percentage of your users have those browsers, you may want to provide a fallback. Pick an RGB color that approximates the look you're going for and list it first in the style rule. IE ignores the RGBa value, and supporting browsers will override the opaque color when they get to the second declaration.

```
h1 {
  color: rgb(120, 120, 120);
  color: rgba(0, 0, 0, .5);
}
```

HSL Color

CSS3 introduced the ability to specify colors by their HSL values: Hue (color), Saturation, and Lightness (or Luminosity). In this system, the colors are spread out around a circle in the order of the rainbow, with red at the top (12 o'clock) position. Hue values are then measured in degrees around the circle: red at 0°/360°, green at 120°, and blue at 240°, with other colors in between. Saturation is a percentage value from 0% (gray) to 100% (color at full blast). Lightness (or brightness) is also a percentage value from 0% (darkest) to 100% (lightest).

BROWSER SUPPORT NOTE

HSL and HSLa color are not supported in Internet Explorer versions 8 and earlier, so use a fallback if you must support those browsers.

FIGURE 13-8 shows one hue, cyan (located at 180° on the wheel) with its associated saturation and lightness levels. You can see why some people find this system more intuitive to use, because once you lock into a hue, it is easy to make it stronger, darker, or lighter by increasing or decreasing the percentage values. RGB values are not intuitive at all, although some practiced designers develop a feel for them.

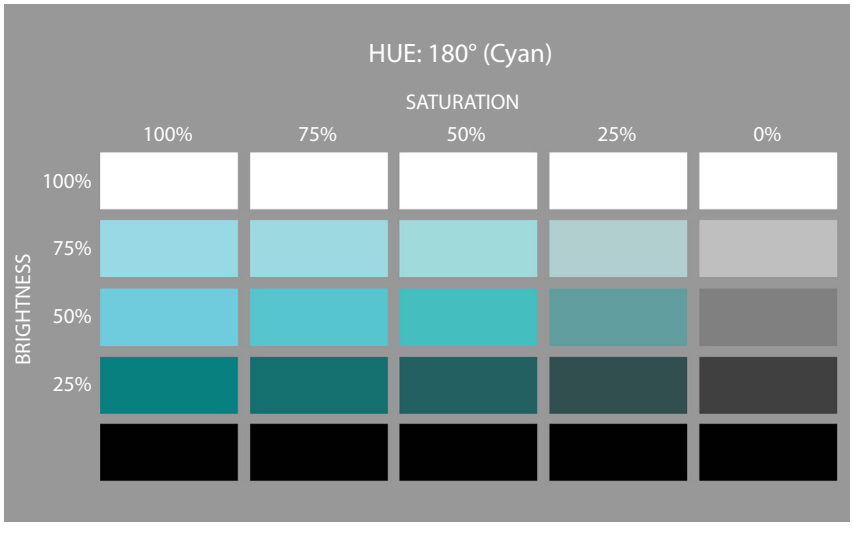


FIGURE 13-8. One hue in the HSL color model, with its associated saturation and lightness values.

In CSS, HSL values are provided as the hue value and two percentages. They are never converted to hexadecimal values, as may be done for RGB. Here is that lavender from FIGURE 13-3 as it would be specified using HSL:

```
color: hsl(265, 51%, 80%);
```

Picking HSL color

There are a number of HSL color pickers online. In the Google color picker, click "Show color values" below the panel to reveal the HSL values for your selected color. Here are some other cool tools worth checking out:

- A Most Excellent HSL Color Picker by Brandon Mathis (*hslpicker.com/*)
- HSL Color Picker (*www.workwithcolor.com/hsl-color-picker-01.htm*)
- HSLa Explorer by Chris Coyier at CSS-Tricks (*css-tricks.com/examples/HSLaExplorer/*)

WARNING

Be aware that the HSB color model listed in Photoshop's color picker is not *the same as HSL and cannot be used for CSS.*

HSLa color

As with RGB, you can add an alpha channel to set the transparency of HSL colors, resulting in the HSLa color model. As for RGBa, the fourth value is the degree of transparency on a scale from 0 (fully transparent) to 1 (fully opaque). This example specifies a spring green color that is 65% opaque:

```
color: hsla(70, 60%, 58%, .65);
```

Summing Up Color Values

It took us a few pages to get here, but the process for picking and specifying colors in style sheets is actually easy:

- Pick one of the predefined color names,

or

- Use a color picker to select a color and copy down the RGB values (preferably the six-digit hex values). Put those values in the style rule using one of the four RGB value formats, and you're done. Or you could use HSL, if that feels easier to you.

There is one more colorful way to fill an element, and that's gradients (colors that fade from one hue to another), but I'm going to save them for the end of this chapter.

FOREGROUND COLOR

Now that we know how to write color values, let's get to the color-related properties. You can specify the foreground and background colors for any HTML element. There are also **`border-color`** properties that take color values, but we'll get to those in **Chapter 14, Thinking Inside the Box.**

The foreground of an element consists of its text and border (if one is specified).

The foreground of an element consists of its text and border (if one is specified). You specify a foreground color with the **`color`** property, as we saw in the last chapter when we rolled it out to give text a little pizzazz. Here are the details for the **`color`** property one more time.

`color`

Values:	*color value (name or numeric)*
Default:	depends on the browser and user's preferences
Applies to:	all elements
Inherits:	yes

In the following example, the foreground of a **`blockquote`** element is set to green with a color name. You can see that applying the **`color`** property to the **`blockquote`** element means the color is inherited by the **`p`** and **`em`** elements it

contains (FIGURE 13-9). The thick dashed border around the whole blockquote is green as well; however, if we were to apply a **border-color** property to this same element, that color would override the green foreground setting.

THE STYLE RULE

```
blockquote {
  border: 4px dashed;
  color: green;
}
```

THE MARKUP

```
<blockquote>
In the latitude of central New England, cabbages are not secure ...
</blockquote>
```

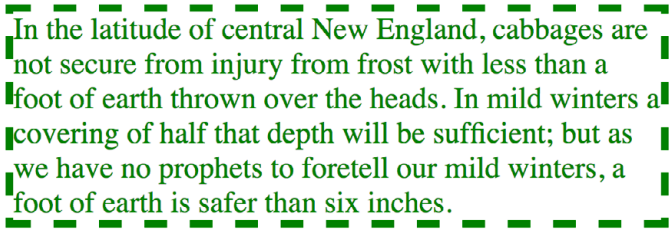


FIGURE 13-9. Applying a color to the foreground of an element.

BACKGROUND COLOR

Use **background-color** to apply a background color to any element.

background-color

Values: *color value (name or numeric)* | transparent

Default: transparent

Applies to: all elements

Inherits: no

A background color fills the canvas behind the element that includes the content area, and any padding (extra space) added around the content, extending behind the border out to its outer edge. Let's see what happens when we use the **background-color** property to make the background of the same sample **blockquote** light green (FIGURE 13-10):

```
blockquote {
  border: 4px dashed;
  color: green;
  background-color: #c6de89;
}
```

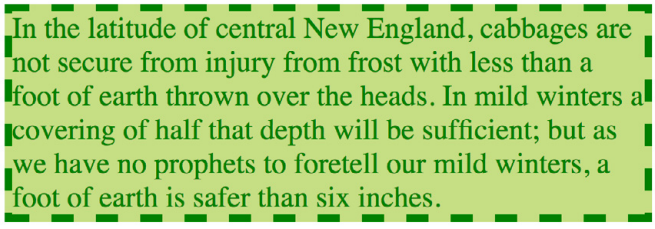


FIGURE 13-10. Adding a light green background color to the sample blockquote.

As expected, the background color fills the area behind the text, all the way to the border. Look closely at the gaps in the border, and you'll see that the background color goes to its outer edge. But that's where the background stops; if we apply a margin around this element, the background will not extend into the margin. We'll revisit all these components of an element when we talk about the CSS box model. For now, just know that, by default, if your border has gaps, the background will show through.

It's worth noting that background colors do not inherit, but because the default background setting for all elements is **transparent**, the parent's background color shows through its descendant elements. For example, you can change the background color of a whole page by applying the **background-color** property to the **body** element and the color will show through all the elements on the page (see **"An Important Exception"**).

In addition to setting the color of the whole page, you can change the background color of any element, both block-level (like the **blockquote** shown in the previous example) as well as inline. In this example, I've used the **color** and **background-color** properties to highlight a word marked up as a "glossary" term. You can see in FIGURE 13-11 that the background color fills the little box created by the inline **dfn** element.

AN IMPORTANT EXCEPTION

When you apply a background to the **body** (or more generically, on the root **html**) element, it is treated specially. It doesn't get clipped to the box, but instead extends to cover the entire viewport.

To color the background of the whole page, apply the background-color property to the body element.

THE STYLE RULE

```
.glossary {
  color: #0378a9;  /* blue */
  background-color: yellow;
}
```

THE MARKUP

```
<p>Every variety of cabbage had their origin in the wild cabbage of
Europe (<dfn class="glossary"><i>Brassica oleracea</i></dfn>)</p>
```

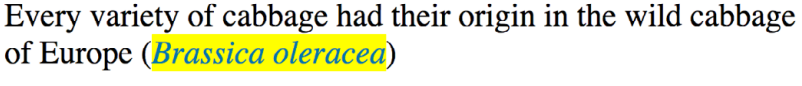


FIGURE 13-11. Applying the background-color property to an inline element.

CLIPPING THE BACKGROUND

Traditionally, the background painting area (the area on which fill colors are applied) of an element extends all the way out to the outer edge of the border, as we saw in FIGURE 13-10. CSS3 introduced the **background-clip** property to give designers more control over where the painting area begins and ends.

background-clip

Values: border-box | padding-box | content-box

Default: border-box

Applies to: all elements

Inherits: no

The default **border-box** value draws the painting area to the outside edge of the border, as we've seen. FIGURE 13-12 shows that **padding-box** starts the painting area on the outside edge of the padding area for the element (and to the inside edge of the border). Finally, **content-box** allows the background to fill only the content area for the element.

I can't help but feel like I'm spoiling the surprise of the element box model and its properties here a little, since I was saving that for the next chapter. I've added some padding (space between the content and the border) so the effects of the clip settings will be more apparent.

```
blockquote {
  padding: 1em; border: 4px dashed; color: green; background-color: #C6DE89;}
}
```

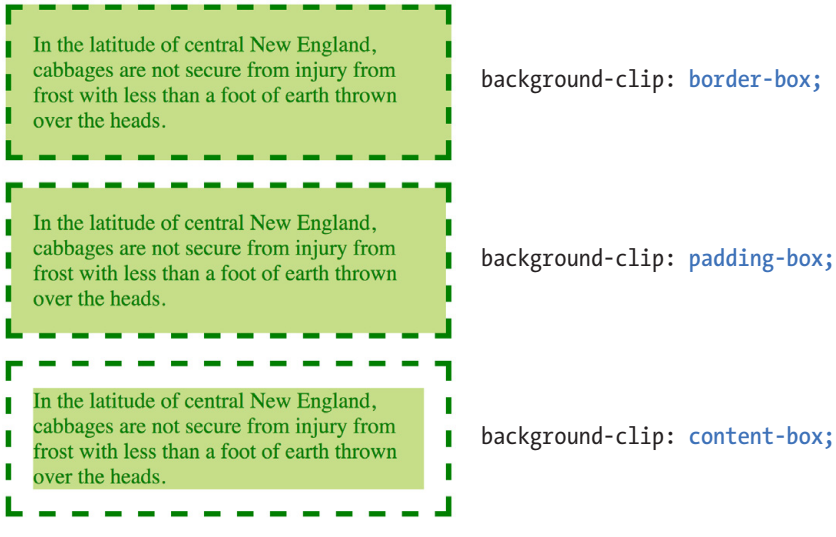


FIGURE 13-12. The **background-clip** property.

■ DESIGN TIP

Using Color

Here are a few quick tips related to working with color:

- Limit the number of colors you use on a page. Nothing creates visual chaos faster than too many colors. I tend to choose one dominant color and one highlight color. I may also use a couple of shades of each, but I resist adding too many different hues.
- When specifying a foreground and background color, make sure that there is adequate contrast. People tend to prefer reading dark text on very light backgrounds online.
- Keep color-blind users in mind when selecting colors. Chris Coyier's article "Accessibility Basics: Testing Your Page for Color Blindness" (*css-tricks.com/accessibility-basics-testing-your-page-for-color-blindness/*) is a good place to start researching strategies for color-blind-friendly design.

Color contributes to both the aesthetics and usability of a site, so it is important to get it right. Geri Coady's book ***Color Accessibility Workflows*** (A Book Apart) provides many best practices.

PLAYING WITH OPACITY

Earlier, we talked about the RGBa color format, which adds a level of transparency when it is applied to a color or background. There is another way to make an element slightly see-through, however—the CSS3 **opacity** property.

opacity

Values: *number* (0 to 1)

Default: 1

Applies to: all elements

Inherits: no

The value for **opacity** is a number between 0 (completely transparent) and 1 (completely opaque). A value of .5 gives the element an opacity of 50%. The **opacity** setting applies to the entire element—both the foreground and the background (if one has been set). If you want to affect just one or the other, use an RGBa color value instead.

The opacity setting applies to the entire element—both the foreground and the background.

In the following code example (and FIGURE 13-13), a heading has been given a color of gold and a background color of white. When the **opacity** property is set, it allows the blue background of the page to show through both the text and the element box.

```
h1 {color: gold; background: white; opacity: .25;}
h1 {color: gold; background: white; opacity: .5;}
h1 {color: gold; background: white; opacity: 1;}
```

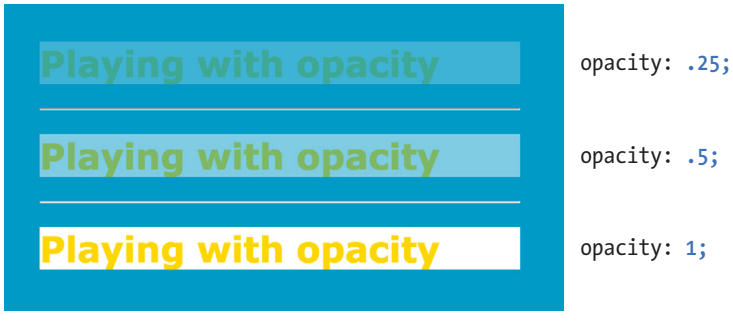


FIGURE 13-13. Setting the opacity on an element affects both the foreground and background colors.

You may be itching to take these color and background properties out for a spin, and we will in a moment, but first, I want to introduce you to some of the fancier CSS selectors and round out your collection. The **"At a Glance"** sidebar lists the selectors you should feel comfortable with so far.

BROWSER SUPPORT NOTE

The **opacity** *property is not supported in Internet Explorer versions 8 and earlier. If you need to support IE8, use a style rule with Microsoft's proprietary* **filter** *property, then override it with the standard opacity style rule.*

```
h1 {
  filter:alpha(opacity=50);
  opacity: .5;
}
```

PSEUDO-CLASS SELECTORS

■ AT A GLANCE

Selector Review

Here is a quick summary of the selector types we've covered already ("E" stands for "Element"):

Element type selector
```
E {property: value;}
```
Grouped selectors
```
E1, E2, E3 {property: value;}
```
Descendant selector
```
E1 E2 {property: value;}
```
Child selector
```
E1 > E2 {property: value;}
```
Next-sibling selector
```
E1 + E2 {property: value;}
```
Subsequent-sibling selector
```
E1 ~ E2 {property: value;}
```
ID selector
```
E#id {property: value;}
#id {property: value;}
```
Class selector
```
E.class {property: value;}
.class {property: value;}
```
Universal selector
```
* {property: value;}
```

Have you ever noticed that a link is often one color when you click it and another color when you go back to that page? That's because, behind the scenes, your browser is keeping track of which links have been clicked (or "visited," to use the lingo). The browser keeps track of other states too, such as whether the user's cursor is over an element (hover state), whether an element is the first of its type, whether it's the first or last child of its parent, and whether a form element has been checked or disabled, just to name a few.

In CSS, you can apply styles to elements in these states by using a special kind of selector called a pseudo-class selector. It's an odd name, but you can think of it as though elements in a certain state belong to the same class. However, the class name isn't in the markup—it's something the browser just keeps track of. So it's *kinda* like a class...it's a *pseudo-class*.

Pseudo-class selectors are indicated by the colon (`:`) character. They typically go immediately after an element name—for example, `li:first-child`.

There are quite a few pseudo-classes in CSS3, and the W3C has been going a little crazy in the CSS Selector Module Level 4 slinging around new pseudo-classes, the majority of which have no browser support as of this writing. In this section, I'll introduce you to the most commonly used and the best supported as a solid starter kit. You can explore the cutting-edge selectors as you gain more experience. The full list of CSS selectors (including Level 4), with descriptions, can be found in **Appendix C**.

Link Pseudo-Classes

The most basic pseudo-class selectors target links (`a` elements) based on whether they have been clicked. Link pseudo-classes are a type of dynamic pseudo-class because they are applied as the result of the user interacting with the page rather than something in the markup.

`:link`	Applies a style to unclicked (unvisited) links
`:visited`	Applies a style to links that have already been clicked

■ USABILITY TIP

When you alter the appearance of links and visited links, be sure that they still look like links.

By default, browsers typically display linked text as blue and links that have been clicked as purple, but you can change that with a few style rules. There are limitations on what properties may be applied to `:visited` links, as explained in the **"Visited Links and Security"** sidebar.

In these examples, I've changed the color of unclicked links to maroon and visited links to gray. It is common for visited links to be a more muted color than unclicked links:

```
a:link {
  color: maroon;
}
a:visited {
  color: gray;
}
```

User Action Pseudo-Classes

Another type of dynamic pseudo-class targets states that result from direct user actions.

`:focus`	Applies when the element is selected and ready for input
`:hover`	Applies when the mouse pointer is over the element
`:active`	Applies when the element (such as a link or button) is in the process of being clicked or tapped

Focus state

If you've ever used a web form, then you should be familiar with how a browser visually emphasizes a form element when you select it. When an element is highlighted and ready for input, it is said to have "focus." The **`:focus`** selector lets you apply custom styles to elements when they are in the focused state.

In this example, when a user selects a text input, it gets a yellow background color to make it stand out from the other form inputs:

```
input:focus { background-color: yellow; }
```

Hover state

The **`:hover`** selector is an interesting one. It targets elements while the user's mouse pointer is directly over them. You can use the hover state with any element, although it is most commonly used with links to give the user visual feedback that an action is possible. Hover states are also used to trigger pop-up menus for navigation or for revealing more information about an object on the page.

This rule gives links a light pink background color while the mouse hovers over them:

```
a:hover {
  color: maroon;
  background-color: #ffd9d9;
}
```

In the previous chapter, we saw the **`text-decoration`** property used to turn off underlines under links. You could use the **`:hover`** selector to make the underlines appear only "on hover":

```
a:hover {
  text-decoration: underline;
}
```

It is important to note that there is no true hover state on touch-screen devices such as smartphones and tablets, so hover effects must be used with care and alternative solutions (see the sidebar **"Hover on Touch Devices"**).

Visited Links and Security

Browsers keep track of what links have been visited, but for some users, a record of their visited links (which could be stolen by a malicious site) may be undesirable. For people in regions with severe restrictions on viewing online content, that record in the wrong hands could even be life threatening. When it was determined that visual styles applied to visited links, as well as the methods browsers use to keep track of them, could be used to track users' viewing histories, some changes were made to how visited links are handled.

The first change was to limit the visual presentation properties that can be applied to visited links. Style rules with **`:visited`** pseudo-class selectors may use only the following properties: **`color`**, **`background-color`**, **`border-color`** (and individual side border properties), and **`outline-color`**. Any other property will be ignored. Furthermore, you cannot use any value that makes the link transparent, including the **`transparent`** keyword and RGBa and HSLa color values.

Under the hood, the DOM mechanism that keeps track of what links have been visited will always return a "not visited" state, even when visited styles are displayed on the screen. This keeps browsing history hidden at the DOM level as well.

The fate of the **`:visited`** pseudo-class is uncertain, so do not apply styles that are critical to the usability of your site.

Active state

Finally, the **:active** selector applies styles to an element while it is in the process of being activated. In the case of a link, it is the style that is applied while it is being clicked or while a fingertip is in contact with it on a touch screen. This style may be displayed only for an instant, but it can give a subtle indication that something has happened. In this example, I've brightened up the color for the active state (from maroon to red):

```
a:active {
  color: red;
  background-color: #ffd9d9;
}
```

Putting It All Together

Web designers commonly provide styles for all of these link states because it is an easy way to give a nice bit of feedback at every stage of clicking a link (and it usually improves on the browser defaults). In fact, users have come to expect this feedback: seeing at a glance which links have been followed, having links do something when they point at them, and receiving confirmation when the links are successfully clicked.

The required order for pseudo-classes is:
:link
:visited
:focus
:hover
:active

When you apply styles to **a** elements with all five pseudo-classes, the order in which they appear is important for them to function properly. For example, if you put **:link** or **:visited** last, they override the other states, preventing them from appearing. The required order for link pseudo-classes is **:link**, **:visited**, **:focus**, **:hover**, **:active** (LVFHA, which you can remember with LoVe For Hairy Animals, or the mnemonic device of your choice).

Hover on Touch Devices

On the desktop, the mouse pointer can hover over elements on the screen, but touch devices respond only when the screen is actually touched. This can make hover effects problematic on smartphones and tablets.

When hover effects are applied to a link (an **a** element), mobile operating systems may display the hover state styles after a single tap. To follow the link, the user must tap again. Other hover-triggered elements, such as pop-up menus, may get stuck open, requiring the user to tap elsewhere or reload the page to clear it (not a good user experience, and a deal-breaker for some designs).

There is no single CSS-based solution to this issue. Always including **:focus** and **:active** state styles along with the **:hover** styles may help in some situations. Otherwise, your options are to use JavaScript to program the desired effect for mobile devices or to avoid the **:hover** state and stick with outright clicks. It is possible to serve the hover-free styles in a style sheet targeted specifically to touch devices.

JavaScript solutions are beyond the scope of this chapter, so I recommend these resources to get started. Some knowledge of JavaScript is required.

- "4 novel ways to deal with sticky **:hover** effects on mobile devices" (*www.javascriptkit.com/dhtmltutors/sticky-hover-issue-solutions.shtml*).
- Search for "hover states on touch devices" on StackOverflow.com and see questions and answers related to this issue. Stack Overflow is a forum where programmers can ask questions and get help from fellow programmers. You'll find a lot of solutions, but also some dead ends.

It is recommended that you provide a **:focus** style for users who use the keyboard to tab through links on a page rather than clicking with a mouse. Applying the same style used for **:hover** is common, although not required.

To sum things up, the link styles I've shown should look like this in the style sheet. FIGURE 13-14 shows the results.

```
a { text-decoration: none; }  /* turns underlines off for all links */
a:link { color: maroon; }
a:visited { color: gray; }
a:focus { color: maroon; background-color: #ffd9d9; }
a:hover { color: maroon; background-color: #ffd9d9; }
a:active { color: red; background-color: #ffd9d9; }
```

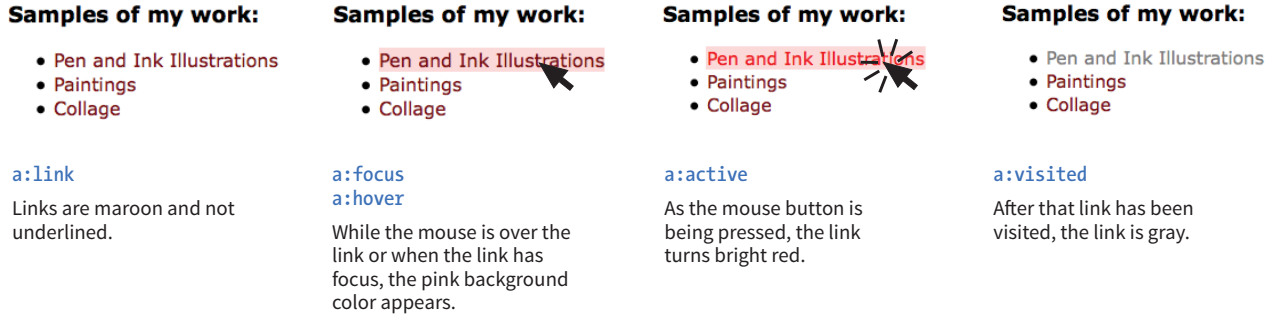


FIGURE 13-14. Changing the colors and backgrounds of links with pseudo-class selectors.

Other Pseudo-Class Selectors

OK...five CSS3 pseudo-classes down, only 40 more to go! Well, I don't know about you, but that sounds like it would take a while, and we have other selector types to explore. However, I do want you to know what is possible today and what is in the works, so I've tucked the CSS3 pseudo-class selectors into the **"More CSS Pseudo-Classes"** sidebar. In addition, you can find the complete list of Level 3 and 4 selectors in **Appendix C, CSS Selectors, Level 3 and 4** with brief descriptions.

I also highly recommend reading "An Ultimate Guide to CSS Pseudo-Classes and Pseudo-Elements" by Ricardo Zea of *Smashing Magazine* (*www.smashingmagazine.com/2016/05/an-ultimate-guide-to-css-pseudo-classes-and-pseudo-elements/*). He's done the hard work of providing explanations and examples of all of the CSS3 pseudo-class selectors in one big roundup.

More CSS3 Pseudo-Classes

The W3C has been creating all sorts of interesting ways to select content for styling based on states the browser keeps track of on the fly.

CSS3 introduced a whole slew of pseudo-classes, most of which are supported by browsers today. Of course, Internet Explorer 8 and earlier lack support, but you could use the Selectivizr polyfill (*selectivizr.com*) to emulate support in the rare event you need to support IE 6–8.

An excellent resource for learning more about these CSS Level 3 and 4 selectors, including browser support information, is *CSS4-selectors.com* by Nelly Brekardin.

Structural pseudo-classes

These allow selection based on where the element is in the structure of the document (the document tree):

`:root`
`:empty`
`:first-child`
`:last-child`
`:only-child`
`:first-of-type`
`:last-of-type`
`:only-of-type`
`:nth-child()`
`:nth-last-child()`
`:nth-of-type()`
`:nth-last-of-type()`

Input pseudo-classes

These selectors apply to states that are typical for form inputs:

`:enabled`
`:disabled`
`:checked`

Location pseudo-classes (in addition to :link and :visited)

`:target` (fragment identifier)

Linguistic pseudo-class

`:lang()`

Logical pseudo-class

`:not()`

PSEUDO-ELEMENT SELECTORS

Pseudo-classes aren't the only kind of pseudo-selectors. There are also four pseudo-elements that act as though they are inserting fictional elements into the document structure for styling. In CSS3, pseudo-elements are indicated by a double colon (`::`) symbol to differentiate them from pseudo-classes. However, all browsers support the single-colon syntax (`:`) as they were defined in CSS2, so many developers stick with that to ensure backward compatibility with older browsers.

First Letter and Line

The following pseudo-elements are used to select the first line or the first letter of text in an element as displayed in the browser.

`::first-line`

This selector applies a style rule to the first line of the specified element. The only properties you can apply, however, are as follows:

`color`
`font` properties
`background` properties
`word-spacing`
`letter-spacing`
`text-decoration`
`vertical-align`
`text-transform`
`line-height`

NOTE

There are a few properties in this list that you haven't seen yet. We'll cover the box-related properties (`margin`*,* `padding`*,* `border`*) in* **Chapter 14, Thinking Inside the Box.** *The* `float` *property is introduced in* **Chapter 15, Floating and Positioning.**

::first-letter

This applies a style rule to the first letter of the specified element. The properties you can apply are limited to the following:

color	vertical-align (if float is none)
font properties	padding properties
background properties	margin properties
letter-spacing	border properties
word-spacing	line-height
text-decoration	float
text-transform	

FIGURE 13-15 shows examples of the **::first-line** and **::first-letter** pseudo-element selectors.

```
p::first-line { letter-spacing: 9px; }
p::first-letter { font-size: 300%; color: orange; }
```

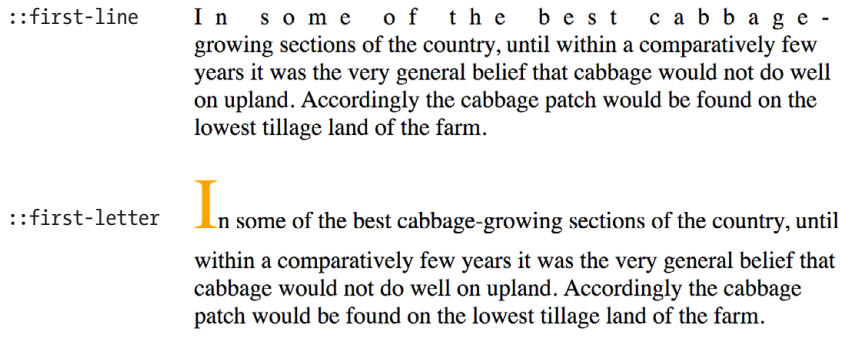


FIGURE 13-15. Examples of **::first-line** and **::first-letter** pseudo-element selectors.

Generated Content with ::before and ::after

You've seen how browsers add bullets and numbers to lists automatically, even though they are not actually in the HTML source. That is an example of generated content, content that browsers insert on the fly. It is possible to tell browsers to generate content before or after any element you like by using the **::before** and **::after** pseudo-elements (see **Note**).

Generated content could be used to add icons before list items, to display URLs next to links when web documents get printed out, to add language-appropriate quotation marks around a quote, and much more. Here's a simple example that inserts an image by using the **url()** function before the paragraph and "Thank you." at the end of the paragraph. Compare the markup to what you see rendered in the browser (FIGURE 13-16).

NOTE

Although double colons are specified in CSS3, you can use single colons for backward compatibility. Browsers are also required to support single colons going forward.

THE STYLES:

```
p.warning::before {
  content: url(exclamation.png);
  margin-right: 6px;
}

p.warning::after {
  content: " Thank you.";
  color: red;
}
```

THE MARKUP:

```
<p class="warning">We are required to warn you that undercooked food is
a health risk.</p>
```

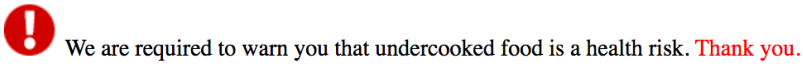


FIGURE 13-16. Generated content added with the **::before** and **::after** pseudo-selectors.

There are a few things of note in this example:

- The pseudo-element selector goes immediately after the target element without any space.
- The pseudo-element rule both inserts the content and specifies how it should be styled in one declaration block.
- The **content** property, which provides the content you want inserted, is required. The selector won't do anything without it.
- If you want spaces between the generated content and the content from the source document, you must include the character spaces inside the value's quotation marks or apply a margin.

If you want to insert an image, such as an icon or other mark, specify the URL without quotations marks:

```
li:before { content: url(images/star.png) }
```

When using generated content, keep in mind that whatever you insert does not become part of the document's DOM. It exists in the browser's display only and is not accessible to assistive devices like screen readers. It is best to use generated content for decorations and other "extras" that are not critical to your meaning and message.

■ FURTHER READING

"Learning to Use the :before and :after Pseudo-Elements in CSS" by Louis Lazaris (*www.smashingmagazine.com/2011/07/learning-to-use-the-before-and-after-pseudo-elements-in-css/*).

ATTRIBUTE SELECTORS

We're finally in the home stretch with selectors. Attribute selectors target elements based on attribute names or values, which provides a lot of flexibility for selecting elements without needing to add a lot of **class** or **id** markup. The CSS3 attribute selectors are listed here:

FUN FACT

Class and ID selectors are just special types of attribute selectors.

element[*attribute*]

The simple attribute selector targets elements with a particular attribute regardless of its value. The following example selects any image that has a **title** attribute.

```
img[title] {border: 3px solid;}
```

element[*attribute*="*exact value*"]

The exact attribute value selector selects elements with a specific value for the attribute. This selector matches images with exactly the **title** value "first grade".

```
img[title="first grade"] {border: 3px solid;}
```

element[*attribute*~="*value*"]

The partial attribute value selector (indicated with a tilde, **~**) allows you to specify one part of an attribute value. The following example looks for the word "grade" in the title, so images with the **title** value "first grade" and "second grade" would be selected.

```
img[title~="grade"] {border: 3px solid;}
```

element[*attribute*|="*value*"]

The hyphen-separated attribute value selector (indicated with a bar, **|**) targets hyphen-separated values. This selector matches any link that points to a document written in a variation on the English language (**en**), whether the attribute value is **en-us** (American English), **en-in** (Indian English), **en-au-tas** (Australian English), and so on.

```
[hreflang|="en"] {border: 3px solid;}
```

element[*attribute*^="*first part of the value*"]

The beginning substring attribute value selector (indicated with a carat, **^**) matches elements whose specified attribute values *start* in the string of characters in the selector. This example applies the style only to images that are found in the */images/icons* directory.

```
img[src^="/images/icons"] {border: 3px solid;}
```

element[*attribute*$="*last part of the value*"]

The ending substring attribute value selector (indicated with a dollar sign, **$**) matches elements whose specified attribute values *end* in the string of characters in the selector. In this example, you can apply a style to just the **a** elements that link to PDF files.

```
a[href$=".pdf"] {border-bottom: 3px solid;}
```

`element[attribute*="any part of the value"]`

The arbitrary substring attribute value selector (indicated with an asterisk, `*`) looks for the provided text string in any part of the attribute value specified. This rule selects any image that contains the word "February" somewhere in its **`title`**.

```
img[title*="February"] {border: 3px solid;}
```

OK, we're done with selectors! You've been a real trouper. I think it's definitely time to try out foreground and background colors as well as a few of these new selector types in EXERCISE 13-1 before moving on to background images.

BACKGROUND IMAGES

We've seen how to add images to the content of the document by using the **`img`** element, but most decorative images are added to pages and elements as backgrounds with CSS. After all, decorations such as tiling background patterns are firmly part of presentation, not structure. We've come a long way from the days when sites were giant graphics cut up and held together with tables (*shudder*).

In this section, we'll look at the collection of properties used to place and push around background images, starting with the basic **`background-image`** property.

Adding a Background Image

The **`background-image`** property adds a background image to any element. Its primary job is to provide the location of the image file.

background-image

Values: `url(`*location of image*`)` | none

Default: none

Applies to: all elements

Inherits: no

The value of **`background-image`** is a sort of URL holder that contains the location of the image (see **Note**).

NOTE

The proper term for that "URL holder" is a functional notation. It is the same syntax used to list decimal and percentage RGB values.

The URL is relative to wherever the CSS rule is at the time. If the rule is in an embedded style sheet (a **`style`** element in the HTML document), then the pathname in the URL should be relative to the location of the HTML file. If the CSS rule is in an external style sheet, then the pathname to the image should be relative to the location of the *.css* file.

As an alternative, providing site root relative URLs for images ensures that the background image can be found regardless of the location of the style rules.

EXERCISE 13-1. Adding color to a document

In this exercise, we'll start with a simple black-and-white menu and give it some personality with foreground and background colors (FIGURE 13-17). You should have enough experience writing style rules by this point that I'm not going to hold your hand as much as I have in previous exercises. This time, you write the rules. You can check your work against the finished style sheet provided with the materials for this chapter.

Open the file *summer-menu.html* (get it at *learningwebdesign.com/5e/materials*) in a text editor. You will find that there is already an embedded style sheet that provides basic text formatting. You'll just need to work on the colors. Feel free to save the document at any step along the way and view your progress in a browser.

1. Make the **h1** heading purple (R:153, G:51, B:153, or **#993399**) by adding a new declaration to the existing **h1** rule. Note that because this value has all double digits, you can use the condensed version (**#939**).
2. Make the **h2** headings light brown (R:204, G:102, B:0, **#cc6600** or **#c60**).
3. Make the background of the entire page a light green (R:210, G:220, B:157, or **#d2dc9d**). Now might be a nice time to save, have a look in a browser, and troubleshoot if the background and headings do not appear in color.
4. Make the background of the **header** white with 50% transparency (R:255, G:255, B:255, .5) so a hint of the background color shows through.
5. I've already added a rule that turns underlines off under links (**text-decoration:none**), so we'll be relying on color to make the links pop. Write a rule that makes links the same purple as the **h1** (**#939**).
6. Make visited links a muted purple (**#937393**).
7. When the mouse is placed over links, make the text a brighter purple (**#c700f2**) and add a white background color (**#fff**). This will look a little like the links are lighting up when the mouse is pointing at it. Use these same style rules for when the links are in focus.
8. As the mouse is being clicked (or tapped on a touch device), add a white background color and make the text turn a vibrant purple (**#ff00ff**). Make sure that all of your link pseudo-classes are in the correct order.

When you are done, your page should look like FIGURE 13-17. We'll be adding background images to this page later, so if you'd like to continue experimenting with different colors on different elements, make a copy of this document and give it a new name. Remember that the Google color picker is an easy destination for colors and their RGB equivalents.

WARNING

Don't forget the **#** *character before hex values. The rule won't work without it.*

FIGURE 13-17. The Black Goose Bistro menu page with colors applied.

The root directory is indicated by a slash at the beginning of the URL. For example:

```
background-image: url(/images/background.jpg);
```

The downside, as for all site root relative URLs, is that you won't be able to test it locally (from your own computer) unless you have it set up as a server.

These examples and FIGURE 13-18 show background images applied behind a whole page (**body**) and a single **blockquote** element with padding and a border applied.

```
body {
  background-image: url(star.png);
}

blockquote {
  background-image: url(dot.png);
  padding: 2em;
  border: 4px dashed;
}
```

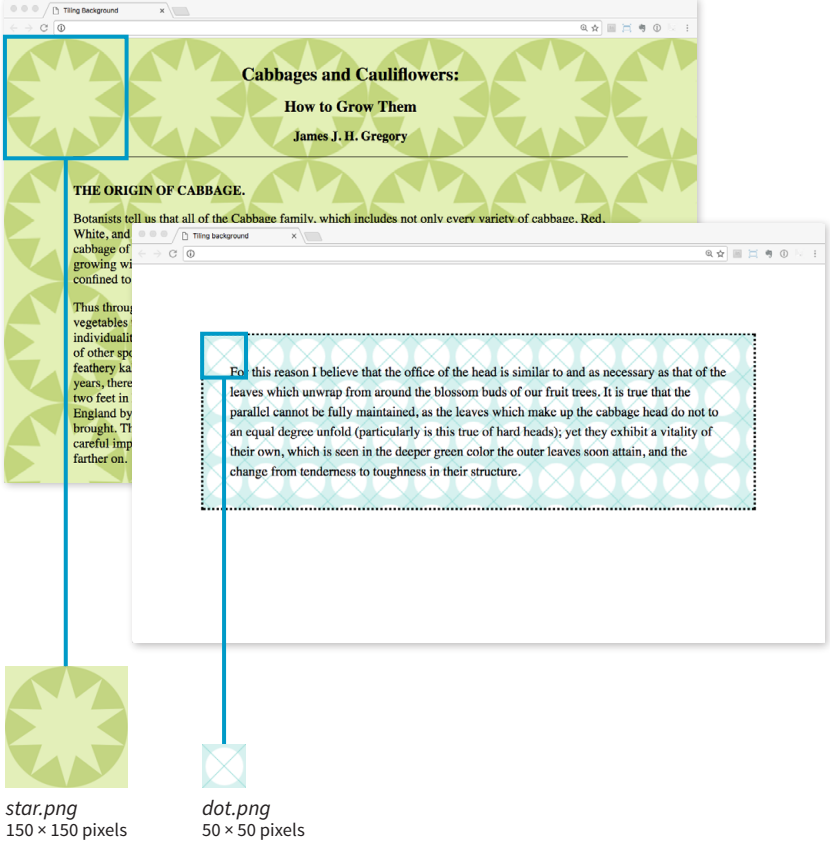


FIGURE 13-18. Tiling background images added with the **background-image** property.

AT A GLANCE

Background Properties

The properties related to the background are:

- background-color
- background-image
- background-repeat
- background-position
- background-attachment
- background-clip
- background-size
- background

DESIGN TIP

Tiling Background Images

When working with background images, keep these guidelines and tips in mind:

- Use a simple image that won't interfere with the legibility of the text over it.
- Always provide a background-color value that matches the primary color of the background image. If the background image fails to display, at least the overall design of the page will be similar. This is particularly important if the text color would be illegible against the browser's default white background.
- As usual for the web, keep the file size of background images as small as possible.

Here you can see the default behavior of **background-image**. The image starts in the top-left corner and tiles horizontally and vertically until the entire element is filled (although you'll learn how to change that in a moment). Like background colors, tiling background images fill the area behind the content area, fill the extra padding space around the content, and extend to the outer edge of the border (if there is one). You can change the background painting area with the **background-clip** property.

If you provide both a **background-color** and a **background-image** to an element, the image is placed on top of the color. In fact, it is recommended that you *do* provide a backup color that is similar in hue, in the event that the image fails to download.

Now you can try your hand at adding a tiling background image to a page in EXERCISE 13-2.

Always specify a similar background color should your background image fail to load.

EXERCISE 13-2. Adding a tiling background image

In this exercise, we're going to add a simple tiling background image to the menu. The images provided for this exercise should be in the *images* directory.

Add a declaration to the **body** style rule that makes the image *bullseye.png* tile in the background of the page. Be sure to include the pathname relative to the style sheet (in this case, the current HTML document).

```
background-image: url(images/bullseye.png);
```

Easy, isn't it? When you save and view the page in the browser, it should look like FIGURE 13-19.

I want to point out that *bullseye.png* is a slightly transparent PNG graphic, so it blends into any background color. Try temporarily changing the **background-color** for the **body** element by adding a second **background-color** declaration lower in the stack so it overrides the previous one. Play around with different colors and notice how the circles blend in. When you are done experimenting, delete the second declaration so the background is green again and you're ready to go for upcoming exercises.

FIGURE 13-19. The menu with a simple tiling background image.

Background Repeating

As we saw in FIGURE 13-18, images tile left and right, up and down, when left to their own devices. You can change this behavior with the **background-repeat** property.

background-repeat

Values: repeat | no-repeat | repeat-x | repeat-y | space | round

Default: repeat

Applies to: all elements

Inherits: no

If you want a background image to appear just once, use the **no-repeat** keyword value:

```
body {
  background-image: url(star.png);
  background-repeat: no-repeat;
}
```

You can also restrict the image to tiling only horizontally (**repeat-x**) or vertically (**repeat-y**), as shown in these examples:

```
body {
  background-image: url(star.png);
  background-repeat: repeat-x;
}
body {
  background-image: url(star.png);
  background-repeat: repeat-y;
}
```

FIGURE 13-20 shows examples of each of these keyword values. Notice that in all the examples, the tiling begins in the top-left corner of the element (or browser window when an image is applied to the **body** element). In the next section, I'll show you how to change that.

The remaining keyword values, **space** and **round**, attempt to fill the available background painting area an even number of times.

When **background-repeat** is set to **space**, the browser calculates how many background images can fit across the width and height of the background area, then adds equal amounts of space between each image. The result is even rows and columns and no clipped images (FIGURE 13-21).

The **round** keyword makes the browser squish the background image horizontally and vertically (not necessarily proportionally) to fit in the background area an even number of times (FIGURE 13-21).

Let's try out some background repeating patterns in EXERCISE 13-3.

BROWSER SUPPORT NOTE

Internet Explorer 8 and earlier do not support the **space** *and* **round** *keywords for* **background-repeat**.

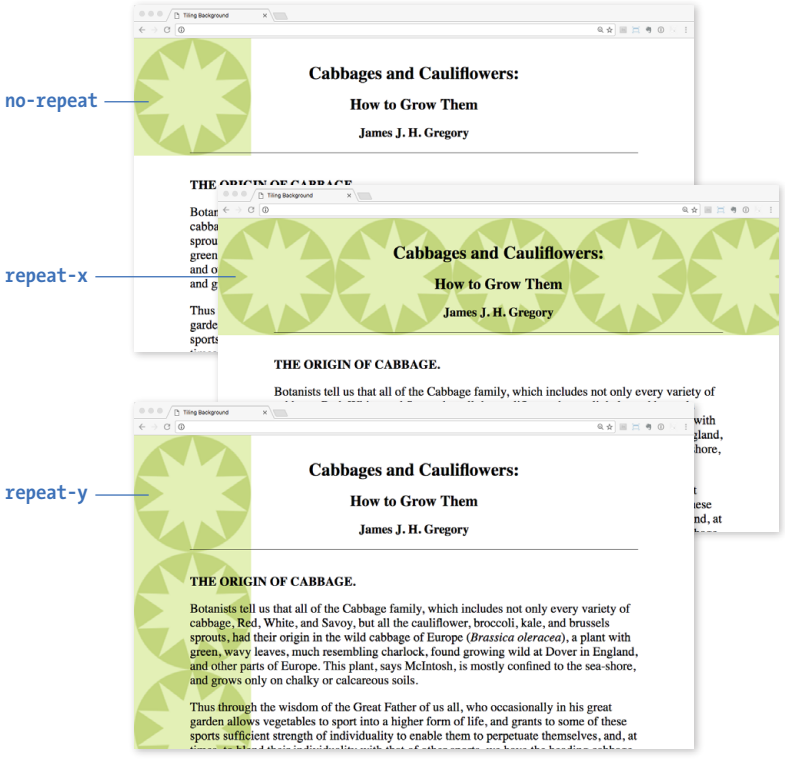


FIGURE 13-20. Turning off automatic tiling with **no-repeat** (top), applying horizontal-axis tiling with **repeat-x** (middle), and applying vertical-axis tiling with **repeat-y** (bottom).

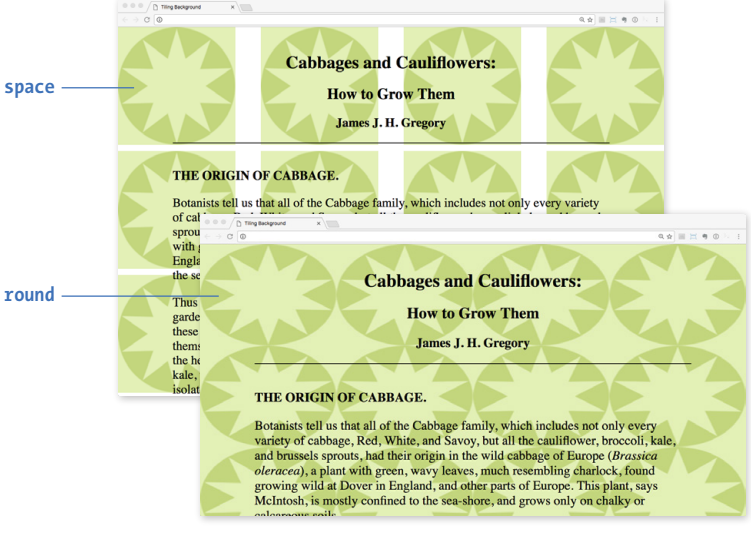


FIGURE 13-21. Examples of **space** and **round** keywords for **background-repeat**. The "space" example would be less clunky if the background color matched the image, but I've left it white to better demonstrate how the **space** value works.

EXERCISE 13-3. Controlling tile direction

Now let's try some slightly more sophisticated tiling on the Summer Menu page. This time we'll add a tiling background just along the top edge of the **header** element.

1. In the **header** rule, add the image *purpledot.png* and set it to repeat horizontally only:

```
header {
  margin-top: 0;
  padding: 3em 1em 2em 1em;
  text-align: center;
  background-color: rgba(255,255,255,.5);
  background-image: url(images/purpledot.png);
  background-repeat: repeat-x;
}
```

2. Save the file and look at it in the browser. It should look something like FIGURE 13-22. I recommend resizing your browser window wider and narrower and paying attention to the position of the background pattern. See how it's always anchored on the left? You're going to learn how to adjust position next. Try changing the style rule to make the dot repeat vertically only; then make it not repeat at all (set it back to **repeat-x** and save when you're done).

FIGURE 13-22. Adding a horizontal tiling image to the **header**.

3. Finally, try out the **space** and **round** repeat values on the **body** background image and see if you like the effect. Note that the tiles are evenly spaced within the body of the document, not just the viewport, so you may see some cut-off circles at the bottom edge of your browser. Delete the **background-repeat** declaration so it goes back to the default **repeat** for upcoming exercises:

```
body {
  …
  background-repeat: space;
}
```

Background Position

The **background-position** property specifies the position of the origin image in the background. You can think of the origin image as the first image that is placed in the background from which tiling images extend. Here is the property and its various values.

background-position

Values: *length measurement* | *percentage* | left | center | right | top | bottom

Default: 0% 0% (same as left top)

Applies to: all elements

Inherits: no

To position the origin image, provide horizontal and vertical values that describe where to place it. There are a variety of ways to do it.

Keyword positioning

The keyword values (**left**, **right**, **top**, **bottom**, and **center**) position the origin image relative to the outer edges of the element's padding. For example, **left** positions the image all the way to the left edge of the background area. The default origin position corresponds to **left top**.

Keywords are typically used in pairs, as in these examples:

```
background-position: left bottom;
background-position: right center;
```

The keywords may appear in any order. If you provide only one keyword, the missing keyword is assumed to be **center**. Thus, **background-position: right** has the same effect as **background-position: right center**.

Length measurements

Specifying position using length measurements such as pixels or ems indicates an amount of offset from the top-left corner of the element to the top-left corner of the background origin image. When you are providing length values, the horizontal measurement always goes first. Specifying negative values is allowed and causes the image to hang outside the visible background area.

When you are providing length or percentage values, the horizontal measurement always goes first.

This example positions the top-left corner of the image 200 pixels from the left edge and 50 pixels down from the top edge of the element (or more specifically, the padding edge by default):

```
background-position: 200px 50px;
```

Percentages

Percentage values are provided in horizontal/vertical pairs, with **0% 0%** corresponding to the top-left corner and **100% 100%** corresponding to the bottom-right corner. As with length values, the horizontal measurement always goes first.

Background Edge Offsets

The CSS3 specification also includes a four-part syntax for **background-position** that allows you to specify an offset (in length or percentage from a particular edge). This is the syntax:

```
background-position:
  edge-keyword offset
  edge-keyword offset;
```

In this example, an origin image is positioned 50 pixels from the right edge and 50 pixels from the bottom of the element's positioning area:

```
background-position:
  right 50px bottom 50px;
```

This four-part syntax is not supported by IE 8 and earlier, Safari and iOS Safari 6 and earlier, and Android 4.3 and earlier.

It is important to note that the percentage value applies to both the canvas area *and* the image itself. A horizontal value of 25% positions the point 25% from the left edge of the image at a point that is 25% from the left edge of the background positioning area. A vertical value of 100% positions the bottom edge of the image at the bottom edge of the positioning area.

```
background-position: 25% 100%;
```

As with keywords, if you provide only one percentage, the other is assumed to be 50% (centered).

FIGURE 13-23 shows the results of each of the aforementioned **background-position** examples with the **background-repeat** set to **no-repeat** for clarity. It is

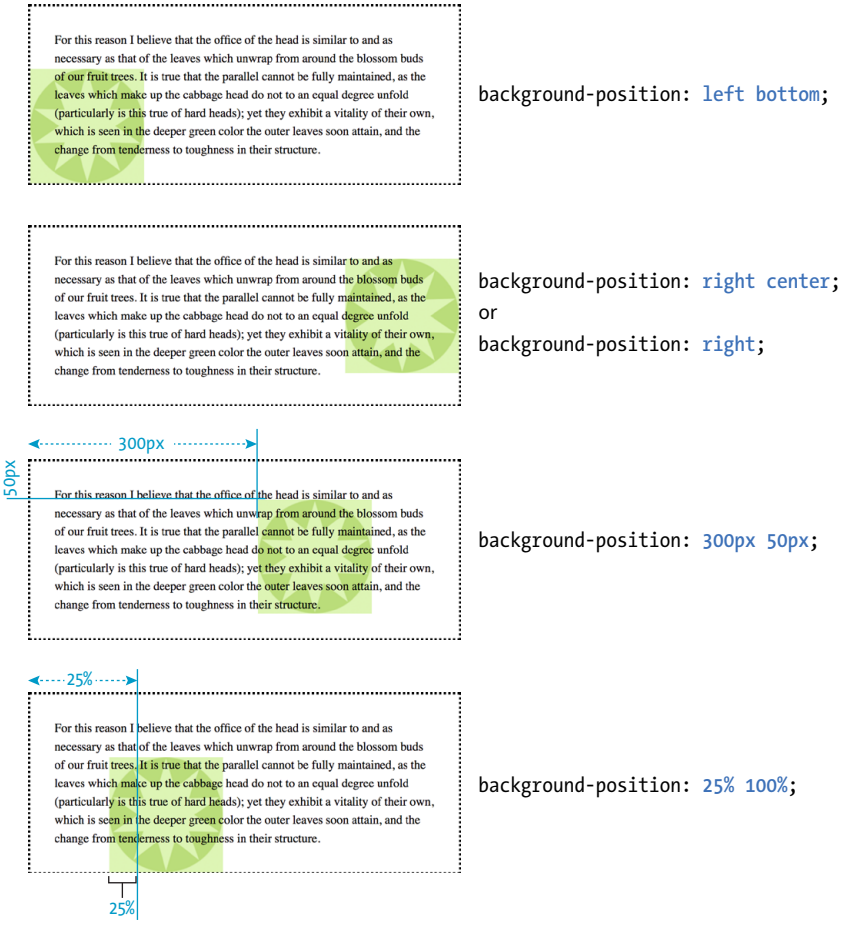


FIGURE 13-23. Positioning a non-repeating background image. If these background images were allowed to repeat, they would extend left and right and/or up and down from the initial positions.

possible to position the origin image and let it tile from there, in both directions or just horizontally or vertically. When the image tiles, the position of the initial image might not be obvious, but you can use **background-position** to make a tile pattern start at a point other than the left edge of the image. This might be used to keep a background pattern centered and symmetrical.

Background Position Origin

Notice in FIGURE 13-23 that when the origin image was placed in the corner of an element, it was placed inside the border (only repeated images extend under the border to its outer edge). This is the default position, but you can change it with the **background-origin** property.

BROWSER SUPPORT NOTE

background-origin *is not supported by Internet Explorer 8 and earlier.*

background-origin

Values: border-box | padding-box | content-box

Default: padding-box

Applies to: all elements

Inherits: no

This property defines the boundaries of the background positioning area in the same way **background-clip** defined the background painting area. You can set the boundaries to the **border-box** (so the origin image is placed under the outer edge of the border, **padding-box** (outer edge of the padding, just inside the border), or **content-box** (the actual content area of the element). These terms will become more meaningful once you get more familiar with the box model in the next chapter. In the meantime, FIGURE 13-24 shows the results of each of the keyword options.

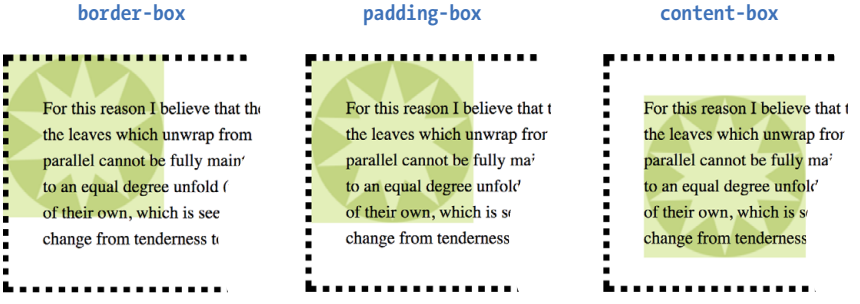


FIGURE 13-24. Examples of **background-origin** keywords.

Before we move on to the remaining background properties, check out EXERCISE 13-4 to get a feel for background positioning.

EXERCISE 13-4. Positioning background images

Let's have some fun with the position of the background image in the menu. First we're going to make some subtle adjustments to the background images that are already there, and then we'll swap them out for a whole different background and play around some more. We are still working with the *summer-menu.html* document, which should have repeating tile patterns in the **body** and **header** elements.

1. I'm thinking that because the main elements of the menu are centered, it would be nice if the background patterns stayed centered, too. Add this declaration to both the **body** and **header** rules; then save and look at it in the browser.

   ```
   background-position: center top;
   ```

   You may not notice the difference until you resize the browser wide and narrow again. Now the pattern is anchored in the center and reveals more or less on both edges, not just the right edge as before.

2. For kicks, alter the **background-position** values so that the purple dots are along the bottom edge of the **header** (**center bottom**). (That doesn't look so good; I'm putting mine back to **top**.) Then try moving *bullseye.png* down 200 pixels (**center 200px**). Notice that the pattern still fills the entire screen—we moved the origin image down, but the background is still set to tile in all directions. FIGURE 13-25 shows the result of these changes.

3. That looks good, but let's get rid of the background on the **body** for now. I want to show you a little trick. During the design process, I prefer to hide styles in comments instead of deleting them entirely. That way, I don't need to remember them or type them in again; I only have to remove the comment indicators, and they're back. When the design is done and it's time to publish, I strip unused styles out to keep the file size down.

   Here's how to hide declarations as CSS comments:

   ```
   body {
       …
       background-color: #d2dc9d;
       /*  background-image: url(images/bullseye.png);
       background-position: center 200px; */
   }
   ```

4. Now, add the *blackgoose.png* image (also a semi-transparent PNG) to the background of the page. Set it to not repeat, and center it at the top of the page:

   ```
   background-image: url(images/blackgoose.png);
   background-repeat: no-repeat;
   background-position: center top;
   ```

   Take a look in the browser window and watch the background scroll up with the content when you scroll the page.

5. I want you to get a feel for the various position keywords and numeric values. Try each of these out and look at it in the browser. Be sure to scroll the page and watch what happens. Note that when you provide a percentage or keyword to the vertical position, it is based on the height of the entire document, not just the browser window. You can try your own variations as well.

   ```
   background-position: right top;

   background-position: right bottom;

   background-position: left 50%;

   background-position: center 100px;
   ```

6. Leave the image positioned at **center 100px** so you are ready to go for the next exercise. Your page should look like the one shown on the right in FIGURE 13-25.

Centered background pattern

Positioned non-repeating image

FIGURE 13-25. The results of positioning the origin image in the tiling background patterns (left) and positioning a single background logo (right).

Background Attachment

In the previous exercise, I asked you to scroll the page and watch what happens to the background image. As expected, it scrolls along with the document and off the top of the browser window, which is its default behavior. However, you can use the **background-attachment** property to free the background from the content and allow it to stay fixed in one position while the rest of the content scrolls.

background-attachment

Values: scroll | fixed | local

Default: scroll

Applies to: all elements

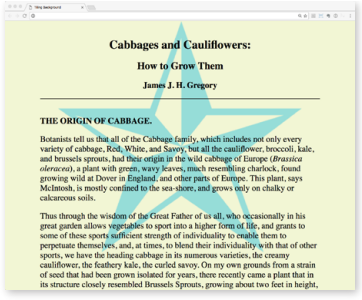
Inherits: no

With the **background-attachment** property, you have the choice of whether the background image scrolls with the content or stays in a fixed position. When an image is **fixed**, it stays in the same position relative to the viewport of the browser (as opposed to being relative to the element it fills). You'll see what I mean in a minute (and you can try it yourself in EXERCISE 13-5).

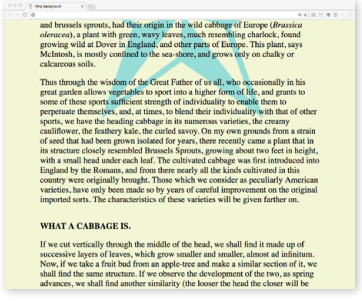
In the following example, a large, non-tiling image is placed in the background of the whole document (the **body** element). By default, when the document scrolls, the image scrolls too, moving up and off the page, as shown in FIGURE 13-26. However, if you set the value of **background-attachment** to **fixed**, it stays where it is initially placed, and the text scrolls up over it.

```
body {
  background-image: url(images/bigstar.gif);
  background-repeat: no-repeat;
  background-position: center 300px;
  background-attachment: fixed;
}
```

The **local** value, which was added in CSS3, is useful when an element has its own scrolling mechanism. Instead of scrolling with the viewport's scroller, **local** makes the background image fixed to the content of the scrolling element. This keyword is not supported in IE8 and earlier and may also be problematic on mobile browsers.

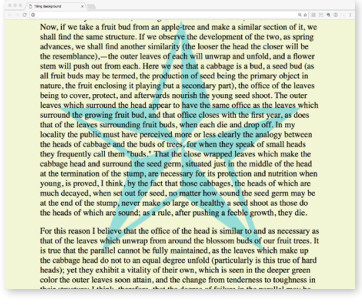


A large non-repeating background image in the **body** of the document.



background-attachment: scroll;

By default, the background image is attached to the **body** element and scrolls off the page when the content scrolls.



background-attachment: fixed;

When **background-attachment** is set to **fixed**, the image stays in its position relative to the browser viewing area and does not scroll with the content.

FIGURE 13-26. Preventing the background image from scrolling with the **background-attachment** property.

EXERCISE 13-5.
Fixed position

When we last left the bistro menu, we had applied a large, non-repeating logo image to the background of the page. We'll leave it just like that, but we'll use the **background-attachment** property to keep it in the same place even when the page scrolls:

```
body {
  background-image: url(images/
blackgoose.png);
  background-repeat: no-repeat;
  background-position: center
100px;
  background-attachment: fixed;
}
```

Save the document, open it in the browser, and try scrolling. The background image stays put in the viewing area of the browser. Cool, huh?

For extra credit, see what happens when you fix the attachment of the dot pattern in the **header**. (Spoiler: it stays in the same place, but only within the **header** itself. When the **header** slides out of view, so does its background.)

Background Size

OK, we have just one more background image property to cover before we wrap it all up with the **background** shorthand property. So far, the background images we've seen are displayed at the actual size of the image itself. You can change the size of the image by using the **background-size** property.

background-size

Values: *length* | *percentage* | auto | cover | contain

Default: auto

Applies to: all elements

Inherits: no

There are several ways to specify the size of the background image. Perhaps the most straightforward is to specify the dimensions in length units such as pixels or ems. As usual, when two values are provided, the first one is used as the horizontal measurement. If you provide just one value, it is used as the horizontal measurement, and the vertical value is set to **auto**.

This example resizes the *target.png* background image, which has an intrinsic size of 300 pixels by 300 pixels (FIGURE 13-27):

```
header {
  background-image: url(images/target.png);
  background-size: 600px 150px;
}
```

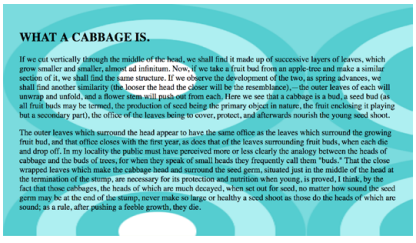
Percentage values are calculated based on the background positioning area, which by default runs to the inside edge of the border, but may have been altered with **background-origin**—something to keep in mind. So a horizontal value of 50% does not make the image half its width; rather, it sizes it to 50% of the width of the positioning area (FIGURE 13-27). Again, the horizontal value goes first. It is OK to mix percentage and length values, as shown in this example:

```
header {
  background-image: url(images/target.png);
  background-size: 50% 10em;
}
```

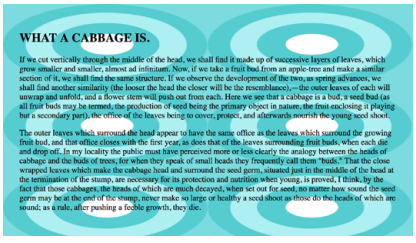
The **auto** keyword resizes the image in whatever direction is necessary to maintain its proportions. Bitmapped images such as GIF, JPEG, and PNG have intrinsic proportions, so they will always stay proportional when one sizing value is set to **auto**. Some images, such as SVG and CSS gradients, don't have intrinsic proportions. In that case, **auto** sets the width or height to 100% of the width or height of the background positioning area.

The **cover** and **contain** keywords are interesting additions in CSS3. When you set the background size to **cover**, the browser resizes a background image large enough to reach all the sides of the background positioning area. There will be only one image because it fills the whole element, and it is likely that

target.png
300 × 300 pixels



`background-size: 600px 300px;`



`background-size: 50% 10em;`

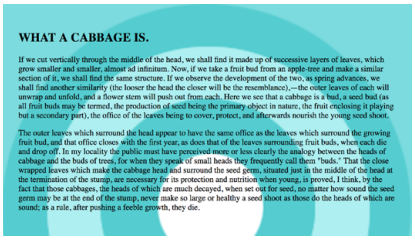
FIGURE 13-27. Resizing a background image with specific length units and percentages.

portions of the image will fall outside the positioning area if the proportions of the image and the positioning area do not match (FIGURE 13-28).

By contrast, **`contain`** sizes the image just large enough to fill either the width or the height of the positioning area (depending on the proportions of the image). The whole image will be visible and "contained" within the background area (FIGURE 13-28). If there is leftover space, the background image repeats unless **`background-repeat`** is set to **`no-repeat`**.

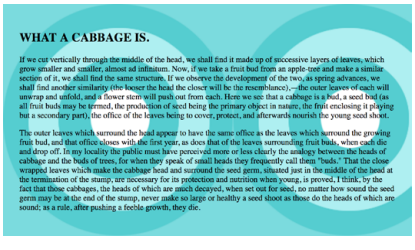
```
div#A {
  background-image: url(target.png);
  background-size: cover; }

div#B {
  background-image: url(target.png);
  background-size: contain; }
```



`background-size: cover;`

The entire background area of the element is covered, and the image maintains its proportions even if it is clipped.



`background-size: contain;`

The image is sized proportionally so it fits entirely in the element. There may be room left over for tiling (as shown).

FIGURE 13-28. Examples of the **`cover`** and **`contain`** background size keywords.

WARNING

When sizing a bitmapped image such as a GIF or PNG larger, you run the risk that it will end up blurry and pixelated. Use background sizing with care.

THE SHORTHAND BACKGROUND PROPERTY

You can use the handy **background** property to specify *all* of your background styles in one declaration.

background

Values: *background-color background-image background-repeat background-attachment background-position background-clip background-origin background-size*

Default: see individual properties

Applies to: all elements

Inherits: no

The value of the **background** property is a list of values that would be provided for the individual background properties previously listed. For example, this one background rule

```
body { background: white url(star.png) no-repeat right top fixed; }
```

replaces this rule with five separate declarations:

```
body {
  background-color: white;
  background-image: url(star.png);
  background-repeat: no-repeat;
  background-position: right top;
  background-attachment: fixed;
}
```

All of the property values for **background** are optional and may appear in any order. The only restriction is that when you are providing the coordinates for the **background-position** property, the horizontal value must appear first, immediately followed by the vertical value. As with any shorthand property, be aware that if any value is omitted, it will be reset to its default value. See the **"Watch Out for Overrides"** sidebar.

In EXERCISE 13-6, you can convert your long-winded background properties to a single declaration with **background**.

Watch Out for Overrides

The **background** property is efficient, but use it carefully. We've addressed this before, but it bears repeating. Because **background** is a shorthand property, when you omit a value, that property will be reset to its default. Be careful that you do not accidentally override style rules earlier in the style sheet with a later shorthand rule that reverts your settings to their defaults.

In this example, the background image *dots.gif* will *not* be applied to **h3** elements because by omitting the value for **background-image**, you essentially set that value to **none**:

```
h1, h2, h3 {
background: red url(dots.gif)
repeat-x;
}
h3 {
background: green;
}
```

To override particular properties, use the specific background property you intend to change. For example, if the intent in the preceding example were to change just the background color of **h3** elements, the **background-color** property would be the correct choice.

EXERCISE 13-6. Convert to shorthand property

This one is easy. Replace all of the background-related declarations in the **body** of the bistro menu with a single **background** property declaration:

```
body {
  font-family: Georgia, serif;
  font-size: 100%;
  line-height: 175%;
  margin: 0 15%;
  background: #d2dc9d url(images/blackgoose.png)
no-repeat center 100px fixed;
}
```

Do the same for the **header** element, and you're done.

Multiple Backgrounds

CSS3 introduced the ability to apply multiple background images to a single element. To apply multiple values for **background-image**, put them in a list separated by commas. Additional background-related property values also go in comma-separated lists; the first value listed applies to the first image, the second value to the second, and so on.

Although CSS declarations usually work on a "last one wins" rule, for multiple background images, whichever is listed last goes on the bottom, and each image prior in the list layers on top of it. You can think of them like Photoshop layers in that they get stacked in the order in which they appear in the list. Put another way, the image defined by the first value will go in front, and others line up behind it, in the order in which they are listed.

```
body {
  background-image: url(image1.png), url(image2.png), url(image3.png);
  background-position: left top, center center, right bottom;
  background-repeat: no-repeat, no-repeat, no-repeat;
  …
}
```

Alternatively, you can take advantage of the **background** shorthand property to make the rule simpler. Now the **background** property has three value series, separated by commas:

```
body {
  background:
    url(image1.png) left top no-repeat,
    url(image2.png) center center no-repeat,
    url(image3.png) right bottom no-repeat;
}
```

FIGURE 13-29 shows the result. The big, orange 1 is positioned in the top-left corner, the 2 is centered vertically and horizontally, and the 3 is in the bottom-right corner. All three background images share the background positioning area of one **body** element. Try it out for yourself in EXERCISE 13-7.

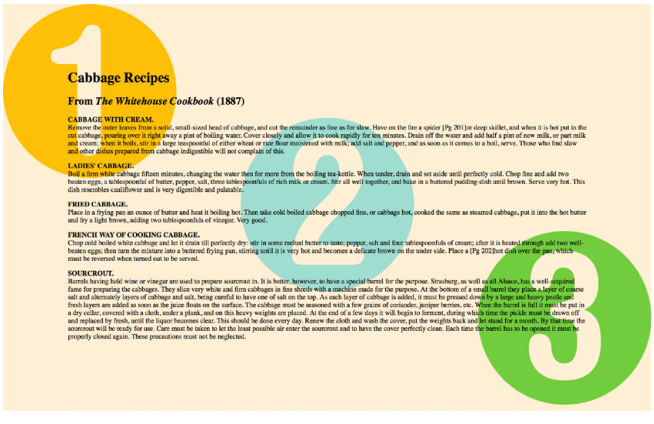


FIGURE 13-29. Three separate background images added to the **body** element.

BROWSER SUPPORT NOTE

Internet Explorer 8 and earlier do not support multiple background images and will entirely ignore any background declaration with more than one value. The fix is to choose one **background-image** *for the element as a fallback for IE and other non-supporting browsers, and then specify the multiple* **background** *rules that override it:*

```
body {
/* for non-supporting browsers */
  background: url(image_fallback.
png) top left no-repeat;
/* multiple backgrounds */
background:
  url(image1.png) left top
no-repeat,
  url(image2.png) center center
no-repeat,
  url(image3.png) right bottom
no-repeat;
/* background color */
background-color: papayawhip;
}
```

EXERCISE 13-7. Multiple background images

In this exercise, we'll give multiple background images a try (be sure you aren't using an old version of IE, or this won't work).

I'd like the dot pattern in the **header** to run along the left and right sides. I also have a little goose silhouette (*gooseshadow.png*) that might look cute walking along the bottom of the header. I'm making this example friendly for non-supporting browsers (IE8 and earlier) by providing a fallback declaration with just one image and separating out the **background-color** declaration so it doesn't get overridden. If IE8 is not a concern, you don't need the fallback.

You can see in the example that we are placing three images in a single header: dots on the left side, dots on the right, and a goose at the bottom.

```
header {
  …
  background: url(images/purpledot.png) center top
repeat-x;
  background:
    url(images/purpledot.png) left top repeat-y,
    url(images/purpledot.png) right top repeat-y,
    url(images/gooseshadow.png) 90% bottom no-repeat;
  background-color: rgba(255,255,255,.5);
}
```

FIGURE 13-30 shows the final result. Meh, I liked it better before, but you get the idea.

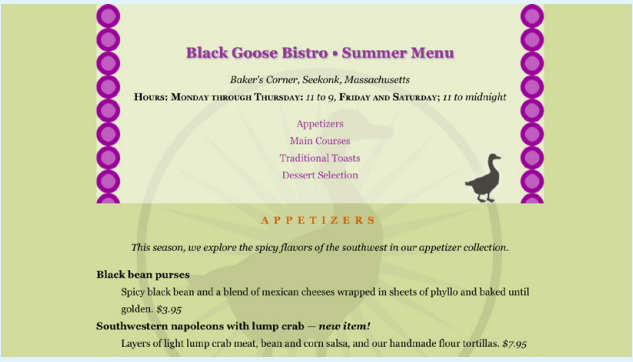


FIGURE 13-30. The bistro menu header with two rows of dots and a small goose graphic in the **header** element.

LIKE A RAINBOW (GRADIENTS)

Gradients are images that browsers generate on the fly. Use them as you would use a background image.

A gradient is a transition from one color to another, sometimes through multiple colors. In the past, the only way to put a gradient on a web page was to create one in an image-editing program and add the resulting image with CSS.

Now we can specify color gradients by using CSS notation alone, leaving the task of rendering color blends to the browser. Although they are specified with code, gradients are *images*. They just happen to be generated on the fly. A gradient image has no intrinsic size or proportions; the size matches the element it gets applied to. Gradients can be applied anywhere an image may be applied: **background-image**, **border-image**, and **list-style-image**. We'll stick with **background-image** examples in this chapter.

There are two types of gradients:

- Linear gradients change colors along a line, from one edge of the element to the other.
- Radial gradients start at a point and spread outward in a circular or elliptical shape.

Linear Gradients

The **linear-gradient()** notation provides the angle of the gradient line and one or more points along that line where the pure color is positioned (color stops). You can use color names or any of the numerical color values discussed earlier in the chapter, including transparency. The angle of the gradient line is specified in degrees (*n***deg**) or with keywords. With degrees, **0deg** points upward, and positive angles go around clockwise so that **90deg** points to the right. Therefore, if you want to go from aqua on the top edge to green on the bottom edge, set the rotation to **180deg**:

```
background-image: linear-gradient(180deg, aqua, green);
```

The keywords describe direction in increments of 90° (**to top**, **to right**, **to bottom**, **to left**). Our **180deg** gradient could also be specified with the **to bottom** keyword. The result is shown in FIGURE 13-31 (top):

```
background-image: linear-gradient(to bottom, aqua, green);
```

You can use the "to" syntax to point to corners as well. The following gradient would be drawn from the bottom-left corner to the top-right corner. The resulting angle of a gradient drawn between corners is determined by the aspect ratio of the box.

```
background-image: linear-gradient(to top right, aqua, green);
```

In the following example, the gradient now goes from left to right (**90deg**) and includes a third color, orange, which appears 25% of the way across the gradient line (FIGURE 13-31, middle). You can see that the placement of the color stop is indicated after the color value. You can use percentages or any length measurement. The first and last color stops don't require positions because they are set to 0% and 100%, respectively, by default.

```
background-image: linear-gradient(90deg, yellow, orange 25%, purple);
```

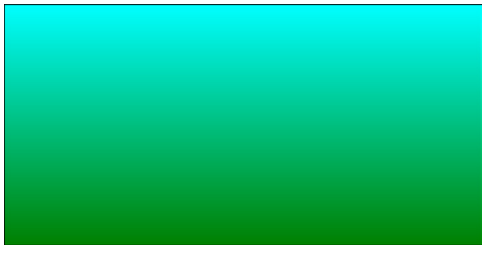
You certainly aren't limited to right angles. Specify any degree you like to make the linear gradient head in that direction. You can also specify as many colors as you like. If no positions are specified, the colors are spaced evenly across the length of the gradient line. If you position the last color stop short of the end of the gradient line (such as the blue at 50% in this example), the last color continues to the end of the gradient line (FIGURE 13-31, bottom):

```
background-image: linear-gradient(54deg, red, orange, yellow, green,
blue 50%);
```

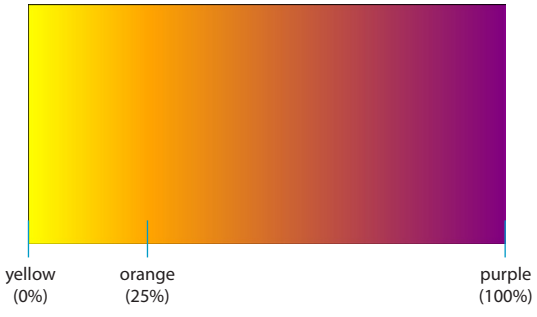
■ PERFORMANCE TIP

Gradients offer both advantages and disadvantages when it comes to performance. On the plus side, they do not require an extra call to the server and require fewer bytes to download than images. On the other hand, all that rendering on the fly requires time and processing power that can hurt performance. Radial gradients are the worst culprits. They can be particularly problematic on mobile devices, where processing power may be limited. Consider serving a separate style sheet without gradients to mobile devices.

```
linear-gradient(180deg, aqua, green);
or
linear-gradient(to bottom, aqua, green);
```



```
linear-gradient(90deg, yellow, orange 25%, purple);
```




```
linear-gradient(54deg, red, orange, yellow, green, blue 50%);
```

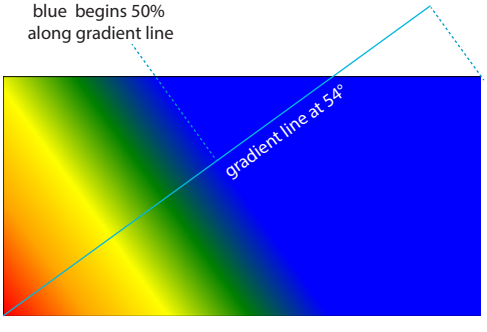


FIGURE 13-31. Examples of linear gradients.

These examples are pretty garish, but if you choose your colors and stops right, gradients are a nice way to give elements subtle shading and a 3-D appearance. The button in FIGURE 13-32 uses a background gradient to achieve a 3-D look without graphics.

FIGURE 13-32. A 3-D button made with only CSS.

```
a.button-like {
  background: linear-gradient(to bottom, #e2e2e2 0%, #dbdbdb 50%,
  #d1d1d1 51%, #fefefe 100%);
}
```

That concludes our quick-and-dirty tour of linear gradients. You should know that I really only scratched the surface of linear gradient behavior and

possibilities, so you may want to check out the resources in the **"Further Reading"** sidebar. It's time to move on to radial gradients.

Radial Gradients

Radial gradients, like the name says, radiate out from a point in a circle along a gradient ray (like a gradient line, but it always points outward from the center). At minimum, a radial gradient requires two color stops, as shown in this example:

```
background-image: radial-gradient(yellow, green);
```

By default, the gradient fills the available background area, and its center is positioned in the center of the element (FIGURE 13-33). The result is an ellipse if the containing element is a rectangle and a circle if the element is square.

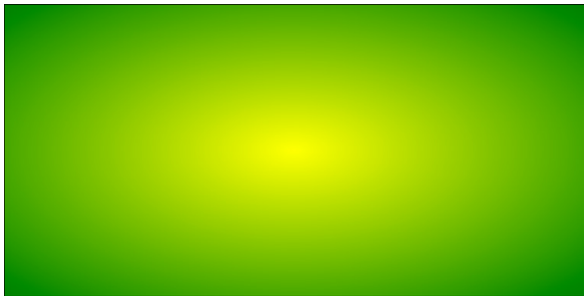


FIGURE 13-33. A minimal radial gradient with default size and position.

That looks pretty spiffy already, but you don't have to settle for the default. The **`radial-gradient()`** notation allows you to specify the shape, size, and center position of the gradient:

Shape

In most cases, the shape of the radial gradient will result from the shape of the element or an explicit size you apply to it, but you can also specify the shape by using the **`circle`** or **`ellipse`** keywords. When you make a gradient a **`circle`** (without conflicting size specifications), it stays circular even when it is in a rectangular element (FIGURE 13-34, top).

```
background-image: radial-gradient(circle, yellow, green);
```

Size

The size of the radial gradient can be specified in length units or percentages, which apply to the gradient ray, or with keywords. If you supply just one length, it is used for both width and height, resulting in a circle. When you provide two lengths, the first one is the horizontal measurement and the second is vertical (FIGURE 13-34, middle). For ellipses, you can provide percentage values as well, or mix percentages with length values.

```
background-image: radial-gradient(200px 80px, aqua, green);
```

FURTHER READING

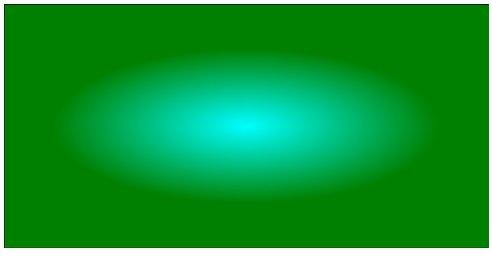
The most in-depth coverage of CSS gradient syntax that I've read is in Eric Meyer's book, ***Colors, Backgrounds, and Gradients*** (O'Reilly). The same content is available in ***CSS: The Definitive Guide***, by Eric Meyer and Estelle Weyl (also from O'Reilly).

Online, I recommend these overviews and tutorials:

- "CSS Gradients" by Chris Coyier (*css-tricks.com/css3-gradients/*)
- "Using CSS Gradients" at MDN Web Docs (*developer.mozilla.org/en-US/docs/Web/CSS/CSS_Images/Using_CSS_gradients*)
- "CSS3 Gradients," part of the ***CSS Mine*** e-book by Martin Michalek (*www.cssmine.com/ebook/css3-gradients*)

```
radial-gradient(circle, yellow, green);
```

```
radial-gradient(200px 80px, aqua, green);
```



```
radial-gradient(farthest-side at right bottom, yellow, orange 50%, purple);
```

FIGURE 13-34. Examples of sizing and positioning radial gradients.

There are also four keywords—**closest-side**, **closest-corner**, **farthest-side**, and **farthest-corner**—that set the length of the gradient ray relative to points on the containing element.

Position

By default, the center of the gradient is positioned at **center center**, but you can change that by using the positioning syntax we covered for the **background-position** property. The syntax is the same, but it should be preceded by the **at** keyword, as in this example (FIGURE 13-34, bottom). Notice that in this example, I have included an additional color stop of orange at the 50% mark.

```
background-image: radial-gradient(farthest-side at right bottom,
yellow, orange 50%, purple);
```

Repeating Gradients

If you'd like your gradient pattern to repeat, use the **`repeating-linear-gradient()`** or **`repeating-radial-gradient()`** notation. The syntax is the same as for single gradients, but adding "repeating-" causes the pattern to repeat the color stops infinitely in both directions. This is commonly used to create interesting striped patterns. In this simple example, a gradient from white to silver (light gray) repeats every 30 pixels because the silver color stop is set to 30px (FIGURE 13-35, top):

```
background: repeating-linear-gradient(to bottom, white, silver 30px);
```

This example makes a diagonal pattern of orange and white stripes (FIGURE 13-35, bottom). The edges are sharp because the white stripe starts at exactly the point where the orange one ends (at 12px) with no fading:

```
background: repeating-linear-gradient(45deg, orange, orange 12px, white
12px, white 24px);
```

`repeating-linear-gradient(to bottom, white, silver 30px);`

`repeating-linear-gradient(45deg, orange, orange 12px, white 12px, white 24px);`

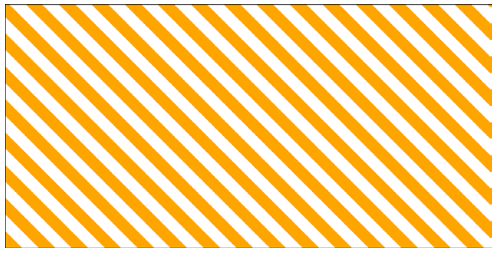


FIGURE 13-35. Repeating gradient pattern.

Browser Support and Vendor Prefixes

All of the major browsers started adding support for the standard gradient syntax between 2012 and 2013 (see **Browser Support Note**), so they've been reliable for a good number of years. However, if you need to support older browsers, you can do so using each browser's proprietary gradient syntax with a vendor prefix (see the **"Vendor Prefixes"** sidebar). For Internet Explorer 9 and earlier, you can use its proprietary **`filter`** function. Or, go the progressive enhancement route and use a solid color as a fallback.

BROWSER SUPPORT NOTE

Standard gradient syntax is supported in Internet Explorer 10+, Edge, Firefox 16+, Chrome 26+, Safari 6.1+, iOS 7.1+, and Android 4.4+.

Vendor Prefixes

Browser makers usually start tinkering with proprietary solutions for cutting-edge web technologies before the specs are fully settled. For many years, they kept their experimentation separate from the final implementation by adding a vendor prefix (or browser prefix) to the property or function name. The prefix indicates that the implementation is proprietary and still a work in progress. For example, while Safari was implementing text-wrap shapes, it used its own **`-webkit-`** prefixed version of the standard **`shape-outside`** property:

```
-webkit-shape-outside: url(cube.png);
```

TABLE 13-1 lists the prefixes used by the major browsers.

TABLE 13-1. Browser vendor prefixes

Prefix	Organization	Most popular browsers
`-ms-`	Microsoft	Internet Explorer
`-moz-`	Mozilla Foundation	Firefox, Camino, SeaMonkey
`-o-`	Opera Software	Opera, Opera Mini, Opera Mobile
`-webkit-`	Originally Apple; now open source	Safari, Chrome, Android, Silk, BlackBerry, WebOS, many others

Vendor prefixes allowed developers to start using cool new CSS features on the browsers that supported them, which was a plus for moving web design and the specification forward. On the downside, the whole system turned out to be complicated and often misused. In the end, the browser makers agreed to put the prefix system to rest and not release any more proprietary properties.

These days, browsers hide experimental features behind "flags" (options you can turn on or off) or in separate technology preview releases that developers can access for testing purposes only. When a feature seems stable, it is made public in the formal browser release. We'll look at methods for testing for individual CSS features in **Chapter 19, More CSS Techniques**.

However, there are a few CSS properties and features that came into vogue during the prefix era that still require prefixes in order to work in older browsers, should you choose to support them. Gradient syntax is one of those features.

Prefixing Tools

Writing all those redundant prefixed properties is a big pain, but fortunately, there are some tools that will generate them for you automatically.

If you use one of the CSS preprocessor syntaxes (like Sass, LESS, or Stylus), you can take advantage of their prefixing "mixins." We'll talk more about preprocessors in **Chapter 19**.

If you write your CSS in the standard syntax, you can run it through a postprocessor like AutoPrefixer when you are done. Autoprefixer parses your styles, then automatically adds prefixes just for the properties and notations that need them. The prefixing happens as part of a "build step" via a build tool like Grunt. For a good overview, see "Autoprefixer: A Postprocessor Dealing with Vendor Prefixes in the Best Possible Way" at CSS-Tricks (*css-tricks.com/autoprefixer/*). I'll talk more about build tools in **Chapter 20, Modern Web Development Tools**.

A gradient for all browsers

The following example shows the yellow-to-green linear gradient written to address every browser, past and present, with the Internet Explorer **filter** equivalent thrown in for good measure. Notice that there are differences in syntax. Where the CSS3 spec uses the **to bottom** keyword, most of the others use **top**. A very old version used by WebKit browsers used **-webkit-gradient** for both linear and radial gradients, but it was quickly replaced with separate functions. Another difference not evident in this example is that in the old syntax, **0deg** pointed to the right edge, not to the top edge as was standardized in CSS3, and the angles increased counterclockwise.

This is a serious chunk of code for a single gradient, and thankfully, we are very close to this no longer being necessary:

```
background: #ffff00; /* Old browsers */
background: -moz-linear-gradient(top, #ffff00 0%, #00ff00 100%);
/* FF3.6+ */
background: -webkit-gradient(linear, left top, left bottom, color-
stop(0%,#ffff00), color-stop(100%,#00ff00));
/* Chrome,Safari4+ */
background: -webkit-linear-gradient(top, #ffff00 0%,#00ff00 100%);
/* Chrome10+,Safari5.1+ */
background: -o-linear-gradient(top, #ffff00 0%,#00ff00 100%);
/* Opera 11.10+ */
background: -ms-linear-gradient(top, #ffff00 0%,#00ff00 100%);
/* IE10+ */
background: linear-gradient(to bottom, #ffff00 0%,#00ff00 100%);
/* W3C Standard */
filter: progid:DXImageTransform.Microsoft.gradient(
startColorstr='#ffff00', endColorstr='#00ff00',GradientType=0 );
/* IE6-9 */
```

In upcoming chapters, whenever a property requires vendor prefixes, I will be sure to note it. Otherwise, you can assume that the standard CSS is all you need.

Designing Gradients

That last code example was a doozy! Vendor prefixes aside, just the task of describing gradients can be daunting. Although it is not impossible to write the code by hand, I recommend you do what I do—use an online gradient tool. One option is the Ultimate CSS Gradient Generator from Colorzilla (*www.colorzilla.com/gradient-editor/*), shown in FIGURE 13-36. Simply enter as many color stops as you'd like, slide the sliders around until you get the look you want, and then copy the code. That's exactly what I did to get the example we just looked at. The CSS Gradient Generator by Virtuosoft is another fine option that also includes support for repeating gradients (*www.virtuosoft.eu/tools/css-gradient-generator/*).

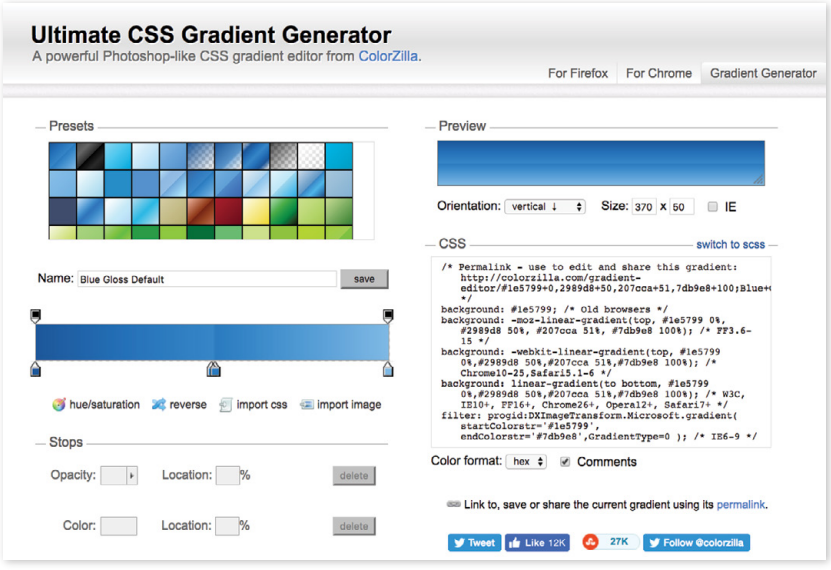


FIGURE 13-36. The Ultimate CSS Gradient Generator (*www.colorzilla.com/gradient-editor*) makes creating CSS gradients a breeze.

FIGURE 13-37. CSS3 Patterns Gallery assembled by Lea Verou (*lea.verou.me/css3patterns*). You may also enjoy Lea's book, ***CSS Secrets: Better Solutions to Everyday Web Design Problems*** (O'Reilly).

If you want your mind blown, take a look at the wild background patterns made with gradients assembled by Lea Verou in her CSS3 Patterns Gallery (*lea.verou.me/css3patterns*) (FIGURE 13-37). It's inspirational, and you can take a peek at the code used to create them.

FINALLY, EXTERNAL STYLE SHEETS

Back in **Chapter 11, Introducing Cascading Style Sheets,** I told you that there are three ways to connect style sheets to an HTML document: inline with the **style** attribute, embedded with the **style** element, and as an external *.css* document linked to or imported into the document. In this section, we finally get to that third option.

External style sheets are by far the most powerful way to use CSS because you can make style changes across an entire site simply by editing a single style sheet document. That is the advantage to having all the style information in one place, and not mixed in with the document source.

Furthermore, because a single style document is downloaded and cached by the browser for the whole site, there is less code to download with every document, resulting in better performance.

First, a little bit about the style sheet document itself. An external style sheet is a plain-text document with at least one style sheet rule. It may *not* include any HTML tags (there's no reason to include them, anyway). It may contain

comments, but they must use the CSS comment syntax that you've seen already:

```
/* This is the end of the section */
```

The style sheet should be named with the *.css* suffix (there are some exceptions to this rule, but you're unlikely to encounter them as a beginner). It may also begin with the **@charset** at-rule to declare the character encoding, although you really need to do that only if you are using an encoding other than UTF-8. If you use **@charset**, it must be the first element in the style sheet, with no characters, including comments or style rules, preceding it.

FIGURE 13-38 shows how a short style sheet document looks in my text editor.

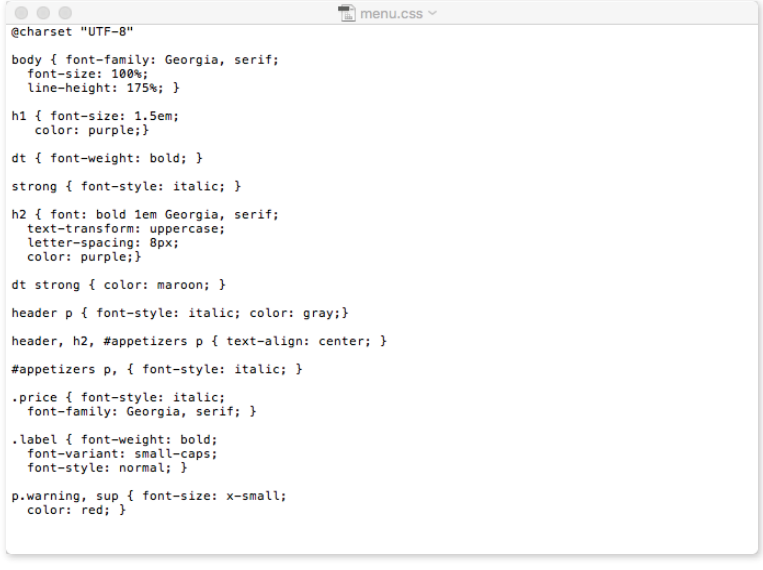


FIGURE 13-38. External style sheets contain only CSS rules and comments in a plain-text document.

There are two ways to apply an external style sheet: the **link** element and an **@import** rule. Let's look at both of these attachment methods.

Using the link Element

The **link** element defines a relationship between the current document and an external resource. By far, its most popular use is to link to style sheets. The **link** element goes in the **head** of the document, as shown here:

```
<head>
  <title>Titles are required.</title>
  <link rel="stylesheet" href="/path/stylesheet.css">
</head>
```

You need to include two attributes in the **link** element:

EXERCISE 13-8. Making an external style sheet

It is OK to use an embedded style sheet while designing a page, but it is probably best moved to an external style sheet once the design is finished so it can be reused by multiple documents in the site. We'll do just that for the summer menu style sheet.

1. Open the latest version of *summer-menu.html*. Select and cut all of the rules within the **style** element, but leave the **<style>...</style>** tags because we'll be using them in a moment.
2. Create a new plain ASCII text document and paste all of the style rules. Make sure that no markup got in there by accident.
3. Save this document as *menustyles.css* in the same directory as the *summer-menu.html* document.
4. Now, back in *summer-menu.html*, add an **@import** rule to attach the external style sheet:

   ```
   <style>
   @import url(menustyles.css);
   </style>
   ```

   Save the file and reload it in the browser. It should look exactly the same as it did when the style sheet was embedded. If not, go back and make sure that everything matches the examples.
5. Delete the whole **style** element, and this time we'll add the style sheet with a **link** element in the **head** of the document.

   ```
   <link rel="stylesheet"
   href="menustyles.css">
   ```

   Again, test your work by saving the document and taking a look at it in the browser.

rel="stylesheet"

Defines the linked document's relation to the current document. The value of the **rel** attribute is always **stylesheet** when you are linking to a style sheet.

href="*url*"

Provides the location of the *.css* file.

You can include multiple **link** elements to different style sheets, and they'll all apply. If there are conflicts, whichever one is listed last will override previous settings, because of the rule order and the cascade.

Importing with @import

The other method for attaching an external style sheet to a document is to import it with an **@import** rule. The **@import** at-rule is another type of rule you can add to a style sheet, either in an external *.css* style sheet document, or right in the **style** element, as shown in the following example:

```
<head>
  <style>
     @import url("/path/stylesheet.css");
     p { font-face: Verdana;}
  </style>
  <title>Titles are required.</title>
</head>
```

In this example, a relative URL is shown, but it could also be an absolute URL (beginning with **http://**). The **@import** rule must go at the beginning of the style sheet *before any selectors*. You can import more than one style sheet, and they all will apply, but rules from the last style sheet listed take precedence over earlier ones.

You can also limit a style sheet's import to specific media types (such as screen, print, or projection, to name a few) or viewing environments (orientation, screen size, etc.) using media queries. Media queries are a method for applying styles based on the medium used to display the document. They appear after the **@import** rule in a comma-separated list. For example, if you have created a style sheet that should be imported and used only when the document is printed, use this rule:

```
@import url(print_styles.css) print;
```

Or to serve a special style sheet just for small devices, you could also query the viewport:

```
@import url(small_device.css) screen and (max-width: 320px;);
```

We'll talk a lot more about media queries in **Chapter 17, Responsive Web Design**, but I mention them here as they are relevant to importing style sheets.

You can try both the **link** and **@import** methods in EXERCISE 13-8.

Using Modular Style Sheets

Because you can compile information from multiple external style sheets, modular style sheets have become a popular technique for style management. Many developers keep styles they frequently reuse—such as typography treatments, layout rules, or form-related styles—in separate style sheets, then combine them in mix-and-match fashion using **`@import`** rules. Again, the **`@import`** rules need to go before rules that use selectors.

Here's an example of a style sheet that imports multiple external style sheets:

```
/* basic typography */
@import url("type.css");

/* form inputs */
@import url("forms.css");

/* navigation */
@import url("list-nav.css");

/* site-specific styles */
body { background: orange; }

/* more style rules */
```

This is a good technique to keep in mind as you build experience in creating sites. You'll find that there are some solutions that work well for you, and it is nice not to have to reinvent the wheel for every new site. Modular style sheets are a good time-saving and organizational device; however, they can be a problem for performance and caching.

If you use this method, it is recommended that you compile all of the styles into a single document before delivering them to a browser. Not to worry, you don't need to do it manually; there are tools out there that will do it for you. The LESS and Sass CSS preprocessors (which will be formally introduced in **Chapter 20**) are just two tools that offer compiling functionality.

NOTE

You can also supply the URL without the **`url()`** *notation:*

```
@import "/path/style.css";
```

Again, absolute pathnames, beginning at the root, will ensure that the .css document will always be found.

WRAPPING IT UP

We've covered a lot of ground (or *background*, to be more accurate) in this chapter. We looked at ways to set the foreground and background colors for an element by using various numeric systems and color names. We looked at options for adjusting the level of transparency with the **`opacity`** property and RGBa, and HSLa color spaces. We spent a long time exploring the various ways to add a background image and adjust how it repeats, where the origin image is placed, and how it is sized. We saw how linear and radial gradients can be used as background images as well. Along the way, you picked up pseudo-class, pseudo-element, and attribute selectors and looked at ways to attach external style sheets. I think that's enough for one chapter! See how much you remember with this little quiz.

TEST YOURSELF

This time I'll test your background prowess entirely with matching and multiple-choice questions. Answers appear in **Appendix A**.

1. Which of these areas gets filled with a background color by default?
   a. The area behind the content
   b. Any padding added around the content
   c. The area under the border
   d. The margin space around the element
   e. All of the above
   f. a and b
   g. a, b, and c

2. Which of these is *not* a way to specify the color white in CSS?
   a. `#FFFFFF`
   b. `#FFF`
   c. `rgb(255, 255, 255)`
   d. `rgb(FF, FF, FF)`
   e. `white`
   f. `rgb(100%, 100%, 100%)`

3. Match the pseudo-class with the elements it targets.

a. `a:link`	1. Links that have already been clicked
b. `a:visited`	2. An element that is highlighted and ready for input
c. `a:hover`	3. An element that is the first child element of its parent
d. `a:active`	4. A link with the mouse pointer over it
e. `:focus`	5. Links that have not yet been visited
f. `:first-child`	6. A link that is in the process of being clicked

4. Match the following rules with their respective samples as shown in FIGURE 13-39. All of the samples in the figure use the same source document, consisting of one paragraph element to which some padding and a border have been applied.

FIGURE 13-39. Samples for Question 4.

a.
```
body {
  background-image: url(graphic.gif);
}
```

b.
```
p {
  background-image: url(graphic.gif);
  background-repeat: no-repeat;
  background-position: 50% 0%;
}
```

c.
```
body {
  background-image: url(graphic.gif);
  background-repeat: repeat-x;
}
```

d.
```
p {
  background: url(graphic.gif) no-repeat right center;
}
```

e.
```
body {
  background-image: url(graphic.gif);
  background-repeat: repeat-y;
  }
```

f.
```
body {
  background: url(graphic.gif) no-repeat right center;
}
```

CSS REVIEW: COLOR AND BACKGROUND PROPERTIES

Here is a summary of the properties covered in this chapter, in alphabetical order.

Property	Description
`background`	Shorthand property that combines background properties
`background-attachment`	Specifies whether the background image scrolls or is fixed
`background-clip`	Specifies how far the background image should extend
`background-color`	Specifies the background color for an element
`background-image`	Provides the location of an image to use as a background
`background-origin`	Determines how the **`background-position`** is calculated (from edge of border, padding, or content box)
`background-position`	Specifies the location of the origin background image
`background-repeat`	Specifies whether and how a background image repeats (tiles)
`background-size`	Specifies the size of the background image
`color`	Specifies the foreground (text and border) color
`opacity`	Specifies the transparency level of the foreground and background

CHAPTER
14

THINKING INSIDE THE BOX

In **Chapter 11, Introducing Cascading Style Sheets,** I described the box model as one of the fundamental concepts of CSS. According to the box model, every element in a document generates a box to which properties such as width, height, padding, borders, and margins can be applied. You probably already have a feel for how element boxes work from adding backgrounds to elements. This chapter covers all the box-related properties, beginning with an overview of the components of an element box, and then taking on the box properties from the inside out: content dimensions, padding, borders, and margins.

IN THIS CHAPTER

THE ELEMENT BOX

As we've seen, every element in a document, both block-level and inline, generates a rectangular element box. The components of an element box are diagrammed in FIGURE 14-1. Pay attention to the new terminology—it will be helpful in keeping things straight later in the chapter.

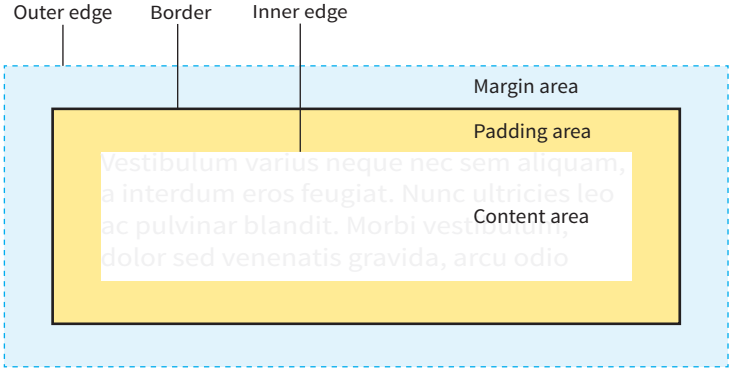


FIGURE 14-1. The parts of an element box according to the CSS box model.

Content area

At the core of the element box is the content itself. In FIGURE 14-1, the content area is indicated by a white box.

Inner edges

The edges of the content area are referred to as the inner edges of the element box. Although the inner edges are made distinct by a color change in FIGURE 14-1, in real pages, the edge of the content area is invisible.

The amount of space taken up by an element on the page includes the content plus the total amount of padding, borders, and margins applied to the element.

Padding

The padding is the area between the content area and an optional border. In the diagram, the padding area is indicated by a yellow-orange color. Padding is optional.

Border

The border is a line (or stylized line) that surrounds the element and its padding. Borders are also optional.

Margin

The margin is an optional amount of space added on the *outside* of the border. In the diagram, the margin is indicated with light-blue shading, but in reality, margins are always transparent, allowing the background of the parent element to show through.

Outer edge

The outside edges of the margin area make up the outer edges of the element box. This is the total area the element takes up on the page, and it includes the width of the content area plus the total amount of padding, border, and margins applied to the element. The outer edge in the diagram is indicated with a dotted line, but in real web pages, the edge of the margin is invisible.

All elements have these box components; however, as you will see, some properties behave differently based on whether the element is block or inline. In fact, we'll see some of those differences right away as we look at box dimensions.

SPECIFYING BOX DIMENSIONS

`width`

Values:	*length* | *percentage* | `auto`
Default:	`auto`
Applies to:	block-level elements and replaced inline elements (such as images)
Inherits:	no

height

Values: *length* | *percentage* | `auto`

Default: `auto`

Applies to: block-level elements and replaced inline elements (such as images)

Inherits: no

box-sizing

Values: `content-box` | `border-box`

Default: `content-box`

Applies to: all elements

Inherits: no

By default, the width and height of a block element are calculated automatically by the browser (thus the default **`auto`** value). The box will be as wide as the browser window or other containing block element, and as tall as necessary to fit the content. However, you can use the **`width`** and **`height`** properties to make the content area of an element a specific width or height.

Unfortunately, setting box dimensions is not as simple as just dropping those properties in your style sheet. You have to know exactly which part of the element box you are sizing.

There are two ways to specify the size of an element. The default method—introduced way back in CSS1—applies the width and height values to the *content box*. That means that the resulting size of the element will be the dimensions you specify *plus* the amount of padding and borders that have been added to the element. The other method—introduced as part of the **`box-sizing`** property in CSS3—applies the width and height values to the *border box*, which includes the content, padding, and border. With this method, the resulting visible element box, including padding and borders, will be exactly the dimensions you specify. We're going to get familiar with both methods in this section.

Regardless of the method you choose, you can specify the width and height only for block-level elements and non-text inline elements such as images. The **`width`** and **`height`** properties do not apply to inline text (non-replaced) elements and are ignored by the browser. In other words, you cannot specify the width and height of an anchor (**`a`**) or **`strong`** element (see **Note**).

Sizing the Content Box

By default (that is, if you do not include a **`box-sizing`** rule in your styles), the **`width`** and **`height`** properties are applied to the content box. That is the way all current browsers interpret width and height values, but you can explicitly specify this behavior by setting **`box-sizing: content-box`**.

BROWSER SUPPORT TIP

The major browsers began supporting the **`box-sizing`** *property in 2011 and 2012. For browsers released prior to that (Chrome <10, Safari <5.1, Safari iOS <5.1, or Android <4.3), there is the prefixed version* **`-webkit-box-sizing`***, but at this point, the prefix is considered no longer necessary. Internet Explorer 6 and 7 do not support* **`box-sizing`** *at all, but they are fairly extinct.*

NOTE

Actually, there is a way to apply **`width`** *and* **`height`** *properties to inline elements such as anchors (*`a`*): by forcing them to behave as block elements with the* **`display`** *property, covered at the end of this chapter.*

In the following example and in FIGURE 14-2, a simple box is given a width of 500 pixels and a height of 150 pixels, with 20 pixels of padding, a 5-pixel border, and a 20-pixel margin all around. In the default content box model, the **width** and **height** values are applied to the *content area only*.

```
p {
  background: #f2f5d5;
  width: 500px;
  height: 150px;
  padding: 20px;
  border: 5px solid gray;
  margin: 20px;
}
```

The resulting width of the *visible* element box ends up being 550 pixels: the content plus 40px padding (20px left and right) and 10px of border (5px left and right).

Visible element box =

5px + 20px + `500px width` + 20px + 5px = 550 pixels

When you throw in 40 pixels of margin, the width of the *entire* element box is 590 pixels. Knowing the resulting size of your elements is critical to getting layouts to behave predictably.

Element box =

20px + 5px + 20px + `500px width` + 20px + 5px + 20px = 590 pixels

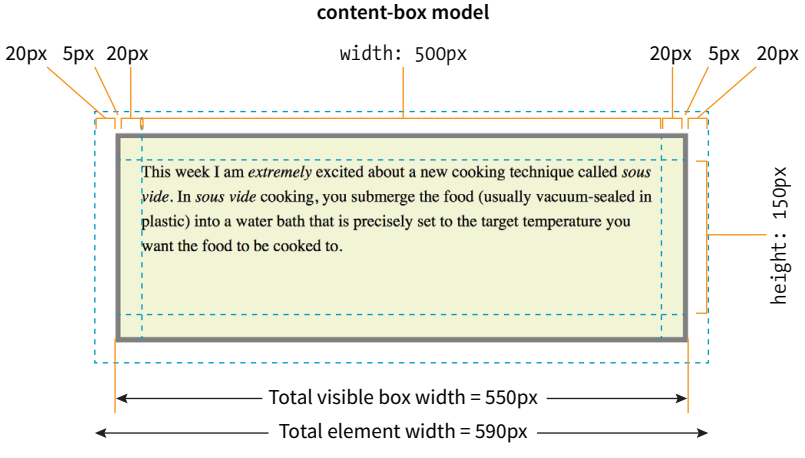


FIGURE 14-2. Specifying the **width** and **height** with the **content-box** model.

Using the border-box Model

The other way to specify the size of an element is to apply width and height dimensions to the entire visible box, including the padding and border. Because this is not the default browser behavior, you need to explicitly set **`box-sizing: border-box`** in the style sheet.

Let's look at the same paragraph example from the previous section and see what happens when we make it 500 pixels using the **border-box** method (FIGURE 14-3). All other style declarations for the box stay the same.

```
p {
  …
  box-sizing: border-box;
  width: 500px;
  height: 150px;
}
```

Now the width of the visible box is 500 pixels (compare to 550 pixels in the content-box model), and the total element widh is 540px. Many developers find the **border-box** model to be a more intuitive way to size elements. It is particularly helpful for specifying widths in percentages, which is a cornerstone of responsive design. For example, you can make two columns 50% wide and know that they will fit next to each other without having to mess around with adding calculated padding and border widths to the mix (although you still need to account for margins).

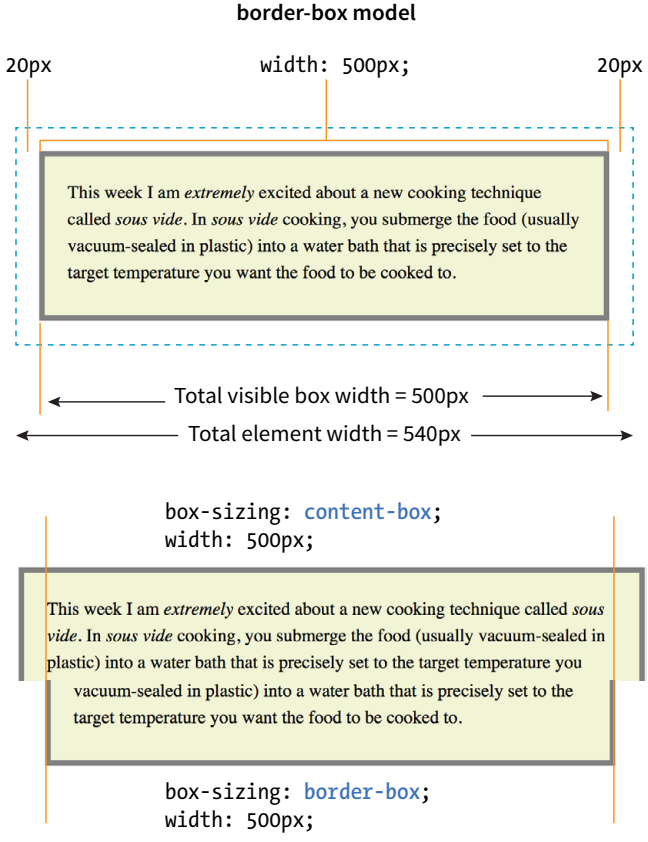


FIGURE 14-3. Sizing an element with the **border-box** method. The bottom diagram compares the resulting boxes from each sizing method.

In fact, many developers simply set *everything* in the document to use the **border-box** model by setting it on the root (**html**) element, then setting all other elements to inherit, like this:

```
html {box-sizing: border-box;}

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

For more information on this technique, read Chris Coyier's article "Inheriting box-sizing Probably Slightly Better Best-Practice" (*css-tricks.com/inheriting-box-sizing-probably-slightly-better-best-practice*).

Maximum and Minimum Dimensions

If you want to set a limit on the size of a block element, use the **max-** and **min-** width and height properties.

max-height, **max-width**, **min-height**, **min-width**

Values: *length* | *percentage* | none

These properties work with block-level and replaced elements (like images) only. When the **content-box** model is used, the value applies to the content area only, so if you apply padding, borders, or margins, it will make the overall element box larger, even if a **max-width** or **max-height** property has been specified. Note also that IE8 does not support **box-sizing** on elements with **max-**/**min-** sizes.

WARNING

Avoid using **max-** *and* **min-** *widths and heights with the* **border-box** *model. They are known to cause browser problems.*

Specifying Height

The **height** property works just the same as **width**. In general practice, it is less common to specify the height of elements. It is more in keeping with the nature of the medium to allow the height to be calculated automatically, allowing the element box to change based on the font size, user settings, or other factors. If you do specify a height for an element containing text, be sure to also consider what happens should the content not fit. Fortunately, CSS gives you some options, as we'll see in the next section.

Handling Overflow

When an element is sized too small for its contents, you can specify what to do with the content that doesn't fit by using the **overflow** property.

overflow

Values:	visible | hidden | scroll | auto
Default:	visible
Applies to:	block-level elements and replaced inline elements (such as images)
Inherits:	no

FIGURE 14-4 demonstrates the predefined values for **overflow**. In the figure, the various values are applied to an element that is 150 pixels square. The background color makes the edges of the content area apparent.

visible

The default value is **visible**, which allows the content to hang out over the element box so that it all can be seen.

hidden

When **overflow** is set to **hidden**, the content that does not fit gets clipped off and does not appear beyond the edges of the element's content area.

scroll

When **scroll** is specified, scrollbars are added to the element box to let users scroll through the content. Be aware that they may become

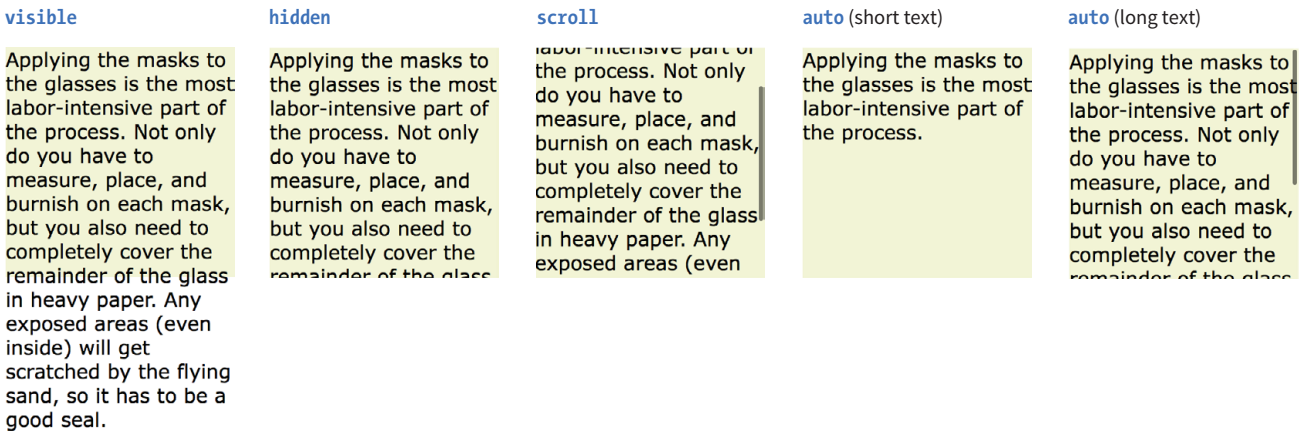


FIGURE 14-4. Options for handling content overflow. The scroll and auto options have narrow gray scrollbars to the right of the text (as rendered on macOS).

visible only when you click the element to scroll it. There is an issue with this value on old iOS (<4), Android (<2.3), and a few other older mobile browsers, so it may be worthwhile to use a simpler alternative to `overflow:scroll` for mobile.

`auto`

The `auto` value allows the browser to decide how to handle overflow. In most cases, scrollbars are added only when the content doesn't fit and they are needed.

PADDING

Padding is the space between the content area and the border (or the place the border would be if one isn't specified). I find it helpful to add padding to elements when using a background color or a border. It gives the content a little breathing room, and prevents the border or edge of the background from bumping right up against the text.

Padding is the space between the content and the border.

You can add padding to the individual sides of any element (block-level or inline). There is also a shorthand `padding` property that lets you add padding on all sides at once.

`padding-top`, `padding-right`, `padding-bottom`, `padding-left`

Values:	*length* | *percentage*
Default:	0
Applies to:	all elements
Inherits:	no

padding

Values: *length* | *percentage*

Default: 0

Applies to: all elements

Inherits: no

The **padding-top**, **padding-right**, **padding-bottom**, and **padding-left** properties specify an amount of padding for each side of an element, as shown in this example and FIGURE 14-5 (note that I've also added a background color to make the outer edges of the padding area apparent).

```
blockquote {
  padding-top: 2em;
  padding-right: 4em;
  padding-bottom: 2em;
  padding-left: 4em;
  background-color: #D098D4; /* light green */
}
```

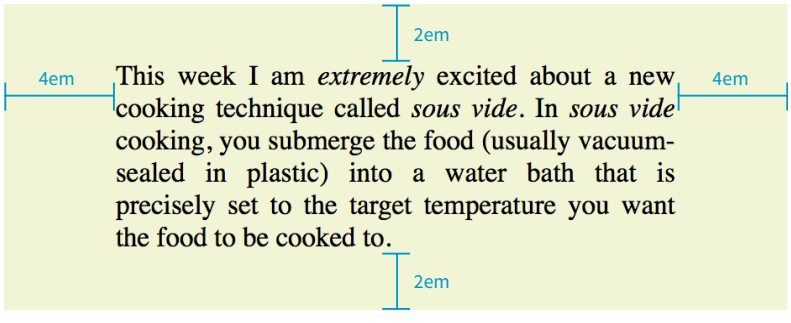


FIGURE 14-5. Adding padding around the content of an element.

Specify padding in any of the CSS length units (**em** and **px** are the most common) or as a percentage of the *width* of the parent element. Yes, the parent's width is used as the basis, even for top and bottom padding. If the width of the parent element changes, so will the padding values on all sides of the child element, which makes percentage values somewhat tricky to manage.

The Shorthand padding Property

As an alternative to setting padding one side at a time, you can use the shorthand **padding** property to add padding all around the element. The syntax is interesting; you can specify four, three, two, or one value for a single **padding** property. Let's see how that works, starting with four values.

When you supply four **padding** values, they are applied to each side in *clockwise* order, starting at the top. Some people use the mnemonic device "TRouBLe" for the order *Top Right Bottom Left*. This is a common syntax for applying shorthand values in CSS, so take a careful look:

```
padding: top right bottom left;
```

Using the **padding** property, we could reproduce the padding specified with the four individual properties in the previous example like this:

```
blockquote {
  padding: 2em 4em 2em 4em;
  background-color: #D098D4;
}
```

If the left and right padding are the same, you can shorten it by supplying only three values. The value for "right" (the second value in the string) will be mirrored and used for "left" as well. It is as though the browser assumes the "left" value is missing, so it just uses the "right" value on both sides. The syntax for three values is as follows:

```
padding: top right/left bottom;
```

This rule would be equivalent to the previous example because the padding on both the left and right edges of the element is set to 4em:

```
blockquote {
  padding: 2em 4em 2em;
  background-color: #D098D4;
}
```

Continuing with this pattern, if you provide only two values, the first one is used for the top and the bottom edges, and the second one is used for the left and right edges:

```
padding: top/bottom right/left;
```

Again, the same effect achieved by the previous two examples could be accomplished with this rule:

```
blockquote {
  padding: 2em 4em;
  background-color: #D098D4;
}
```

Note that all of the previous examples have the same visual effect as shown in FIGURE 14-5.

Finally, if you provide just one value, it will be applied to all four sides of the element. This declaration applies 15 pixels of padding on all sides of a **div** element:

```
div#announcement {
  padding: 15px;
  border: 1px solid;
}
```

Get a feel for adding padding to elements in EXERCISE 14-1.

■ AT A GLANCE

Shorthand Values

1 value

`padding: 10px;`

Applied to all sides.

2 values

`padding: 10px 6px;`

First is top and bottom; second is left and right.

3 values

`padding: 10px 6px 4px;`

First is top; second is left and right; third is bottom.

4 values

`padding: 10px 6px 4px 10px;`

Applied clockwise to top, right, bottom, and left edges consecutively (TRBL).

EXERCISE 14-1. Adding a little padding

In this exercise, we'll begin adding box properties to improve the appearance of a site for the fictional Black Goose Bakery. I've given you a head start by marking up the source (*bakery.html*). Unlike pages in previous exercises, the bakery page uses an external style sheet, *bakery-styles.css*. Everything we will be doing to format this site over the next few chapters happens in the CSS file, so you should never need edit the HTML document; however, that is the file you will open in the browser to see the results of your style changes. All the files are available at *learningwebdesign.com/5e/materials*.

FIGURE 14-6 shows before and after shots of the site. It's going to take several exercises over three chapters to get there, and padding is just the beginning. In **Chapter 16, CSS Layout with Flexbox and Grid**, we'll turn that ugly navigation list into a nice navigation menu bar (in the meantime, please avert your eyes) and give the page a two-column layout suitable for larger screens.

Start by getting familiar with the source document. Open *bakery.html* in a browser and a text editor to see what you've got to work with. The style sheet has been added with an **`@import`** rule in the **`style`** element. The document has been marked up with **`header`** (including a **`nav`** section), **`main`**, **`aside`**, and **`footer`** sections.

Now take a look at *bakery-styles.css* in your text editor. I used comments in the style sheet to organize the styles related to each section (bonus points for you if you keep the styles organized as you go along!). You will find styles for text formatting, colors, and backgrounds—all properties that we've covered so far in this book, so they should look familiar. Now let's add some rules to *bakery-styles.css* to add padding to the elements.

NOTE

This design uses a Google web font called Stint. You will need to have an internet connection in order to see it. If you are working offline, you will see Georgia or some serif font instead, which is just fine for these purposes, but your page won't look exactly like the ones in the figures.

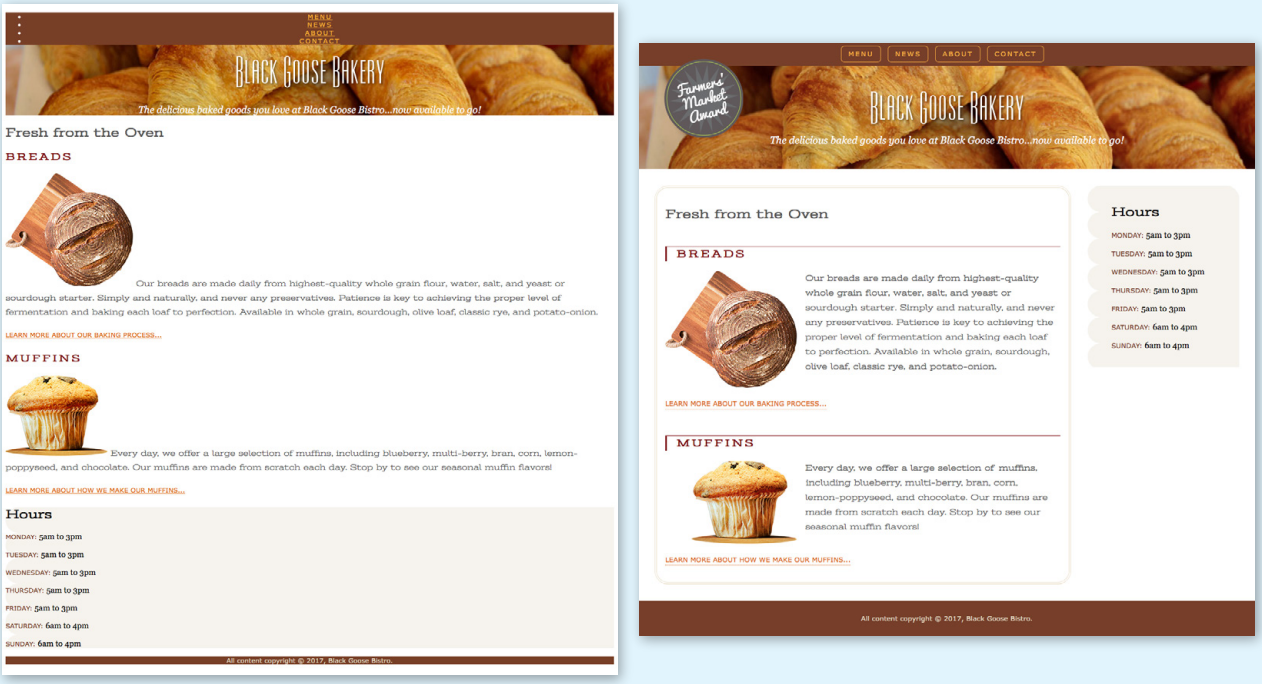


FIGURE 14-6. Before and after shots of the Black Goose Bakery site.

1. The first thing we'll do is to set the **box-sizing** model to **border-box** for all the elements in the document. Add these new rules to the existing **style** element. This will make measurements simpler going forward.

```
html {
  box-sizing: border-box;
}
* {
  box-sizing: inherit;
}
```

2. Now find the styles for the **header** and give it a height. It will fill 100% of the width of the page by default, so the width is taken care of. I picked 15em for the height because it seemed tall enough to accommodate the content and show a nice amount of the croissant background image, but you can play around with it.

```
header {
  …
  height: 15em;
}
```

3. The **main** section is going to need a little padding, so add 1em of padding on all sides. You can add this declaration after the existing **main** styles.

```
main {
  …
  padding: 1em;
}
```

4. Next, we'll get a little fancier with the **aside** element ("Hours"). We'll need extra padding on the left side for the tiling scallop background image to be visible. There are several approaches to applying different padding amounts to each side, but I'm going to do it in a way that gives you experience deliberately overriding earlier declarations.

   Use the **padding** shorthand property to add 1em of padding on all sides of the **aside** element. Then write a second declaration that adds 45 pixels of padding on just the left side. Because the **padding-left** declaration comes second, it will override the 1em setting applied with the shorthand.

```
aside {
  …
  padding: 1em;
  padding-left: 45px;
}
```

5. Finally, that footer is looking skinny and cramped. Let's add padding, which will increase the height of the footer and give the content some space.

```
footer {
  …
  padding: 1em;
}
```

6. Save the *bakery-styles.css* document, and then open (or reload) *bakery.html* in the browser to see the result of your work. The changes at this point are pretty subtle. FIGURE 14-7 highlights the padding additions.

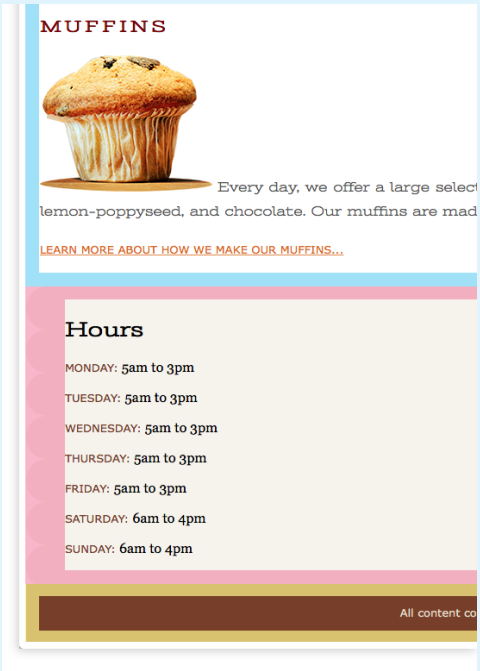


FIGURE 14-7. The shaded areas indicate the padding added to **main** (blue), **aside** (pink), and **footer** (yellow). Colors added for demo purposes but wouldn't render in the browser.

BORDERS

A border is simply a line drawn around the content area and its (optional) padding. You can choose from eight border styles and make them any width and color you like. Borders can be applied all around the element or just on a particular side or sides. CSS3 introduced properties for rounding the corners or applying images to borders. We'll start our border exploration with the various border styles.

Border Style

The style is the most important of the border properties because, according to the CSS specification, if there is no border style specified, the border does not exist (the default is **none**). In other words, you must always declare the style of the border, or the other border properties will be ignored.

Border styles can be applied one side at a time or with the shorthand **border-style** property.

DESIGN TIP

Bottom Borders Instead of Underlines

Turning off link underlines and replacing them with a custom bottom border is a common design technique. It lightens the look of links while still making them stand out from ordinary text.

`border-top-style`, `border-right-style`, `border-bottom-style`, `border-left-style`

Values:	none | solid | hidden | dotted | dashed | double | groove | ridge | inset | outset
Default:	none
Applies to:	all elements
Inherits:	no

`border-style`

Values:	none | solid | hidden | dotted | dashed | double | groove | ridge | inset | outset
Default:	none
Applies to:	all elements
Inherits:	no

The value of the **border-style** property is one of 10 keywords describing the available border styles, as shown in FIGURE 14-8. The value **hidden** is equivalent to **none**.

Use the side-specific border style properties (**border-top-style**, **border-right-style**, **border-bottom-style**, and **border-left-style**) to apply a style to one side of the element. If you do not specify a width, the default medium width will be used. If there is no color specified, the border uses the foreground color of the element (same as the text).

In the following example, I've applied a different style to each side of an element to show the single-side border properties in action (FIGURE 14-9).

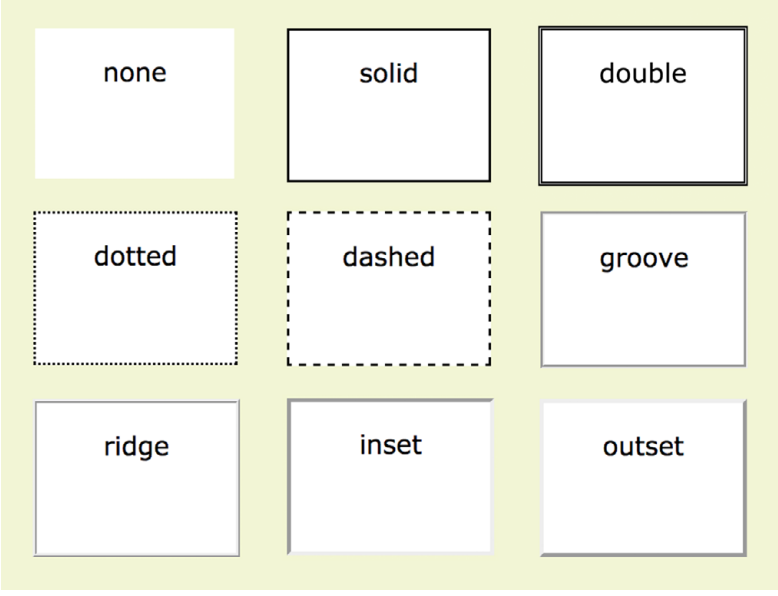


FIGURE 14-8. The available border styles (shown at the default medium width).

```
div#silly {
  border-top-style: solid;
  border-right-style: dashed;
  border-bottom-style: double;
  border-left-style: dotted;
  width: 300px;
  height: 100px;
}
```

The **border-style** shorthand property works on the clockwise (TRouBLe) system described for **padding** earlier. You can supply four values for all four sides or fewer values when the left/right and top/bottom borders are the same. The silly border effect in the previous example could also be specified with the **border-style** property as shown here, and the result would be the same as shown in FIGURE 14-9:

```
border-style: solid dashed double dotted;
```

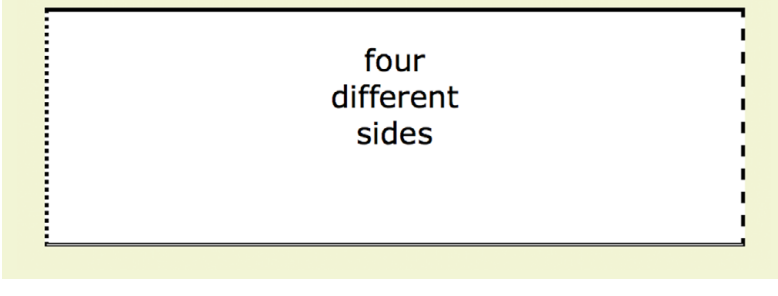


FIGURE 14-9. Border styles applied to individual sides of an element.

Border Width (Thickness)

Use one of the **border-width** properties to specify the thickness of the border. Once again, you can target each side of the element with a single-side property, or specify several sides at once in clockwise order with the shorthand **border-width** property.

`border-top-width`, `border-right-width`, `border-bottom-width`, `border-left-width`

Values: *length* | `thin` | `medium` | `thick`

Default: `medium`

Applies to: all elements

Inherits: no

`border-width`

Values: *length* | `thin` | `medium` | `thick`

Default: `medium`

Applies to: all elements

Inherits: no

The most common way to specify the width of borders is using a pixel or em measurement; however, you can also specify one of the keywords (**thin**, **medium**, or **thick**) and leave the rendering up to the browser.

I've included a mix of values in this example (FIGURE 14-10). Notice that I've also included the **border-style** property because if I didn't, the border would not render at all:

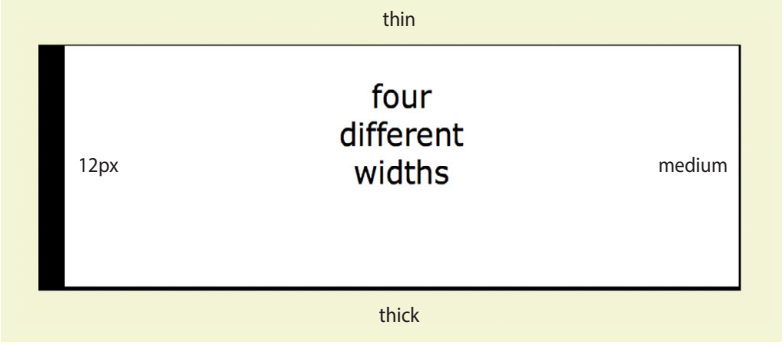


FIGURE 14-10. Specifying the width of borders.

```
div#help {
  border-top-width: thin;
  border-right-width: medium;
  border-bottom-width: thick;
  border-left-width: 12px;
  border-style: solid;
  width: 300px;
  height: 100px;
}
```

or:

```
div#help {
  border-width: thin medium thick 12px;
  border-style: solid;
  width: 300px;
  height: 100px;
}
```

Border Color

Border colors are specified in the same way: via the side-specific properties or the **border-color** shorthand property. When you specify a border color, it overrides the foreground color as set by the **color** property for the element.

border-top-color, border-right-color, border-bottom-color, border-left-color

Values: *color name* or *RGB/HSL value* | transparent

Default: the value of the color property for the element

Applies to: all elements

Inherits: no

border-color

Values: *color name* or *RGB/HSL value* | transparent

Default: the value of the color property for the element

Applies to: all elements

Inherits: no

You know all about specifying color values, and you should be getting used to the shorthand properties as well, so I'll keep this example short and sweet (FIGURE 14-11). Here, I've provided two values for the shorthand **border-color** property to make the top and bottom of a **div** maroon and the left and right sides aqua:

```
div#special {
  border-color: maroon aqua;
  border-style: solid;
  border-width: 6px;
  width: 300px;
  height: 100px;
}
```

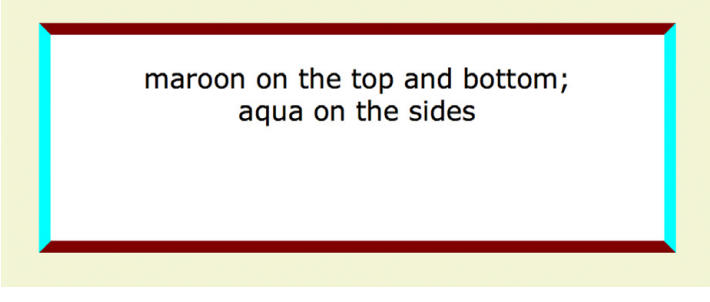


FIGURE 14-11. Specifying the color of borders.

DESIGN TIP

Setting **border-color** to **transparent** allows the background to show through the border, yet holds the width of the border as specified. This may be useful when you're creating rollover (**:hover**) effects with borders, because the space where the border will appear is maintained even when the mouse is not over the element.

Outlines are a good tool for checking your page layout as you work.

CSS Outlines

Another type of rule you can draw around an element is an outline. Outlines look like borders, and the syntax is the same, but there is an important difference. Outlines, unlike borders, are not calculated in the width of the element box. They just lay on top, not interfering with anything. Outlines are drawn on the outside edge of the border (if one is specified) and overlap the margin.

Because outlines do not affect layout, they're a great tool for checking your design. You can turn them on and off without affecting width measurements to see where and how element boxes are positioned.

The outline properties are similar to border properties with one important difference: It is not possible to specify outlines for particular sides of the element box—it's all or nothing.

outline-style

Values: auto | solid | none | dotted | dashed | double | groove | ridge | inset | outset

Default: none

These are the same as the **border-style** values, with the addition of **auto**, which lets the browser choose the style. Also, you cannot set the **outline-style** to **hidden**.

outline-width

Values: *length* | thin | medium | thick

Default: medium

Same as **border-width** values.

outline-color

Values: *color name* or *RGB/HSL value* | invert

Default: invert

The default **invert** value applies the inverse of the background color to the outline, but it has very little browser support.

outline-offset

Values: *length*

Default: 0

By default, the outline is drawn just outside the border edge. **outline-offset** moves the outline beyond the border by a specified length.

outline

Values: *outline-style outline-width outline-color*

Default: Defaults of individual properties

The shorthand **outline** property combines values for **outline-style**, **outline-width**, and **outline-color**. Remember that you can specify them only for all sides of the element at once.

```
div#story { outline: 2px dashed red; }
```

Combining Style, Width, and Color

The authors of CSS didn't skimp when it came to border shortcuts. They also created properties for providing style, width, and color values in one declaration, one side at a time. You can specify the appearance of specific sides, or use the **border** property to change all four sides at once.

border-top, border-right, border-bottom, border-left

Values: *border-style border-width border-color*

Default: defaults for each property

Applies to: all elements

Inherits: no

border

Values: *border-style border-width border-color*

Default: defaults for each property

Applies to: all elements

Inherits: no

The values for **border** and the side-specific border properties may include style, width, and color values in any order. You do not need to declare all three, but if the border style value is omitted, no border will render.

The **border** shorthand property works a bit differently than the other shorthand properties that we've covered in that it takes one set of values and always applies them to all four sides of the element. In other words, it does not use the clockwise TRBL system that we've seen with other shorthand properties.

Here is a smattering of valid border shortcut examples to give you an idea of how they work:

```
h1 { border-left: red .5em solid; }      /* left border only */
h2 { border-bottom: 1px solid; }         /* bottom border only */
p.example { border: 2px dotted #663; }   /* all four sides */
```

Rounded Corners with border-radius

Perhaps you'd like your element boxes to look a little softer and rounder. Well, then, the **border-radius** property is for you! There are individual corner properties as well as a **border-radius** shorthand.

border-top-left-radius, border-top-right-radius, border-bottom-right-radius, border-bottom-left-radius

Values: *length* | *percentage*

Default: 0

Applies to: all elements

Inherits: no

```
p {
  width: 200px;
  height: 100px;
  background: darkorange;
}
```

```
border-radius: 1em;
```

```
border-radius: 50px;
```

```
border-top-right-radius: 50px;
```

```
border-top-left-radius: 1em;
border-top-right-radius: 2em;
border-bottom-right-radius: 1em;
border-bottom-left: 2em;
```

FIGURE 14-12. Make the corners of element boxes rounded with the **border-radius** properties.

border-radius

Values: *1, 2, 3, or 4 length or percentage values*

Default: 0

Applies to: all elements

Inherits: no

To round off the corner of an element, simply apply one of the **border-radius** properties, but keep in mind that you will see the result only if the element has a border or background color. Values are typically provided in ems or pixels. Percentages are allowed and are nice for keeping the curve proportional to the box should it resize, but you may run into some browser inconsistencies.

You can target the corners individually or use the shorthand **border-radius** property. If you provide one value for **border-radius**, it is applied to all four corners. Four values are applied clockwise, starting in the top-left corner (top-left, top-right, bottom-right, bottom-left). When you supply two values, the first one is used for top-left and bottom-right, and the second is for the other two corners.

Compare the **border-radius** values to the resulting boxes in FIGURE 14-12. You can achieve many different effects, from slightly softened corners to a long capsule shape, depending on how you set the values.

BROWSER SUPPORT NOTE

All browsers have been supporting **border-radius** *properties using the standard syntax (that is, without prefixes) since about 2010. There are prefixed properties for Firefox <3.6 and Safari <5.0, but they're so old it's probably not worth worrying about. Internet Explorer 8 and earlier, however, do not support* **border-radius** *at all. But in this case, chances are the usability of your site doesn't depend on rounded corners, so this is a good opportunity to practice progressive enhancement: non-supporting browsers get perfectly acceptable square corners, and all modern browsers get a little something extra.*

Elliptical corners

So far, the corners we've made are sections of perfect circles, but you can also make a corner elliptical by specifying two values: the first for the horizontal radius and the second for the vertical radius (see FIGURE 14-13, Ⓐ and Ⓑ).

Ⓐ
```
border-top-right-radius: 100px 50px;
```

Ⓑ
```
border-top-right-radius: 50px 20px;
border-top-left-radius: 50px 20px;
```

If you want to use the shorthand property, the horizontal and vertical radii get separated by a slash (otherwise, they'd be confused for different corner values). The following example sets the horizontal radius on all corners to 60px and the vertical radius to 40px (FIGURE 14-13, Ⓒ):

Ⓒ
```
border-radius: 60px / 40px;
```

If you want to see something really nutty, take a look at a **border-radius** shorthand property that specifies a different ellipse for each of the four corners. All of the horizontal values are lined up on the left of the slash in clockwise order (top-left, top-right, bottom-right, bottom-left), and all of the corresponding vertical values are lined up on the right (FIGURE 14-13, D):

```
D border-radius: 36px 40px 60px 20px / 12px 10px 30px 36px;
```

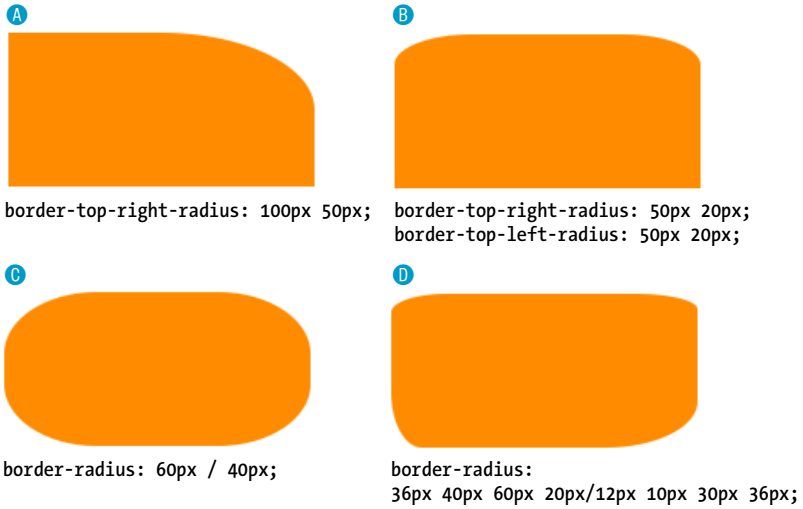


FIGURE 14-13. Applying elliptical corners to boxes.

Now it's time to try your hand at borders. EXERCISE 14-2 will not only give you some practice, but it should also give you some ideas on the ways borders can be used to add visual interest to designs.

EXERCISE 14-2. Border tricks

In this exercise, we'll have some fun with borders on the Black Goose Bakery page. In addition to putting borders around content sections of the page, we'll use borders to beef up the headlines and as an alternative to underlines under links.

1. Open *bakery-styles.css* in a text editor if it isn't already. We'll start with the basics by using the shorthand **border** property to add a tan double rule around the **main** element. Add the new declaration to the existing rule for **main**:

```
main {
  …
  padding: 1em;
  border: double 4px #EADDC4;
}
```

2. Now try out some **border-radius** properties to add generous rounded corners to the **main** and **aside** sections. A 25-pixel radius should do. Pixels are my choice over ems here because I don't want the radius to scale with the text. Start by adding this declaration to the styles for **main**:

```
  border-radius: 25px;
```

And give just the top-right corner of the **aside** a matching rounded corner:

```
aside {
  …
  border-top-right-radius: 25px;
}
```

EXERCISE 14-2. Continued

3. Just for fun (and practice), we'll add a decorative border on two sides of the baked goods headings (**h3**). Find the existing rule for **h3** elements and add a declaration that adds a 1-pixel solid rule on the top of the headline. Add another that adds a thicker 3-pixel solid rule on the left. I want the borders to be the same color as the text, so we don't need to specify the **border-color**. Finally, to prevent the text from bumping into the left border, add a little bit of padding (1em) to the left of the headline content:

```
h3 {
  …
  border-top: 1px solid;
  border-left: 3px solid;
  padding-left: 1em;
}
```

4. The last thing we'll do is to replace the standard underline with a decorative bottom border under links. Start by turning off the underline for all links. Add this rule in the "link styles" section of the style sheet:

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

Then add a 1-pixel dotted border to the bottom edge of links:

```
a {
  text-decoration: none;
  border-bottom: 1px dotted;
}
```

As is often the case when you add a border to an element, it is a good idea to also add a little padding to keep things from bumping together:

```
a {
  text-decoration: none;
  border-bottom: 1px dotted;
  padding-bottom: .2em;
}
```

Now you can save the style sheet and reload *bakery.html* in the browser. FIGURE 14-14 shows a detail of how your page should be looking so far.

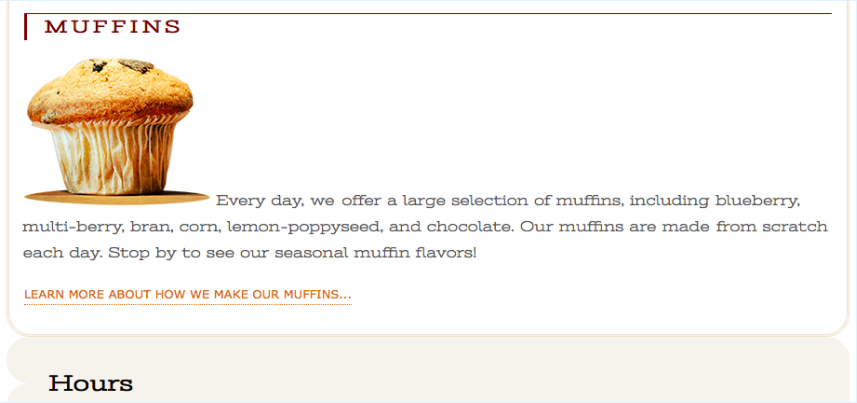


FIGURE 14-14. The results of our border additions.

Picture-Perfect Borders

CSS3 introduced the **`border-image-*`** properties, which let you fill in the sides and corners of a border box with an image of your choice, as shown in FIGURE 14-15.

Border images are applied with a collection of five properties:

- **`border-image-source`** indicates the location of the image
- **`border-image-slice`** divides the image into nine sections using offset measurements
- **`border-image-width`** specifies the width of the border area
- **`border-image-repeat`** specifies whether the image should stretch or repeat along the sides
- **`border-image-outset`** pushes the border away from the content by the specified amount

There is also a shorthand **`border-image`** property that combines the individual properties in the following syntax:

```
border-image: source slice / width / outset
repeat;
```

The style rules for the image border in FIGURE 14-15 are as follows:

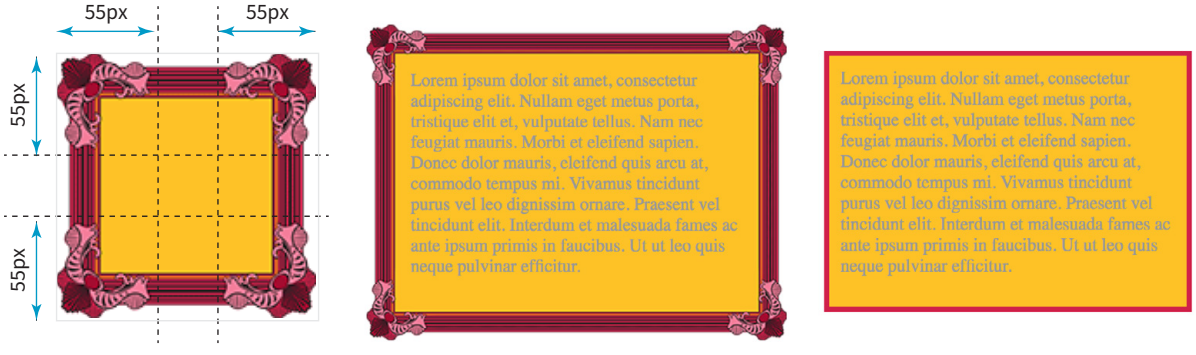
```
border: 5px solid #d1214a; /* red */
border-image: url(fancyframe.png) 55 fill / 55px /
25px stretch;
```

The **`border`** shorthand provides a fallback style for the border should the image not load or if the **`border-image`** isn't supported by the browser.

The **`border-image`** rule tells the browser to apply the image *fancyframe.png* to the border, slice it **`55`** pixels from the edges, and use the center of the image to **`fill`** the center of the box. The width of the border area is **`55px`**, and the image should be pushed toward the margins by **`25px`**. Finally, the image areas that make up the sides should **`stretch`** to fill the width and height of the box.

That's not much of an explanation, I know, but I've written an article, **"Border Images,"** which goes into more detail. You can download it at *learningwebdesign.com/articles/*. For even more information on border images, check out these resources:

- The CSS Background and Borders Module Level 3 (*www.w3.org/TR/css3-background/#the-border-image-source*)
- The **`border-image`** listing on CSS-Tricks (*css-tricks.com/almanac/properties/b/border-image/*), for a less dense explanation



fancyframe.png | With border image | Without border image

FIGURE 14-15. Examples of a border image applied to a box.

MARGINS

A margin is an optional amount of space that you can add on the outside of the border. Margins keep elements from bumping into one another or the edge of the browser window or viewport.

The side-specific and shorthand **margin** properties work much like the **padding** properties we've looked at already; however, margins have some special behaviors to be aware of.

margin-top, margin-right, margin-bottom, margin-left

Values: *length* | *percentage* | auto

Default: auto

Applies to: all elements

Inherits: no

margin

Values: *length* | *percentage* | auto

Default: auto

Applies to: all elements

Inherits: no

The margin properties are very straightforward to use. You can either specify an amount of margin to appear on each side of the element or use the **margin** property to specify all sides at once.

The shorthand **margin** property works the same as the **padding** shorthand. When you supply four values, they are applied in clockwise order (top, right, bottom, left) to the sides of the element. If you supply three values, the middle value applies to both the left and right sides. When two values are provided, the first is used for the top and bottom, and the second applies to the left and right edges. Finally, one value will be applied to all four sides of the element.

As with most web measurements, ems, pixels, and percentages are the most common ways to specify margins. Be aware, however, that if you specify a percentage value, it is calculated based on the *width* of the parent element. If the parent's width changes, so will the margins on all four sides of the child element (padding has this behavior as well). The **auto** keyword allows the browser to fill in the amount of margin necessary to fit or fill the available space (see **Power Tool** sidebar).

FIGURE 14-16 shows the results of the following margin examples. I've added a red border to the elements in the examples to make their boundaries more clear. The dotted rules were added in the figure illustration to indicate the outer edges of the margins for clarity purposes only, but they are not something you'd see in the browser.

■ CSS TIP

Browser Default Margins

You may have noticed that space is added automatically around headings, paragraphs, and other block elements. That's the browser's default style sheet at work, applying margin amounts above and below those elements.

It's good to keep in mind that the browser is applying its own values for margins and padding behind the scenes. These values will be used unless you specifically override them with your own style rules.

If you are working on a design and coming across mysterious amounts of space that you didn't add, the browser's default styles may be the culprit. To troubleshoot, I recommend using your browser's Web Inspector tool, which will show you the source of all the styles applied to the element. Or if you just don't want to worry about browser styles at all, one solution is to reset the padding and margins for all elements to zero, which is discussed in the "CSS Reset" section in **Chapter 19, More CSS Techniques**.

■ POWER TOOL

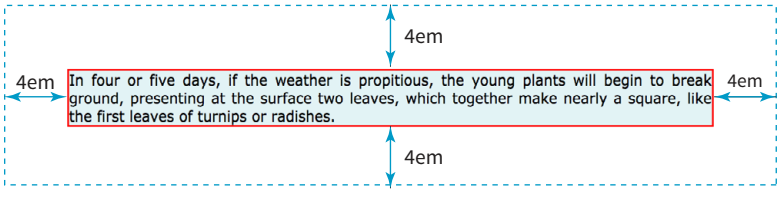
Centering with auto Margins

Setting the margin to **auto** on the left and right sides of a sized element has the effect of centering the element in its container.

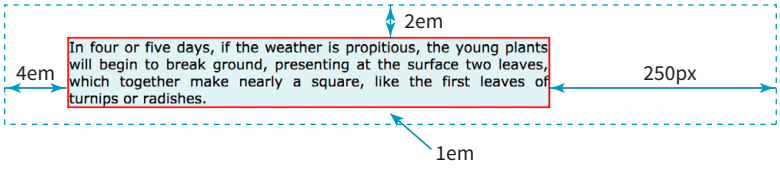
```
A  p#A {
     margin: 4em;
     border: 2px solid red;
     background: #e2f3f5;
   }
B  p#B {
     margin-top: 2em;
     margin-right: 250px;
     margin-bottom: 1em;
     margin-left: 4em;
     border: 2px solid red;
     background: #e2f3f5;
   }
C  body {
     margin: 0 20%;
     border: 3px solid red;
     background-color: #e2f3f5;
   }
```

Adding a margin to the body element adds space between the page content and the edges of the viewport.

A



B



C

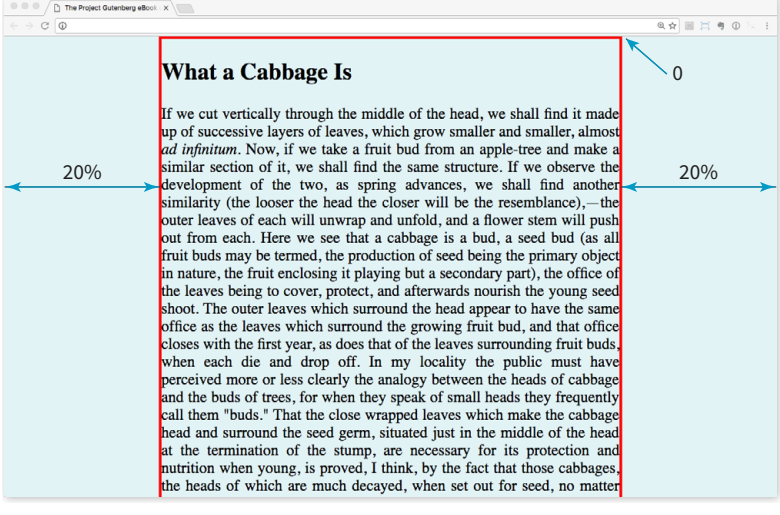


FIGURE 14-16. Applying margins to the **body** and to individual elements.

Take a look at Example C in FIGURE 14-16. Here I've applied the **margin** property to the **body** element of the document. For this particular design, I set the top margin to zero (0) so the **body** starts flush against the top edge of the browser window. Adding equal amounts of margin to the left and right sides of the **body** keeps the content of the page centered and gives it a little breathing room.

Margin Behavior

Although it is easy to write rules that apply margin amounts around HTML elements, it is important to be familiar with some of the quirks of margin behavior.

Collapsing margins

Adjacent margins overlap, and only the largest value will be used.

The most significant margin behavior to be aware of is that the top and bottom margins of neighboring elements collapse. This means that instead of accumulating, adjacent margins overlap, and only the largest value is used.

Using the two paragraphs from the previous figure as an example, if the top element has a bottom margin of 4em, and the following element has a top margin of 2em, the resulting margin space between elements does not add up to 6ems Rather, the margins collapse and the resulting margin between the paragraphs will be 4em, the largest specified value. This is demonstrated in FIGURE 14-17.

The only time top and bottom margins *don't* collapse is for floated or absolutely positioned elements (we'll get to that in **Chapter 15, Floating and Positioning**). Margins on the left and right sides never collapse, so they're nice and predictable.

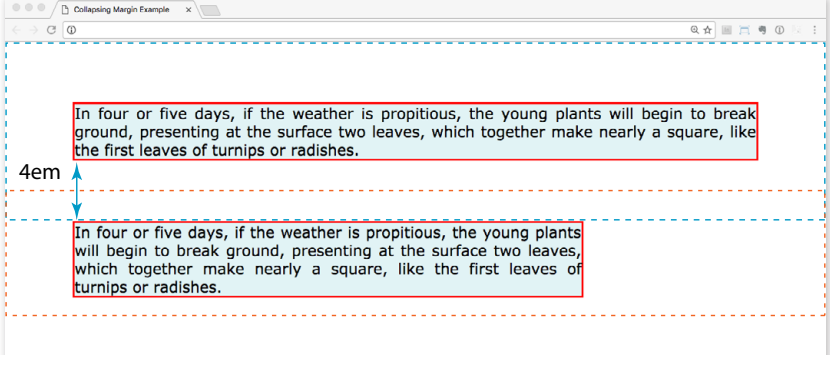


FIGURE 14-17. Vertical margins of neighboring elements collapse so that only the larger value is used.

■ FURTHER READING

Collapsing Margins

When spacing between and around elements behaves unpredictably, collapsing margins are often to blame. Here are a few articles that dig deep into collapsing margin behavior. Although they were written long ago, the information is still solid and may help you understand what is happening behind the scenes in your layouts.

- "No Margin for Error" by Andy Budd (*www.andybudd.com/archives/2003/11/no_margin_for_error*)
- "Uncollapsing Margins" by Eric Meyer (*www.complexspiral.com/publications/uncollapsing-margins*)

Margins on inline elements

You can apply top and bottom margins to inline text elements (or "non-replaced inline elements," to use the proper CSS terminology), but it won't add vertical space above and below the element, and the height of the line will not change. However, when you apply left and right margins to inline text elements, margin space *will* be held clear before and after the text in the flow of the element, even if that element breaks over several lines.

Just to keep things interesting, margins on replaced inline elements, such as images, *do* render on all sides, and therefore do affect the height of the line. See FIGURE 14-18 for examples of each.

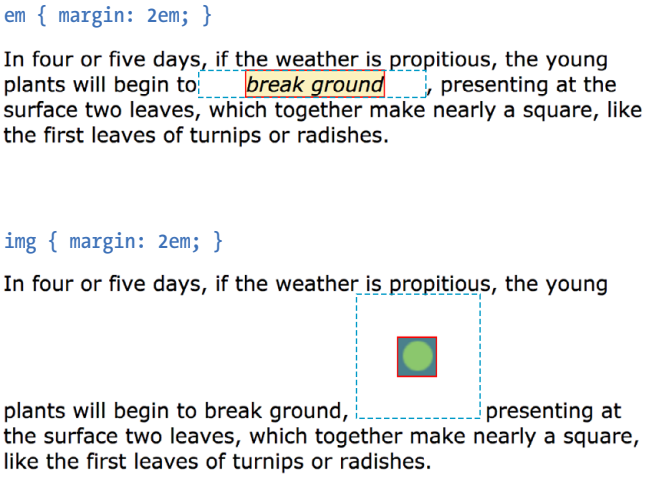


FIGURE 14-18. Examples of margins on inline elements. Only horizontal margins are rendered on non-replaced elements (top). Margins are rendered on all sides of replaced elements such as images.

Negative margins

It is worth noting that it is possible to specify negative values for margins. When you apply a negative margin, the content, padding, and border are moved in the opposite direction that would have resulted from a positive margin value.

I'll make this clear with an example. FIGURE 14-19 shows two neighboring paragraphs with different-colored borders applied to show their boundaries. In the left view, I've added a 3em bottom margin to the top paragraph, which has the effect of pushing the following paragraph *down* by that amount. If I specify a negative value (–3em), the following element moves *up* by that amount and overlaps the element with the negative margin.

`p.top { margin-bottom: 3em; }`

Pushes the following paragraph down by 3 ems.

In four or five days, if the weather is propitious, the young plants will begin to break ground, presenting at the surface two leaves, which together make nearly a square, like the first leaves of turnips or radishes.

In four or five days, if the weather is propitious, the young plants will begin to break ground, presenting at the surface two leaves, which together make nearly a square, like the first leaves of turnips or radishes.

`p.top { margin-bottom: -3em; }`

The following element moves up by 3 ems.

In four or five days, if the weather is propitious, the young plants will begin to break ground, presenting at the surface two leaves, which together make nearly a square, like the first leaves of turnips or radishes.

In four or five days, if the weather is propitious, the young plants will begin to break ground, presenting at the surface two leaves, which together make nearly a square, like the first leaves of turnips or radishes.

FIGURE 14-19. Using negative margins.

This may seem like a strange thing to do, and in fact, you probably wouldn't make blocks of text overlap as shown. The point here is that you can use margins with both positive and negative values to move elements around on the page. This is the basis of some older CSS layout techniques.

Now let's use margins to add some space between parts of the Black Goose Bakery home page in EXERCISE 14-3.

ASSIGNING DISPLAY TYPES

As long as we're talking about boxes and the CSS layout model, this is a good time to introduce the **`display`** property. You should already be familiar with the display behavior of block and inline elements. Although HTML assigns display behaviors (or display types, to use the latest CSS term) to the elements it defines, there are other XML-based languages that can use CSS that don't do the same. For this reason, the **`display`** property was created to allow authors to specify how elements should behave in layouts.

`display`

Values:	`inline` | `block` | `run-in` | `flex` | `grid` | `flow` | `flow-root` | `list-item` | `table` | `table-row-group` | `table-header-group` | `table-footer-group` | `table-row` | `table-cell` | `table-column-group` | `table-column` | `table-caption` | `ruby` | `ruby-base` | `ruby-text` | `ruby-base-container` | `ruby-text-container` | `inline-block` | `inline-table` | `inline-flex` | `inline-grid` | `contents` | `none`
Default:	`inline`
Applies to:	all elements
Inherits:	yes

EXERCISE 14-3. Adding margin space around elements

It's time to adjust the margins around the elements on the bakery page. We'll start by adjusting margins on the whole document, and then make tweaks to each section from top to bottom. You should have *bakery-styles.css* open in a text editor.

1. It is a common practice to set the margin for the **body** element to zero, thus clearing out the browser's default margin setting. Add this margin declaration to the **body** styles, and then save the file and open it in a browser. You'll see that the elements now go to the very edge of the window with no space between.

```
body {
  …
  margin: 0;
}
```

NOTE

When the value is 0, you don't need to provide a specific unit.

2. If you are a careful observer, you may have noticed that there is still a bit of whitespace above the colored navigation bar. That happens to be the top margin of the **ul** list pushing the whole **nav** element down from the top edge of the browser. Let's take care of that. Add a new style rule in the "nav styles" section of the style sheet:

```
nav ul {
  margin: 0;
}
```

3. Margins are good for nudging elements around in the layout. For example, I think I'd like to move the **h1** with the logotype down a bit, so I'll add a margin to its top edge. I played around with a few values before deciding on 1.5em for this new style rule:

```
header h1 {
  margin-top: 1.5em;
}
```

I'd like the intro paragraph in the header to be a little closer to the logotype, so let's get wacky and use a *negative* top margin to pull it *up*. Add this declaration to the existing style rule:

```
header p {
  …
  margin-top: -12px;
}
```

4. Give the **main** section a 2.5% margin on all sides:

```
main {
  …
  margin: 2.5%;
}
```

5. Add a little extra space above the **h3** headings in the main area. I've chosen 2.5em, but you can play around with different values to see what you like best:

```
h3 {
  …
  margin-top: 2.5em;
}
```

6. Finally, add some space around the **aside**. This time, we'll do different amounts on each side for kicks. Put 1em on the top, 2.5% on the right side, 0 on the bottom, and 10% margin on the left. I'm going to let you do this one yourself. Can you make all those changes with one declaration? If you want to check your work, my finished version of the Black Goose Bakery page so far is available with the exercise materials for this chapter.

7. Save the style sheet again, and reload the page in the browser. It should look like the one in FIGURE 14-20. This isn't the most beautiful design, particularly if your browser window is set wide. However, if you resize your browser window narrow, you'll find that it wouldn't be too bad as the small-screen version in a responsive design. (Bet you can't wait for the Responsive Web Design chapter to learn how to fix this!)

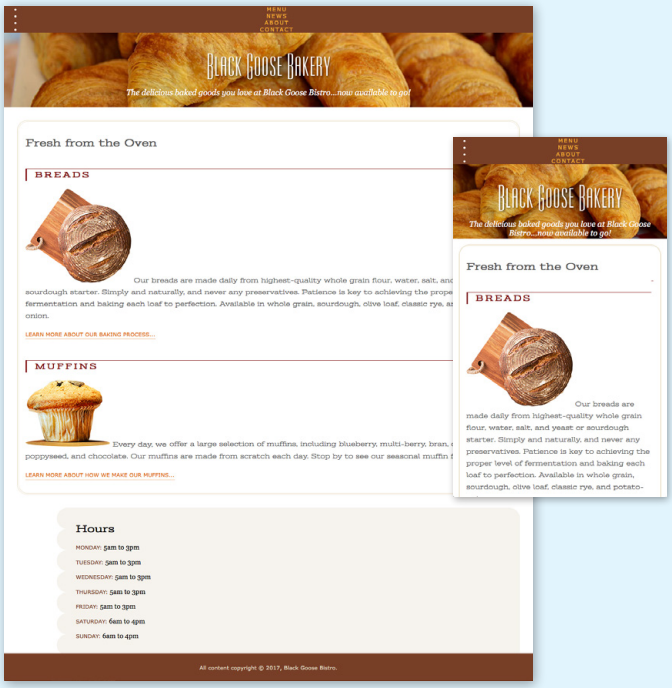


FIGURE 14-20. The Black Goose Bakery home page after padding, borders, and margins are added.

The **display** property defines the type of element box an element generates in the layout. In addition to the familiar **inline** and **block** display types, you can also make elements display as list items or the various parts of a table. There are also a number of values for ruby annotation for East Asian languages. As you can see from the list of values, there are a lot of display types, but there are only a few that are used in everyday practice.

Display type assignment is useful for achieving layout effects while keeping the semantics of the HTML source intact. For example, it is common practice to make **li** elements (which usually display with the characteristics of block elements) display as inline elements to turn a list into a horizontal navigation bar. You may also make an otherwise inline **a** (anchor) element display as a block in order to give it a specific width and height:

```
ul.navigation li { display: inline; }
ul.navigation li a { display: block; }
```

Another useful value for the **display** property is **none**, which removes the content from the normal flow entirely. Unlike **visibility: hidden**, which just makes the element invisible but keeps the space it would have occupied blank, **display: none** removes the content, and the space it would have occupied is closed up.

One popular use of **display: none** is to prevent certain content in the source document from displaying in specific media, such as when the page is printed or displayed on devices with small screens. For example, you could display URLs for links in a document when it is printed, but not when it is displayed on a computer screen where the links are interactive.

Be aware that content that has its **display** set to **none** still downloads with the document. Setting some content to **display:none** for devices with small screens may keep the page shorter, but it is not doing anything to reduce data usage or download times.

WARNING

Bear in mind that changing the presentation of an HTML element with the CSS **display** *property does* ***not*** *change the definition of that element as block-level or inline in HTML. Putting a block-level element within an inline element will always be invalid, regardless of its display role.*

FURTHER READING

"Five Ways to Hide Elements in CSS" by Baljeet Rathi (*www.sitepoint.com/five-ways-to-hide-elements-in-css*) compares various methods for hiding content, including **display: none**.

BOX DROP SHADOWS

We've arrived at the last stop on the element box tour. In **Chapter 12, Formatting Text,** you learned about the **text-shadow** property, which adds a drop shadow to text. The **box-shadow** property applies a drop shadow around the entire visible element box (excluding the margin).

BROWSER SUPPORT NOTE

Browsers released before 2011 require vendor prefixes for **box-shadow**. *Box shadows are not supported at all in Internet Explorer versions 8 and earlier. This is a case for progressive enhancement—it is likely that a box without a shadow will be just fine for users clinging to old browser versions.*

box-shadow

Values:	*'horizontal offset' 'vertical offset' 'blur distance' 'spread distance' color* inset | none
Default:	none
Applies to:	all elements
Inherits:	no

The value of the **box-shadow** property should seem familiar to you after working with **text-shadow**: specify the horizontal and vertical offset distances, the amount the shadow should blur, and a color. For box shadows, you can also specify a spread amount, which increases (or decreases with negative values) the size of the shadow. By default, the shadow color is the same as the foreground color of the element, but specifying a color overrides it.

FIGURE 14-21 shows the results of the following code examples. Ⓐ adds a simple box shadow 6 pixels to the right and 6 pixels down, without blur or spread. Ⓑ adds a blur value of 5 pixels, and Ⓒ shows the effect of a 10-pixel spread value. Box shadows are always applied to the area *outside* the border of the element (or the place it would be if a border isn't specified). If the element has a transparent or translucent background, you will not see the box shadow in the area behind the element.

Ⓐ
```
box-shadow: 6px 6px gray;
```

Ⓑ
```
box-shadow: 6px 6px 5px gray; /* 5 pixel blur */
```

Ⓒ
```
box-shadow: 6px 6px 5px 10px gray; /* 5px blur, 10px spread */
```

You can make the shadow render inside the edges of the visible element box by adding the **inset** keyword to the rule. This makes it look like the element is pressed into the screen (FIGURE 14-22).

```
box-shadow: inset 6px 6px 5px gray;
```

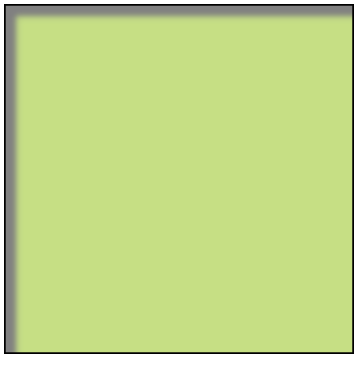


FIGURE 14-22. An inset box shadow renders on the inside of the element box.

As for **text-shadow**, you can specify multiple box shadows on an element by providing the values in a comma-separated list. The values that come first get placed on top, and subsequent shadows are placed behind it in the order in which they appear in the list.

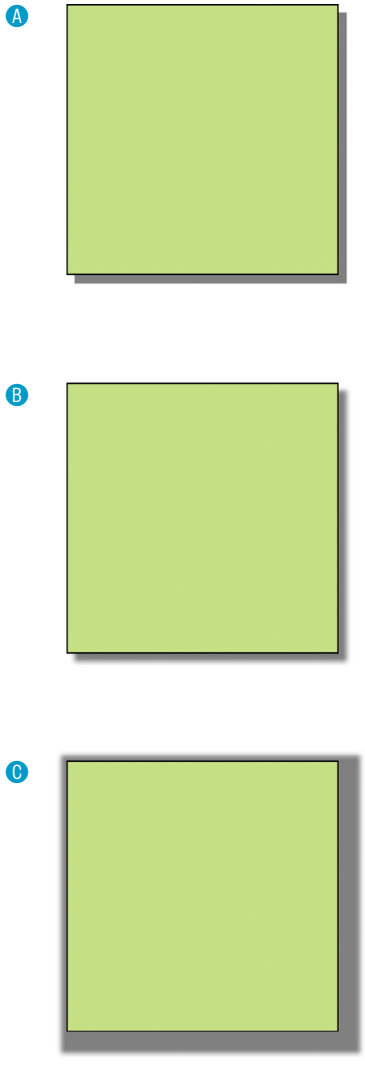


FIGURE 14-21. Adding drop shadows around an element with the **box-shadow** property.

WARNING

Box shadows, text shadows, and gradients take a lot of processor power because you are shifting the burden of interpreting and rendering them onto the browser. The more you use, the slower performance will be, and as we all know, performance is everything on the web. So go easy on them.

TEST YOURSELF

At this point, you should have a good feel for element boxes and how to manipulate the space within and around them. In the next chapter, we'll start moving the boxes around on the page, but first, why not get some practice at writing rules for padding, borders, and margins in the following test?

In this test, your task is to write the declarations that create the effects shown in each example in FIGURE 14-23 (see **Useful Hints**). All the paragraphs shown here share a rule that sets the dimensions and the background color for each paragraph. You just need to provide the box-related property declarations. Answers, as always, appear in **Appendix A**.

USEFUL HINTS

- Outer margin edges are indicated by dotted blue lines.
- All necessary measurements are provided in blue.
- Borders are either **black** or **red**.

Ⓐ

Ⓑ

Ⓒ

Ⓓ

Ⓔ

Ⓕ

CSS REVIEW: BOX PROPERTIES

Property	Description
`border`	A shorthand property that combines border properties

`border-top`
`border-right`
`border-bottom`
`border-left`
	Combines border properties for each side of the element

`border-color`	Shorthand property for specifying the color of borders

`border-top-color`
`border-right-color`
`border-bottom-color`
`border-left-color`
	Specifies the border color for each side of the element

`border-image`	Adds an image inside the border area
`border-image-outset`	How far the border image should be positioned away from the border area.
`border-image-repeat`	The manner in which the image fills the sides of the border
`border-image-slice`	The points at which the border image should be divided into corners and sides
`border-image-source`	The location of the image file to be used for the border image

Table continues...

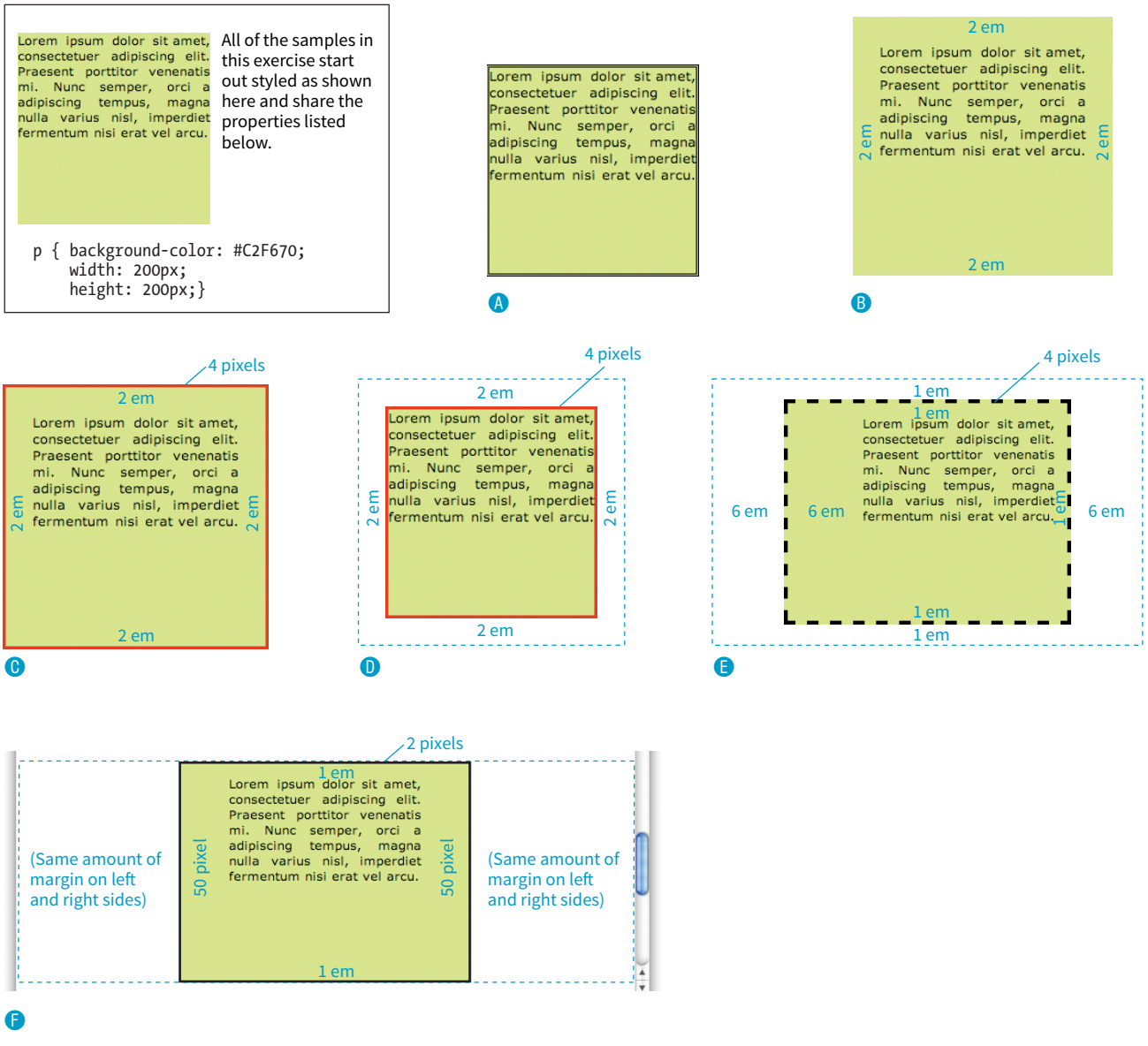


FIGURE 14-23. Write the declarations for these examples.

Property	Description
`border-image-width`	The width of the space the border image should occupy
`border-radius`	Shorthand property for rounding the corners of the visible element box

`border-top-left-radius`
`border-top-right-radius`
`border-bottom-right-radius border-bottom-left-radius`
	Specifies the radius curve for each individual corner

`border-style`	Shorthand property for specifying the style of borders

`border-top-style`
`border-right-style`
`border-bottom-style`
`border-left-style`
	Specifies the border style for each side of the element

`border-width`	Shorthand property for specifying the width of borders

`border-top-width`
`border-right-width`
`border-bottom-width`
`border-left-width`
	Specifies the border width for each side of the element

`box-sizing`	Specifies whether width and height dimensions apply to the content box or the border box
`box-shadow`	Adds a drop shadow around the visible element box
`display`	Defines the type of element box an element generates
`height`	Specifies the height of the element's content box or border box
`margin`	Shorthand property for specifying margin space around an element

`margin-top`
`margin-right`
`margin-bottom`
`margin-left`
	Specifies the margin amount for each side of the element

`max-height`	Specifies the maximum height of an element
`max-width`	Specifies the maximum width of an element
`min-height`	Specifies the minimum height of an element
`min-width`	Specifies the minimum width of an element
`outline`	Shorthand property for adding an outline around an element
`outline-color`	Sets the color of the outline
`outline-offset`	Sets space between an outline and the outer edge of the border
`outline-style`	Sets the style of the outline
`outline-width`	Sets the width of the outline
`overflow`	Specifies how to handle content that doesn't fit in the content area
`padding`	Shorthand property for specifying space between the content area and the border

`padding-top`
`padding-right`
`padding-bottom`
`padding-left`
	Specifies the padding amount for each side of the element

`width`	Specifies the width of an element's content box or border box

CHAPTER 15

FLOATING AND POSITIONING

At this point, you've learned dozens of CSS properties that let you change the appearance of text elements and the boxes they generate. But so far, we've merely been formatting elements as they appear in the flow of the document.

In this chapter, we'll look at floating and positioning, the CSS methods for breaking out of the normal flow and arranging elements on the page. Floating an element moves it to the left or right and allows the following text to wrap around it. Positioning is a way to specify the location of an element anywhere on the page with pixel precision.

Before we start moving elements around, let's be sure we are well acquainted with how they behave in the normal flow.

IN THIS CHAPTER

NORMAL FLOW

We've covered the normal flow in previous chapters, but it's worth a refresher. In the CSS layout model, text elements are laid out from top to bottom in the order in which they appear in the source, and from left to right in left-to-right reading languages (see **Note**). Block elements stack up on top of one another and fill the available width of the browser window or other containing element. Inline elements and text characters line up next to one another to fill the block elements.

When the window or containing element resizes, the block elements expand or contract to the new width, and the inline content reflows to fit as shown in FIGURE 15-1.

Objects in the normal flow affect the layout of the objects around them. This is the behavior you've come to expect in web pages—elements don't overlap or bunch up. They make room for one another.

NOTE

For right-to-left reading languages such as Arabic and Hebrew, the normal flow is top to bottom and right to left.

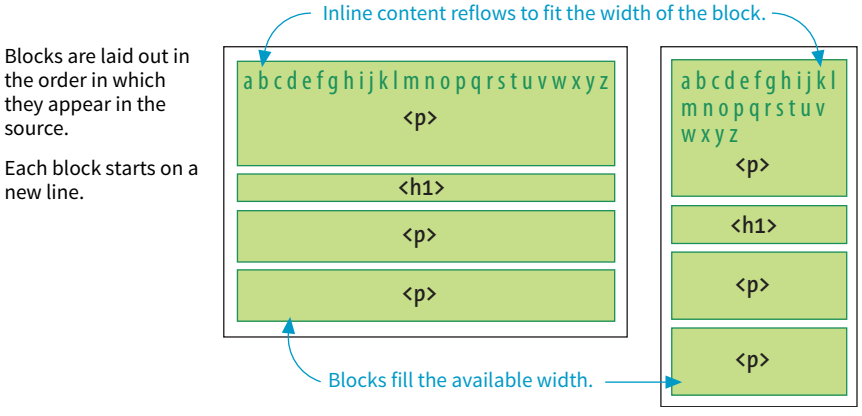


FIGURE 15-1. One more example of the normal flow behavior.

We've seen all of this before, but in this chapter we'll be paying attention to whether elements are in the flow or removed from the flow. Floating and positioning change the relationship of elements to the normal flow in different ways. Let's first look at the special behavior of floated elements (or "floats" for short).

FLOATING

Simply stated, the **float** property moves an element as far as possible to the left or right, allowing the following content to wrap around it. It is a unique feature built into CSS with some interesting behaviors.

Floating an element moves it to the left or right and allows the following text to wrap around it.

float

Values: left | right | none

Default: none

Applies to: all elements

Inherits: no

The best way to explain floating is to demonstrate it. In this example, the **float** property is applied to an **img** element to float it to the right. FIGURE 15-2 shows how the paragraph and the contained image are rendered by default (top) and how it looks when the **float** property is applied (bottom).

THE MARKUP

```
<p><img src="icecreambowl.png" alt=""> After the cream is frozen rather
stiff,…
```

THE STYLES

```
img {
  float: right;
}
```

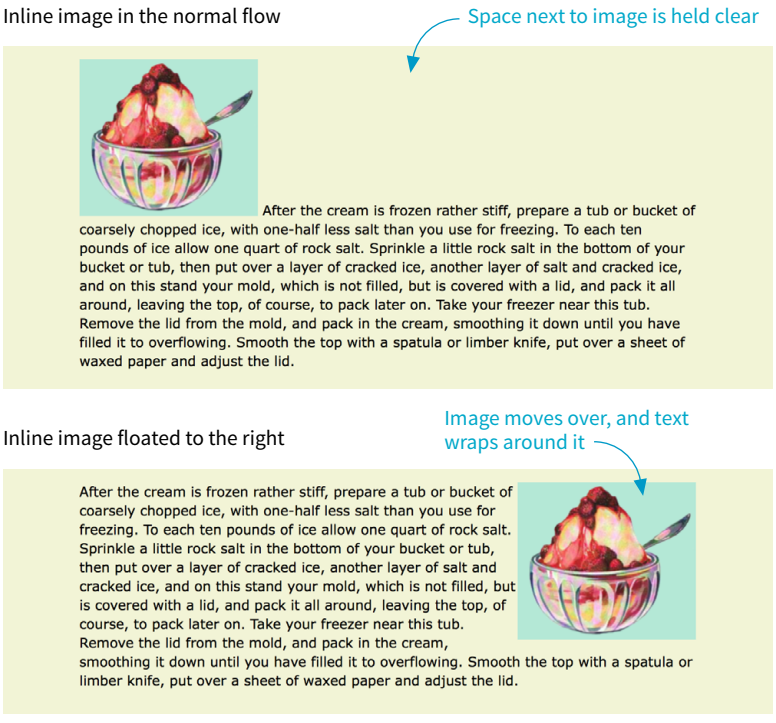


FIGURE 15-2. The layout of an image in the normal flow (top), and with the **float** property applied (bottom).

That's a nice effect. We've gotten rid of a lot of wasted space on the page, but now the text is bumping right up against the image. How do you think you would add some space between the image element and the surrounding text? If you guessed "add a margin," you're absolutely right. I'll add 1em of space on all sides of the image with the **margin** property (FIGURE 15-3). You can begin to see how the box properties work together to improve page layout.

```
img {
  float: right;
  margin: 1em;
}
```

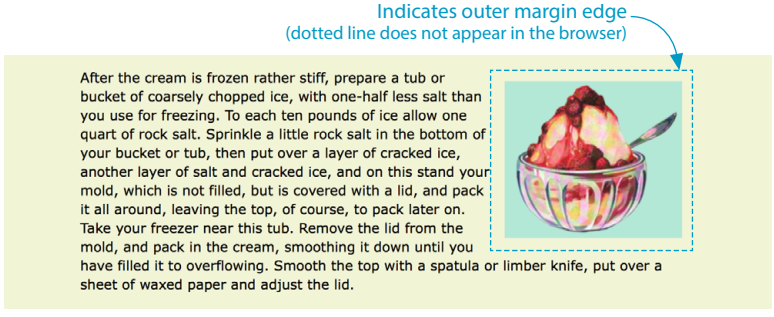


FIGURE 15-3. Adding a 1em margin around the floated image.

The previous two figures demonstrate some key behaviors of floated elements:

A floated element is like an island in a stream.

First and foremost, you can see that the image is removed from its position in the normal flow yet continues to influence the surrounding content. The subsequent paragraph text reflows to make room for the floated `img` element. One popular analogy compares floats to islands in a stream—they are not in the flow, but the stream has to flow around them. This behavior is unique to floated elements.

Floats stay in the content area of the containing element.

It is also important to note that the floated image is placed within the *content area* (the inner edges) of the paragraph that contains it. It does not extend into the padding area of the paragraph.

Margins are maintained.

In addition, margins are held on all sides of the floated image, as indicated in FIGURE 15-3 by the dotted line. In other words, the entire element box, from outer edge to outer edge, is floated.

Floating Inline and Block elements

Those are the basics, so now let's look at more examples and explore additional floating behaviors. It is possible to float any HTML element, both inline and block-level, as we'll see in the following examples.

Floating an inline text element

In the previous example, we floated an inline image element. This time, let's look at what happens when you float an inline text (non-replaced) element—in this case, a span of text (FIGURE 15-4).

THE MARKUP

```
<p><span class="tip">TIP: Make sure that your packing tub or bucket
has a hole below the top of the mold so  the  water will drain
off.</span>After the cream is frozen rather stiff, prepare a tub or
bucket of…</p>
```

THE STYLES

```
span.tip {
  float: right;
  margin: 1em;
  width: 200px;
  color: #fff;
  background-color: lightseagreen;
  padding: 1em;
}
```

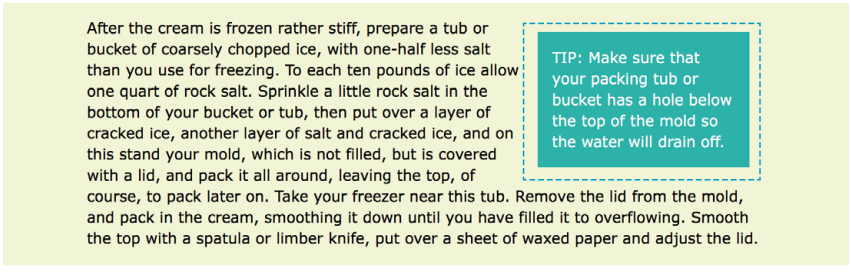


FIGURE 15-4. Floating an inline text (non-replaced) element.

At a glance, it is behaving the same as the floated image, which is what we'd expect. But there are some subtle things at work here that bear pointing out:

Always provide a width for floated text elements.

First, you'll notice that the style rule that floats the `span` includes the `width` property. It is necessary to specify a width for a floated text element because without one, its box is sized wide enough to fit its content (`auto`). For short phrases that are narrower than the container, that might not be an issue. However, for longer, wrapped text, the box expands to the width of the container, making it so wide that there wouldn't be room to wrap anything around it. Images have an inherent width, so we didn't need to specify a width in the previous example (although we certainly could have).

It is necessary to specify the width for floated text elements.

Floated inline elements behave as block elements.

Notice that the margin is held on all four sides of the floated `span` text, even though top and bottom margins are usually not rendered on inline elements (see FIGURE 14-20 in the previous chapter). That is because all floated elements behave like block elements. Once you float an inline element, it follows the display rules for block-level elements, and margins are rendered on all four sides.

Margins on floated elements do not collapse.

In the normal flow, abutting top and bottom margins collapse (overlap), but margins for floated elements are maintained on all sides as specified.

Floating block elements

Let's look at what happens when you float a block within the normal flow. In this example, the whole second paragraph element is floated to the left (FIGURE 15-5).

THE MARKUP

```
<p>If you wish to pack ice cream...</p>
<p id="float">After the ice cream is rather stiff,...</p>
<p>Make sure that your packing tub or bucket...</p>
<p>As cold water is warmer than the ordinary...</p>
```

THE STYLES

```
p {
  border: 2px red solid;
}

#float {
  float: left;
  width: 300px;
  margin: 1em;
  background: white;
}
```

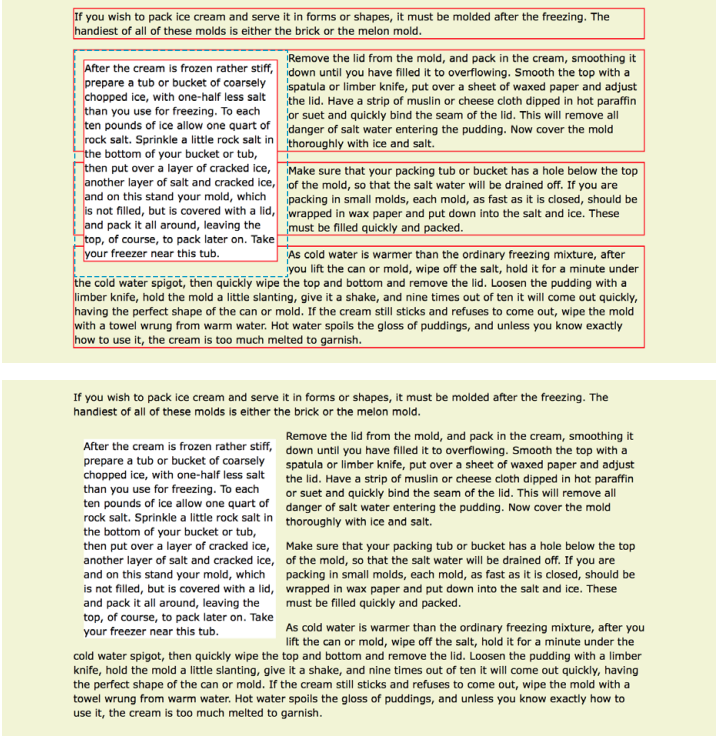


FIGURE 15-5. Floating a block-level element.

I've added a red border around all `p` elements to reveal their boundaries. In addition, I've made the background of the floated paragraph white so it stands out and added a 1em margin on all sides (indicated with a blue dotted line). The bottom view in FIGURE 15-5 shows how it looks with all the extra stuff turned off, as it would more likely appear on a real page.

Just as we saw with the image, the paragraph moves off to the side (left this time), and the following content wraps around it, even though blocks normally stack on top of one another. There are a few things I want to point out in this example:

You must provide a width for floated block elements.

If you do not provide a **`width`** value, the width of the floated block will be set to **`auto`**, which fills the available width of the browser window or other containing element. There's not much sense in having a full-width floated box, because the idea is to wrap text next to the float, not start below it.

Elements do not float higher than their reference in the source.

A floated block will float to the left or right relative to where it occurs in the source, allowing the following elements in the flow to wrap around it. It stays below any block elements that precede it in the flow (in effect, it is "blocked" by them). That means you can't float an element up to the top corner of a page, even if its nearest ancestor is the **`body`** element. If you want a floated element to start at the top of the page, it must appear first in the document source (see **Note**).

Non-floated elements maintain the normal flow.

The red borders in the top image reveal that the element boxes for the surrounding paragraphs still extend the full width of the normal flow. Only the content of those elements wraps around the float. This is a good model to keep in mind.

For example, adding a left margin to the surrounding paragraphs would add space on the left edge of the page, not between the text and the floated element. If you want space between the float and the wrapped text, you need to apply the margin to the float itself.

NOTE

Absolute positioning is the CSS method for placing elements on a page regardless of how they appear in the source. We'll get to absolute positioning later in this chapter. You can also change the order in which elements display by using Flexbox and Grid as discussed in **Chapter 16, CSS Layout with Flexbox and Grid**.

Clearing Floated Elements

If you're going to be floating elements around, it's important to know how to turn the text wrapping *off* and get back to normal flow as usual. You do this by clearing the element that you want to start below the float. Applying the **`clear`** property to an element prevents it from appearing next to a floated element and forces it to start against the next available "clear" space below the float.

`clear`

Values:	`left` | `right` | `both` | `none`
Default:	`none`
Applies to:	block-level elements only
Inherits:	no

Keep in mind that you apply the **`clear`** property to the element you want to start below the floated element, not the floated element itself. The **`left`** value starts the element below any elements that have been floated to the left. Similarly, the **`right`** value makes the element clear all floats on the right edge of the containing block. If there are multiple floated elements, and you

want to be sure an element starts below all of them, use the **both** value to clear floats on both sides.

In this example, the **clear** property has been used to make **h2** elements start below left-floated elements. FIGURE 15-6 shows how the **h2** heading starts at the next available clear edge below the float.

```
img {
  float: left;
  margin-right: .5em;
}
h2 {
  clear: left;
  margin-top: 2em;
}
```

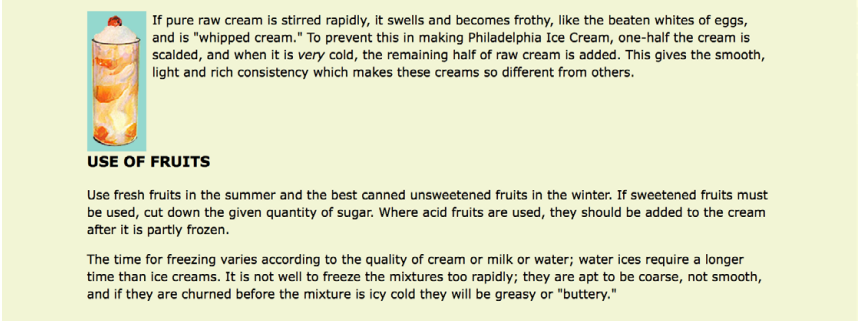


FIGURE 15-6. Clearing a left-floated element.

Notice in FIGURE 15-6 that although there is a 2em top margin applied to the **h2** element, it is not rendered between the heading and the floated image. That's the result of collapsing vertical margins in the flow. If you want to make sure space is held between a float and the following text, apply a bottom margin to the floated element itself.

By now you have enough float know-how to give it a try in EXERCISE 15-1.

Floating Multiple Elements

It's perfectly fine to float multiple elements on a page or even within a single element. In fact, for years, floats have been the primary method for lining up elements like navigation menus and even for creating whole page layouts (please take time to read the sidebar **"Float-Based Layouts"**).

When you float multiple elements, there is a complex system of behind-the-scenes rendering rules that ensures floated elements do not overlap. You can consult the CSS specification for details, but the upshot of it is that floated elements will be placed as far left or right (as specified) and as high up as space allows.

Float-Based Layouts

Curiously, there were no tools for true page layout in CSS1 or CSS2. Some clever designers realized we could use the CSS float behavior to line up elements horizontally, and floats started being used to turn lists into navigation bars and even turn whole sections of a document into columned layouts.

Float-based layouts are still prevalent on the web as I write this, but now that better CSS layout tools like Flexbox and Grid are gaining browser support, we are recognizing our float-based layouts for the hacks they are. Float-based layouts will eventually vanish like the table-based layouts of the 1990s.

That said, we are in a time of transition. Not all browsers in use today support newer standards like Flexbox and Grid, so depending on the browsers you need to support, you may still need to provide a fallback design that is universally supported. Floats will get the job done.

If you need to support older browsers that do not support Flexbox and Grid, you can download my article **"Page Layout with Floats and Positioning"** (PDF), which is available at *learningwebdesign.com/articles/*. It contains lessons on how to create navigation bars with floats and a number of templates for creating multicolumn layouts with floats and positioning. You may never need to know these techniques, but the information is there if you do.

EXERCISE 15-1. Floating images

In the exercises in this chapter, we'll make further improvements to the Black Goose Bakery home page that we worked on in **Chapter 14, Thinking Inside the Box**. If you did not follow along in the last chapter, or if you would just like a fresh start, there is a copy of the document in its most recent state (*bakery_ch15.html*) in the **Chapter 15** materials (*learningwebdesign.com/5e/materials*).

1. Open the CSS file in a text editor and the HTML document in the browser. We'll start by removing wasted vertical space next to the baked good images by floating those images to the left. We'll create a new style rule with a contextual selector to target only the images in the **main** section:

   ```
   main img {
     float: left;
   }
   ```

   Save the CSS file and refresh the page in the browser, and you'll see that we have some post-float tidying up to do.

2. I want the "Learn more" links to always appear below the images so they are clearly visible and consistently on the left side of the page. Fortunately, the paragraphs with those links are marked up with the class "more" and there is already a style rule for them using a class selector. Make those paragraphs clear any floats on the left edge.

   ```
   p.more {
     …
     clear: left;
   }
   ```

3. Lastly, we'll adjust the spacing around the floated images. Give both images a 1em margin on the right and bottom sides by using the shorthand **margin** property:

   ```
   main img {
     float: left;
     margin: 0 1em 1em 0;
   }
   ```

   I feel like the muffin image could use extra space on the left side so it lines up better with the bread. Use this nifty attribute selector to grab any image whose **src** attribute contains the word *muffin* (there's only one):

   ```
   img[src*="muffin"] {
     margin-left: 50px;
   }
   ```

FIGURE 15-7 shows the new and improved "Fresh from the Oven" section.

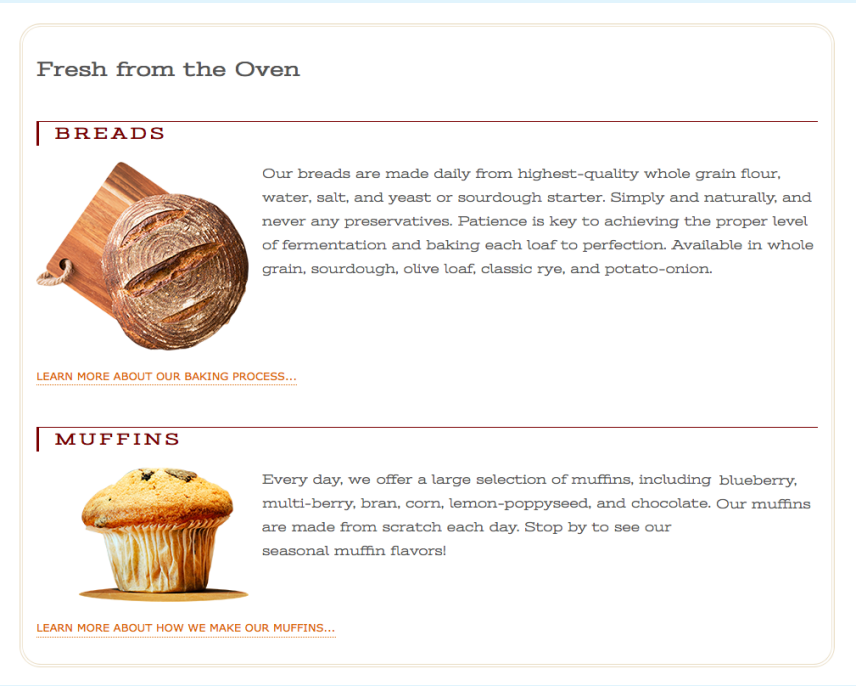


FIGURE 15-7. The product section with floated images and wrapped text has less wasted space.

FIGURE 15-8 shows what happens when a series of sequential paragraphs is floated to the same side. The first three floats start stacking up from the left edge, but when there isn't enough room for the fourth, it moves down and to the left until it bumps into something—in this case, the edge of the browser window. However, if one of the floats, such as P2, had been very long, it would have bumped up against the edge of the long float instead. Notice that the next paragraph in the normal flow (P6) starts wrapping at the highest point it can find, just below P1.

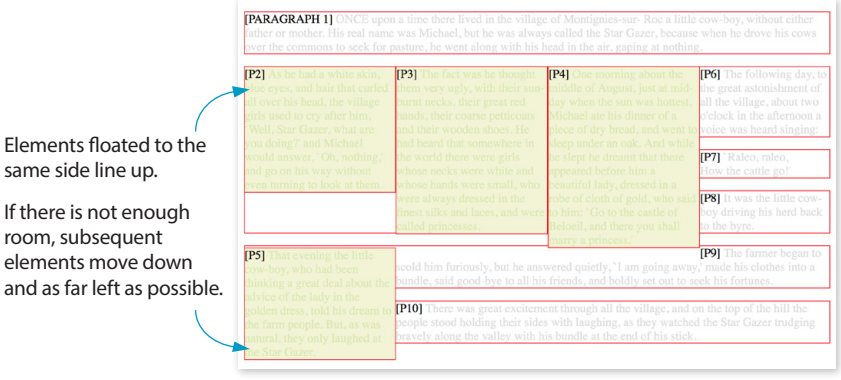


FIGURE 15-8. Multiple floated elements line up and do not overlap.

THE MARKUP

```
<p>[PARAGRAPH 1] ONCE upon a time…</p>
<p class="float">[P2]…</p>
<p class="float">[P3]…</p>
<p class="float">[P4]…</p>
<p class="float">[P5]…</p>
<p>[P6]…</p>
<p>[P7]…</p>
<p>[P8]…</p>
<p>[P9]…</p>
<p>[P10]…</p>
```

THE STYLES

```
p.float {
  float: left;
  width: 200px;
  margin: 0px;
  background: #F2F5d5;
  color: #DAEAB1;
}
```

Containing Floats

This is a good time to address a quirky float behavior: float containment. By default, floats are designed to hang out of the element they are contained in. That's just fine for allowing text to flow around a floated image, but sometimes it can cause some unwanted behaviors.

Take a look at the example in FIGURE 15-9. It would be nicer if the border expanded around all the content, but the floated image hangs out the bottom.

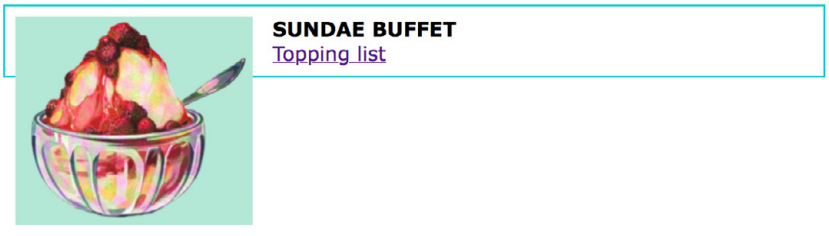


FIGURE 15-9. The containing element does not expand to accommodate the floated image as indicated by its blue border.

If you float *all* the elements in a container element, there will be no elements remaining in the flow to hold the containing element open. This phenomenon is illustrated in FIGURE 15-10. The **#container div** contains two paragraphs. The view of the normal flow (top) shows that the **#container** has a background color and border that wraps around the content.

```
<div id="container">
  <p>…</p>
  <p>…</p>
</div>

#container {
  background: #f2f5d5;
  border: 2px dashed green;
}
```

However, when both paragraphs (that is, all of the content within the **div**) are floated, as shown in the figure on the bottom), the element box for the **#container** closes up to a height of zero, leaving the floats hanging down below (you can still see the empty border at the top). There's no content left in the normal flow to give the containing **div** height. This clearly is not the effect we are after.

```
p {
  float: left;
  width: 44%;
  padding: 2%;
}
```

In the normal flow, the container div encloses the paragraphs.

Etiam convallis, nulla ut ullamcorper mollis, ipsum purus imperdiet tellus, ut ultrices massa tortor vitae nulla. Fusce non arcu quam. Nullam lacinia facilisis lacus, et varius ligula imperdiet ut. Morbi molestie auctor magna, quis venenatis felis adipiscing sed. Aliquam ipsum nibh, dapibus sit amet tristique at, tincidunt in leo. Quisque accumsan lobortis lacus, id gravida tortor luctus et. Donec quis diam et odio volutpat blandit nec nec enim. Nam vitae vestibulum risus. Cras in adipiscing odio. Nam vel dolor id purus pretium suscipit quis in quam. Proin varius tincidunt facilisis. Maecenas eget felis ut nisi ullamcorper ornare. Suspendisse vestibulum leo sed lectus posuere eget convallis nisi placerat. Vestibulum porttitor egestas ornare.

Cras id ipsum dui. Donec semper congue lectus quis vulputate. Ut felis leo, bibendum at blandit non, luctus ac lorem. Nunc vitae ligula ut neque convallis sagittis. Quisque consequat orci sed arcu tincidunt et volutpat tellus tempor. Nulla vulputate ante nec felis elementum auctor. Duis magna neque, posuere eu hendrerit sit amet, dapibus quis quam. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Nunc dapibus dui dignissim dolor rutrum vel consequat nibh sagittis. Morbi non dolor diam, nec iaculis neque. Aenean at eros sit amet velit iaculis porttitor. Nam lobortis sodales augue, sit amet tincidunt erat sagittis eu. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Donec ut ultricies velit. Quisque tempor fermentum ante, quis tempus est fringilla eu.

When both paragraphs are floated, the container does not stretch around them.

Etiam convallis, nulla ut ullamcorper mollis, ipsum purus imperdiet tellus, ut ultrices massa tortor vitae nulla. Fusce non arcu quam. Nullam lacinia facilisis lacus, et varius ligula imperdiet ut. Morbi molestie auctor magna, quis venenatis felis adipiscing sed. Aliquam ipsum nibh, dapibus sit amet tristique at, tincidunt in leo. Quisque accumsan lobortis lacus, id gravida tortor luctus et. Donec quis diam et odio volutpat blandit nec nec enim. Nam vitae vestibulum risus. Cras in adipiscing odio. Nam vel dolor id purus pretium suscipit quis in quam. Proin varius tincidunt facilisis. Maecenas eget felis ut nisi ullamcorper pretium non at nulla. Etiam suscipit aliquet velit ac facilisis. Etiam egestas ante eu velit ullamcorper ornare. Suspendisse vestibulum leo sed lectus posuere eget convallis nisi placerat. Vestibulum porttitor egestas ornare.

Cras id ipsum dui. Donec semper congue lectus quis vulputate. Ut felis leo, bibendum at blandit non, luctus ac lorem. Nunc vitae ligula ut neque convallis sagittis. Quisque consequat orci sed arcu tincidunt et volutpat tellus tempor. Nulla vulputate ante nec felis elementum auctor. Duis magna neque, posuere eu hendrerit sit amet, dapibus quis quam. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Nunc dapibus dui dignissim dolor rutrum vel consequat nibh sagittis. Morbi non dolor diam, nec iaculis neque. Aenean at eros sit amet velit iaculis porttitor. Nam lobortis sodales augue, sit amet tincidunt erat sagittis eu. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Donec ut ultricies velit. Quisque tempor fermentum ante, quis tempus est fringilla eu.

FIGURE 15-10. The container box disappears entirely when all its contents are floated.

Fortunately, there are a few fixes to this problem, and they are pretty straightforward. The most popular and foolproof solution is the "clearfix" technique. It uses the **:after** pseudo-element to insert a character space after the container, set its display to "block," and clear it on both sides. For more information on this version of clearfix, see Thierry Koblentz's article "The very latest clearfix reloaded" (*cssmojo.com/the-very-latest-clearfix-reloaded*). Here it is applied to the **#container div** in FIGURE 15-10:

```
#container:after {
  content: " ";
  display: block;
  clear: both;
  background-color: #f2f5d5; /*light green*/
  border: 2px dashed green;
  padding: 1em;
  }
```

Another option is to float the containing element as well and give it a width of 100%:

```
#container {
  float: left;
  width: 100%;
  …
  }
```

FIGURE 15-11 shows the result of applying a containment technique to the previous examples. Either will do the trick.

FIGURE 15-11. Our hanging floats are now contained.

That covers the fundamentals of floating. If you are thinking that rectangular text wraps are a little *ho-hum*, you could add some pizzazz (or just eliminate extra whitespace) by using CSS Shapes.

The Future of Clearfix

A new display value, **flow-root**, may make the clearfix hack obsolete once and for all. Setting the **display** of a container element to **flow-root** makes the element automatically expand to contain its floats. As of this writing, it is still in an experimental phase, but it's worth keeping an eye on. A potential disadvantage is that it disables collapsing margins between the element and its first/last child, which can produce unpredictable results. You can read more about the **flow-root** method in Rachel Andrew's post "The end of the clearfix hack?" (*rachelandrew.co.uk/archives/2017/01/24/the-end-of-the-clearfix-hack*).

FANCY TEXT WRAP WITH CSS SHAPES

Look at the previous float examples, and you will see that the text always wraps in a rectangular shape around a floated image or element box. However, you can change the shape of the wrapped text to a circle, ellipse, polygon, or any image shape by using the **shape-outside** property. This is an up-and-coming CSS feature, so be sure to check the **Browser Support Note**. Following is a quick introduction to CSS Shapes, which should inspire and prepare you for more exploration on your own.

shape-outside

Values: none | circle() | ellipse() | polygon() | url() | [margin-box | padding-box | content-box]

Default: none

Applies to: floats

Inherits: no

FIGURE 15-12 shows the default text wrap around a floated image (left) and the same wrap with **shape-outside** applied (right). This is the kind of thing you'd expect to see in a print magazine, but now we can do it on the web!

It is worth noting that you can change the text wrap shape around any floated element (see **Note**), but I will focus on images in this discussion, as text elements are generally boxes that fit nicely in the default rectangular wrap.

There are two approaches to making text wrap around a shape. One way is to provide the path coordinates of the wrap shape with **circle()**, **ellipse()**, or **polygon()**. Another way is to use **url()** to specify an image that has transparent areas (such as a GIF or a PNG). With the image method, text flows into the transparent areas of the image and stops at the opaque areas. This is the shape method shown in FIGURE 15-12 and the method I'll introduce first.

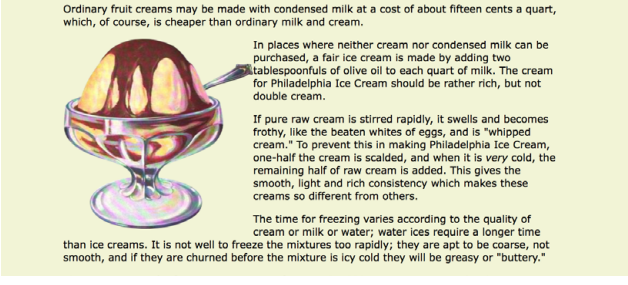
BROWSER SUPPORT NOTE

As of this writing in 2018, text wrap shapes are supported only by Chrome 37+, Opera 24+, Safari 7.1+ (with prefix; without starting in 10.1), iOS Safari 8+ (with prefix; without in 10.3+), and Android 56+. The feature is under consideration at Microsoft Edge and in development at Firefox, so the support situation may be better by the time you are reading this. Check CanIUse.com for the current state of support.

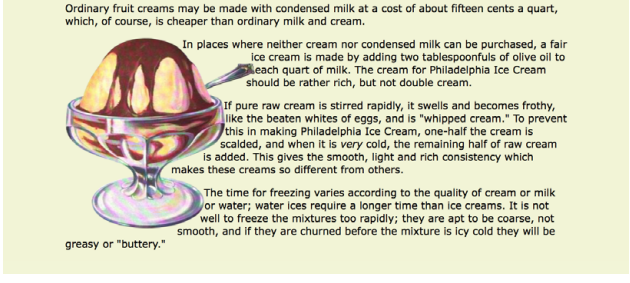
*For the time being, feel free to use it as a progressive enhancement for designs in which a rectangular text wrap would be perfectly acceptable. Another alternative is to use a feature query (***@supports***) to serve a fallback set of styles to non-supporting browsers. Feature queries are introduced in* **Chapter 19, More CSS Techniques**.

NOTE

shape-outside *works only on floated elements for now, but it is believed that this will change in the future.*



Default text wrap



Text wrap with **shape-outside** using the transparent areas of the image as a guide

FIGURE 15-12. Example of text wrapping around an image with **shape-outside**.

Using a Transparent Image

In the example in FIGURE 15-12, I placed the *sundae.png* image in the HTML document to display on the page, and I've specified the same image in the style rule using **url()** so that its transparent areas define the wrap shape (see important Warning). It makes sense to use the same image in the document and for the CSS shape, but it is not required. You could apply a wrap shape derived from one image to another image on the page.

THE MARKUP

```
<p><img src="sundae.png" class="wrap" alt=""> In places…</p>
```

THE STYLES

```
img.wrap {
  float: left;
  width: 300px;
  height: 300px;
  -webkit-shape-outside: url(sundae.png); /* prefix required in 2018 */
  shape-outside: url(sundae.png);
```

Notice that the wrapped text is now bumping right into the image. How about we give it a little extra space with **shape-margin**?

shape-margin

Values:	*length* | *percentage*
Default:	0
Applies to:	floats
Inherits:	no

The **shape-margin** property specifies an amount of space to hold between the shape and the wrapped text. In FIGURE 15-13, you can see the effect of adding 1em of space between the opaque image areas and the wrapped text lines. It gives it a little breathing room the way any good margin should.

```
-webkit-shape-margin: 1em;
shape-margin: 1em;
```

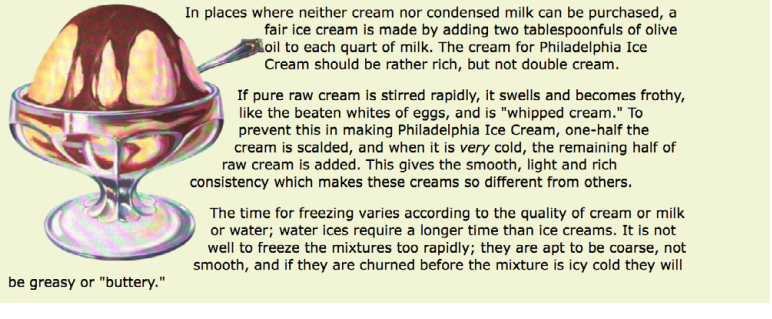


FIGURE 15-13. Adding a margin between the shape and the wrapped text.

WARNING

There is a security setting in Chrome and Opera that makes image-based text wraps a little tricky to use. Without getting into too much sys-admin detail, the browser restricts the use of the image used to create the CSS shape if it isn't on the same domain as the file requesting it. This is not a bug; they are following the rules set out in the specification.

The rule also means that compliant browsers won't allow images to be used for shapes when the files are served locally (i.e., on your computer). They need to be uploaded to a server to work, which makes the design process a little more cumbersome, especially for beginners.

If you use image-based text wraps, you know your CSS is written correctly, but you aren't seeing wrapping in the browser, this security setting (related to Cross-Origin Resource Sharing, or CORS, if you're curious) is probably the culprit.

Opacity Threshold

If you have a source image with multiple levels of transparency, such as the gradient shadow, the **shape-image-threshold** property allows text to creep into the image but stop when it encounters a specific transparency level. The value of this property is a number between 0 and 1, representing a percentage of transparency. For example, if you set the threshold to .2, text will wrap into areas that are up to 20% transparent, but stop when it gets to more opaque levels.

Using a Path

The other method for creating a text wrap shape is to define it using one of the path keywords: **`circle()`**, **`ellipse()`**, and **`polygon()`**.

The **`circle()`** notation creates a circle shape for the text to wrap around. The value provided within the parentheses represents the length of the radius of the circle:

```
circle(radius)
```

In this example, the radius is 150px, half of the image width of 300 pixels. By default, the circle is centered vertically and horizontally on the float:

```
img.round {
  float: left;
  -webkit-shape-outside: circle(150px);
  shape-outside: circle(150px);
}
```

FIGURE 15-14 shows this style rule applied to different images. Notice that the transparency of the image is not at play here. It's just a path overlaid on the image that sets the boundaries for text wrap. Any path can be applied to any image or other floated element.

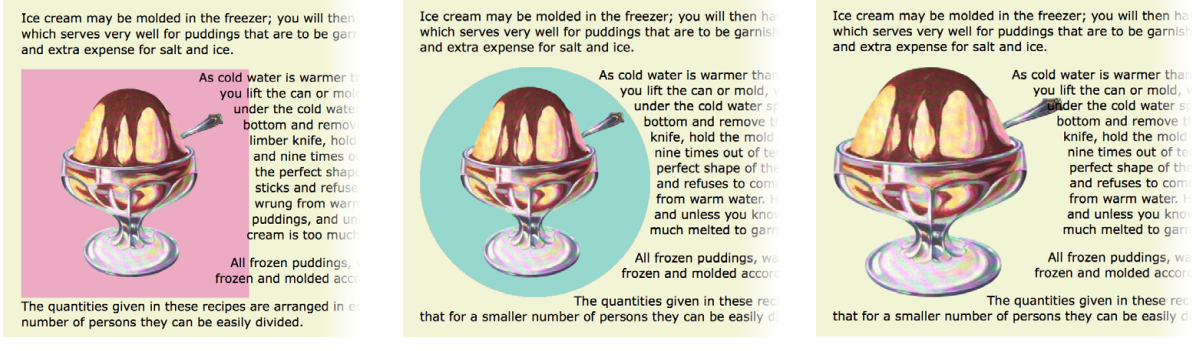
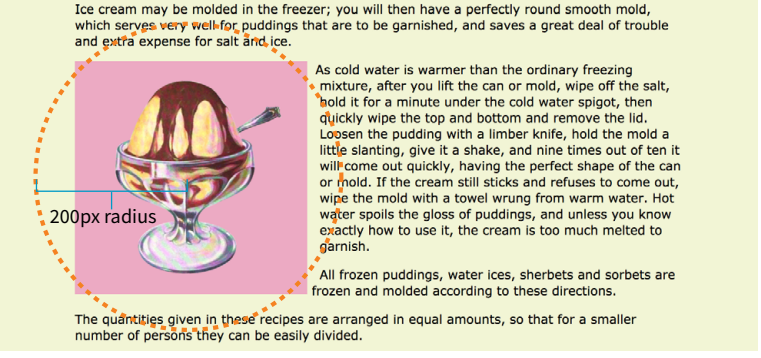


FIGURE 15-14. The same **`circle()`** shape applied to different images in the source.

This is a good point to demonstrate a critical behavior of wrap shapes. They allow text to flow *into* the floated image or element, but they cannot hold space free beyond it.

In the example in FIGURE 15-15, I've increased the diameter of the circle path from 150px to 200px. Notice that the text lines up along the right edge of the image, even though the circle is set 50 pixels beyond the edge. The path does not push text away from the float. If you need to keep wrapped text away from the outside edge of the floated image or element, apply a margin to the element itself (it will be the standard rectangular shape, of course).

```
img.round {
  float: left;
  -webkit-shape-outside: circle(200px);
  shape-outside: circle(200px);
}
```



A CSS shape allows text to wrap into floated elements, but does not push text away from them.

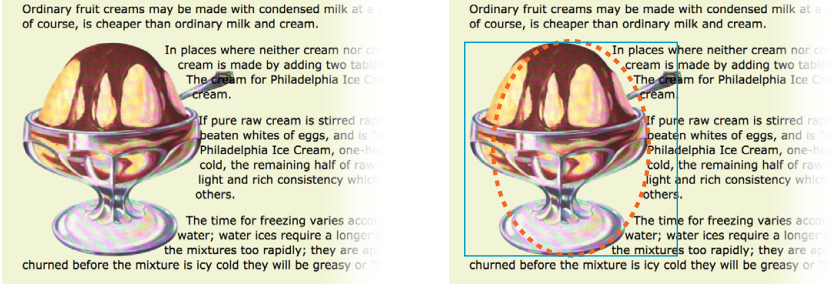
FIGURE 15-15. CSS shapes allow text to wrap into the floated element but do not hold space beyond it.

Elliptical shapes are created with the **`ellipse()`** notation, which provides the horizontal and vertical radius lengths followed by the word **`at`** and then the x,y coordinates for the center of the shape. Here is the syntax:

```
ellipse(rx ry at x y);
```

The position coordinates can be listed as a specific measurement or a percentage. Here I've created an ellipse with a 100-pixel horizontal radius and a 150-pixel vertical radius, centered in the floated element it is applied to (FIGURE 15-16):

```
img.round {
  float: left;
  -webkit-shape-outside: ellipse(200px 100px at 50% 50%);
  shape-outside: ellipse(200px 100px at 50% 50%);
}
```



The edges of the image (blue) and ellipse path (dotted orange) revealed

FIGURE 15-16. An elliptical text wrap created with **`ellipse()`**.

Finally, we come to **polygon()**, which lets you create a custom path using a series of comma-separated x,y coordinates along the path. This style rule creates the wrap effect shown in FIGURE 15-17:

```
img.wrap {
  float: left;
  width: 300px;
  height: 300px;
  shape-outside: polygon(0px 0px, 186px 0px, 225px 34px, 300px 34px,
300px 66px, 255px 88px, 267px 127px, 246px 178px, 192px 211px, 226px
236px, 226px 273px, 209px 300px, 0px 300px);
}
```

The edges of the image (blue) and polygon path (dotted orange) revealed

FIGURE 15-17. A custom path created with **polygon()**.

Holy coordinates! That's a lot of numbers, and my path was fairly simple. I'd like to be able to point you to a great tool for drawing and exporting polygon paths, but sadly, as of this writing I have none to recommend (see **Note**). I got the coordinates for my polygon examples by opening the image in Photoshop and gathering them manually, which, although possible, is not ideal.

CSS Shapes Resources

There are some finer points regarding CSS Shapes that I must leave to you to research further. Here are a few resources to get you started:

- CSS Shapes Module, Level 1 (*www.w3.org/TR/css-shapes-1/*)
- "Getting Started with CSS Shapes" by Razvan Caliman (*www.html5rocks.com/en/tutorials/shapes/getting-started*)
- CSS Shapes at the Experimental Layout Lab of Jen Simmons (*labs.jensimmons.com/#shapes*)
- "A Redesign with CSS Shapes" by Eric Meyer (*alistapart.com/article/redesign-with-css-shapes*)

Why don't we make the text wrap around the images in the Black Goose Bakery page in a more interesting way for users with browsers that support it (EXERCISE 15-2)?

NOTE

A CSS Shapes Editor will be included in a future version of Firefox that will likely be available by the time you are reading this (developer.mozilla.org/en-US/docs/Tools/Page_Inspector/How_to/Edit_CSS_shapes).

■ WEB SEARCH TIP

If you search for "CSS Shapes" you will certainly come across that term used for a technique that uses CSS to draw geometric shapes such as triangles, arrows, circles, and so on. It's a little confusing, although those other "CSS shapes" are pretty nifty and something you might want to tinker with. I introduce them briefly in **Chapter 23, Web Image Basics**.

EXERCISE 15-2. Adding shapes around floats

The bread and muffin images on the Black Goose Bakery page provide a nice opportunity to try out CSS Shapes. You will need to use a supporting browser such as a recent version of Chrome, Safari, or Opera to see the wrapping effect.

Open the latest version of the bakery style sheet and look for the section labeled **/* main "products" styles */**. We'll put the image wrap styles there to keep our style sheet organized.

Target each image individually using an attribute selector (there is one set up for "muffin" already). Start out simply and make the text wrap around a circle. Set the radius of the circle to 125px for the bread image and 110px for the muffin.

```
img[src*="bread"] {
  -webkit-shape-outside: circle(125px);
  shape-outside: circle(125px);
}
img[src*="muffin"] {
  margin-left: 50px;
  -webkit-shape-outside: circle(110px);
  shape-outside: circle(110px);
}
```

Save the styles and take a look at the page in a supporting browser. The circles look pretty good, but I think I could improve the wrap around the bread by making it an ellipse. Add these after the circle declarations, and the ellipse wrap will override the previous styles (or delete and replace):

```
img[src*="bread"] {
  -webkit-shape-outside: ellipse(130px 95px at 50% 50%);
  shape-outside: ellipse(130px 95px at 50% 50%);
}
```

If you're feeling ambitious, you could add a polygon wrap shape around the muffin image instead of the circle. You'll need to copy these coordinates or just copy and paste from the finished exercise provided in the materials for this chapter. Or just stick with the circle, and nobody will judge you.

```
img[src*="muffin"] {
…
shape-outside: polygon(0px 0px, 197px 0px, 241px 31px, 241px 68px, 226px 82px, 219px 131px, 250px 142px, 250px 158px, 0px 158px);
}
```

The final result is shown in FIGURE 15-18. It is most apparent when the browser window is sufficiently narrow that enough lines wrap to reveal the shape. For browsers that don't support shapes, the rectangular whitespace is just fine.

FIGURE 15-18. The bakery page with text wrapping around images in an ellipse (bread) and polygon (muffin) using CSS Shapes.

Well, that covers floating! You've learned how to float elements left and right, clear the following elements so they start below the floats, and even make fancy text wrapping shapes. Now let's move on to the other approach to moving elements around on the page—positioning.

POSITIONING BASICS

CSS provides several methods for positioning elements on the page. They can be positioned relative to where they would normally appear in the flow, or removed from the flow altogether and placed at a particular spot on the page. You can also position an element relative to the viewport, and it will stay put while the rest of the page scrolls.

Types of Positioning

`position`

Values: `static` | `relative` | `absolute` | `fixed`

Default: `static`

Applies to: all elements

Inherits: no

The **`position`** property indicates that an element is to be positioned and specifies which positioning method to use. I'll introduce each keyword value briefly here, and then we'll take a more detailed look at each method in the remainder of this chapter.

`static`

This is the normal positioning scheme in which elements are positioned as they occur in the normal document flow.

`relative`

Relative positioning moves the element box relative to its original position in the flow. The distinctive behavior of relative positioning is that the space the element would have occupied in the normal flow is preserved as empty space.

`absolute`

Absolutely positioned elements are removed from the document flow entirely and positioned with respect to the viewport or a containing element (we'll talk more about this later). Unlike relatively positioned elements, the space they would have occupied is closed up. In fact, they have no influence at all on the layout of surrounding elements.

■ TERMINOLOGY

Viewport

I'll be sticking with the more formal term *viewport* throughout the positioning discussions, but keep in mind it could be a browser window on a desktop computer, the full screen of a mobile device, or the frame of an **`iframe`** element from the perspective of the web page loaded in that frame. It is any space that visually displays a web page.

`fixed`

The distinguishing characteristic of fixed positioning is that the element stays in one position in the viewport even when the document scrolls. Fixed elements are removed from the document flow and positioned relative to the viewport rather than another element in the document.

`sticky`

Sticky positioning is a combination of relative and fixed in that it behaves as though it is relatively positioned, until it is scrolled into a specified position relative to the viewport, at which point it remains fixed.

The MDN Web Docs site has this description for a potential use case:

> *Sticky positioning is commonly used for the headings in an alphabetized listing. The B heading will appear just below the items that begin with A until they are scrolled offscreen. Rather than sliding offscreen with the rest of the content, the B heading will then remain fixed to the top of the viewport until all the B items have scrolled offscreen, at which point it will be covered up by the C heading.*

The **`sticky`** position value is supported by current versions of Chrome, Firefox, Opera, MS Edge, Android, as well as Safari and iOS Safari with the **`-webkit-`** prefix. No version of IE supports it. Happily, **`sticky`** positioning degrades gracefully, as the element simply stays inline and scrolls with the document if it is not supported.

Each positioning method has its purpose, but absolute positioning is the most versatile. With absolute positioning, you can place an object anywhere in the viewport or within another element. Absolute positioning has been used to create multicolumn layouts, but it is more commonly used for small tasks, like positioning a search box in the top corner of a header. It's a handy tool when used carefully and sparingly.

Specifying Position

Once you've established the positioning method, the actual position is specified with some combination of up to four offset properties.

`top`, `right`, `bottom`, `left`

Values: *length* | *percentage* | `auto`

Default: `auto`

Applies to: positioned elements (where `position` value is `relative`, `absolute`, or `fixed`)

Inherits: no

The value provided for each offset property defines the distance the element should be moved *away* from that respective edge. For example, the value of **`top`** defines the distance the top outer edge of the positioned element should be offset from the top edge of the browser or other containing element. A

positive value for **top** results in the element box moving *down* by that amount (see **Note**). Similarly, a positive value for **left** would move the positioned element to the right (toward the center of the containing block) by that amount.

Further explanations and examples of the offset properties will be provided in the discussions of each positioning method. We'll start our exploration of positioning with the fairly straightforward **relative** method.

NOTE

Negative values are acceptable and move the element in the opposite direction of positive values. For example, a negative value for **top** *would have the effect of moving the element up.*

RELATIVE POSITIONING

As mentioned previously, relative positioning moves an element relative to its original spot in the flow. The space it would have occupied is preserved and continues to influence the layout of surrounding content. This is easier to understand with a simple example.

Here I've positioned an inline **em** element. A bright background color on the **em** and a border on the containing paragraph make their boundaries apparent. First, I used the **position** property to set the method to **relative**, and then I used the **top** offset property to move the element 2em down from its initial position, and the **left** property to move it 3em to the right. Remember, offset property values move the element *away* from the specified edge, so if you want something to move to the right, as I did here, you use the **left** offset property. The results are shown in FIGURE 15-19.

When an element is relatively positioned, the space it once occupied is preserved.

```
em {
  position: relative;
  top: 2em; /* moves element down */
  left: 3em; /* moves element right */
  background-color: fuchsia;
}
```

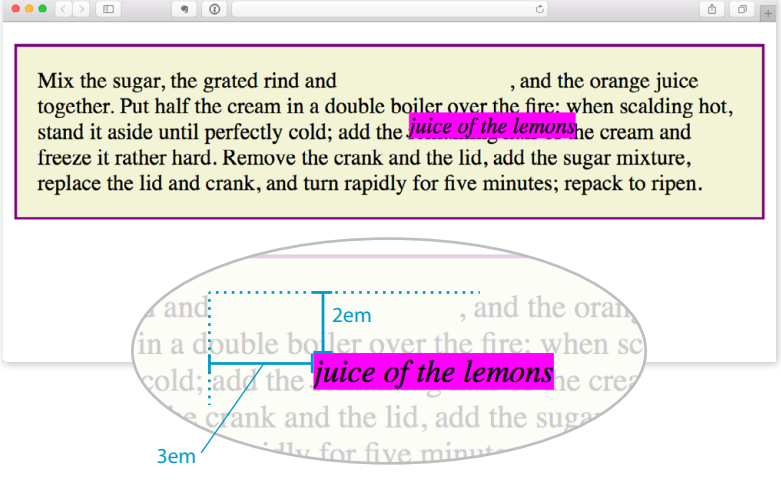


FIGURE 15-19. When an element is positioned with the relative method, the space it would have occupied is preserved.

I want to point out a few things that are happening here:

The original space in the document flow is preserved.

You can see that there is a blank space where the emphasized text would have been if the element had not been positioned. The surrounding content is laid out as though the element were still there, and therefore we say that the element still "influences" the surrounding content.

Overlap happens.

Because this is a positioned element, it can potentially overlap other elements, as happens in FIGURE 15-19.

The empty space left behind by relatively positioned objects can be a little awkward, so this method is not used as often as absolute positioning. However, relative positioning is commonly used to create a "positioning context" for an absolutely positioned element. Remember that term *positioning context*—I'll explain it in the next section.

Columns with Absolute Positioning

Like floats, absolute positioning can be used to create columned layouts. These days, columned layouts should be created with CSS Grid, but you may use positioned columns as a fallback if you need to support old browsers that don't support Grid.

Should you like to learn how absolute positioning can be used for whole page layout, I have included instructions and templates in the supplemental article **"Page Layout with Floats and Positioning"** (PDF), available at *learningwebdesign.com/articles/*.

ABSOLUTE POSITIONING

Absolute positioning works a bit differently and is a more flexible method for accurately placing items on the page than relative positioning.

Now that you've seen how relative positioning works, let's take the same example as shown in FIGURE 15-19, only this time we'll change the value of the **`position`** property to **`absolute`** (FIGURE 15-20):

```
em {
  position: absolute;
  top: 2em;
  left: 3em;
  background-color: fuchsia;
}
```

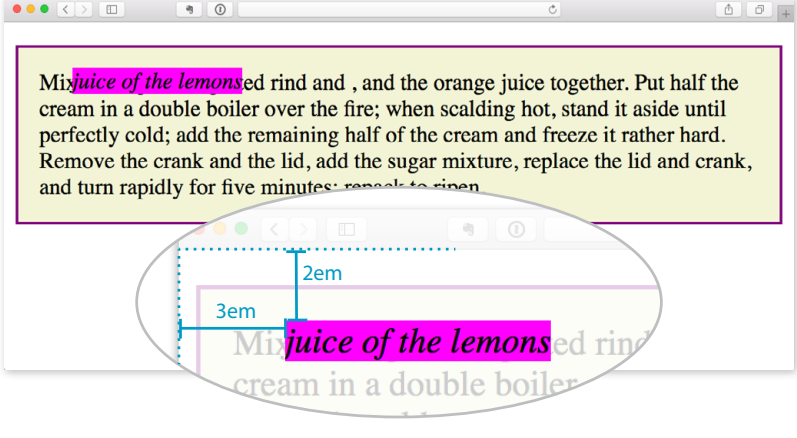


FIGURE 15-20. When an element is absolutely positioned, it is removed from the flow and the space is closed up.

As you can see in FIGURE 15-20, the space once occupied by the `em` element is now closed up, as is the case for all absolutely positioned elements. In its new position, the element box overlaps the surrounding content. In the end, absolutely positioned elements have no influence whatsoever on the layout of surrounding elements.

The most significant difference here, however, is the location of the positioned element. This time, the offset values position the `em` element 2em down and 3em to the right of the top-left corner of the *viewport* (browser window).

When an element is absolutely positioned, the space it once occupied is closed up.

But wait—before you start thinking that absolutely positioned elements are always placed relative to the viewport, I'm afraid that there's more to it than that.

What actually happens in absolute positioning is that the element is positioned relative to its nearest *containing block*. It just so happens that the nearest containing block in FIGURE 15-20 is the root (`html`) element, also known as the initial containing block, so the offset values position the `em` element relative to the whole document.

Getting a handle on the containing block concept is the first step to tackling absolute positioning.

Containing Blocks

The CSS Positioned Layout Module, Level 3, states, "The position and size of an element's box(es) are sometimes computed relative to a certain rectangle, called the containing block of the element." It is critical to be aware of the containing block of the element you want to position. We sometimes refer to this as the positioning context.

The spec lays out a number of intricate rules for determining the containing block of an element, but it basically boils down to this:

- If the positioned element is *not* contained within another positioned element, then it will be placed relative to the initial containing block (created by the `html` element).
- But if the element has an ancestor (i.e., is contained within an element) that has its position set to `relative`, `absolute`, or `fixed`, the element will be positioned relative to the edges of *that* element instead.

FIGURE 15-20 is an example of the first case: the `p` element that contains the absolutely positioned `em` element is *not* positioned itself, and there are no other positioned elements higher in the hierarchy. Therefore, the `em` element is positioned relative to the initial containing block, which is equivalent to the viewport area.

Let's deliberately turn the `p` element into a containing block and see what happens. All we have to do is apply the `position` property to it; we don't have to actually move it. The most common way to make an element into a containing block is to set its `position` to `relative`, but not move it with any

offset values. This is what I was talking about earlier when I said that relative positioning is used to create a *positioning context* for an absolutely positioned element.

In this example, we'll keep the style rule for the **em** element the same, but we'll add a **position** property to the **p** element, thus making it the containing block for the positioned **em** element. FIGURE 15-21 shows the results.

```
p {
  position: relative;
  padding: 15px;
  background-color: #F2F5D5;
  border: 2px solid purple;
}
```

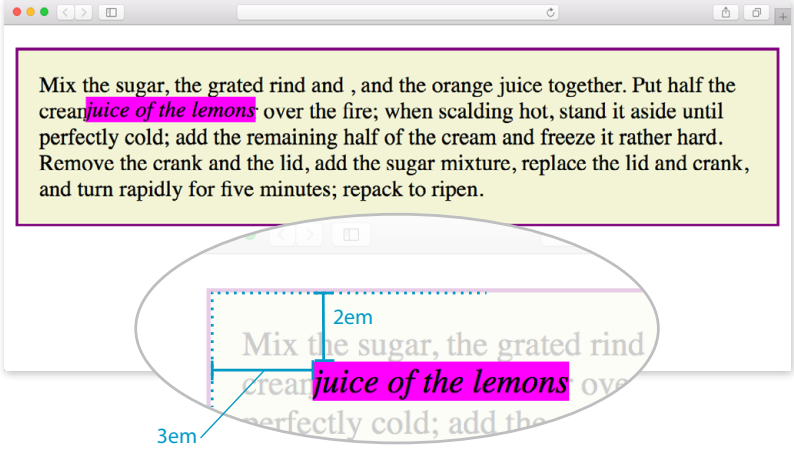


FIGURE 15-21. The relatively positioned **p** element acts as a containing block for the **em** element.

You can see that the **em** element is now positioned 2em down and 3em from the top-left corner of the paragraph box, not the browser window. Notice also that it is positioned relative to the *padding edge* of the paragraph (just inside the border), not the content area edge. This is the normal behavior when block elements are used as containing blocks (see **Note**).

NOTE

When inline elements are used as containing blocks (and they can be), the positioned element is placed relative to the content area edge, not the padding edge.

I'm going to poke around at this some more to reveal additional aspects of absolutely positioned objects. This time, I've added **width** and **margin** properties to the positioned **em** element (FIGURE 15-22):

```
em {
  width: 200px;
  margin: 25px;
  position: absolute;
  top: 2em;
  left: 3em;
  background-color: fuchsia;
}
```

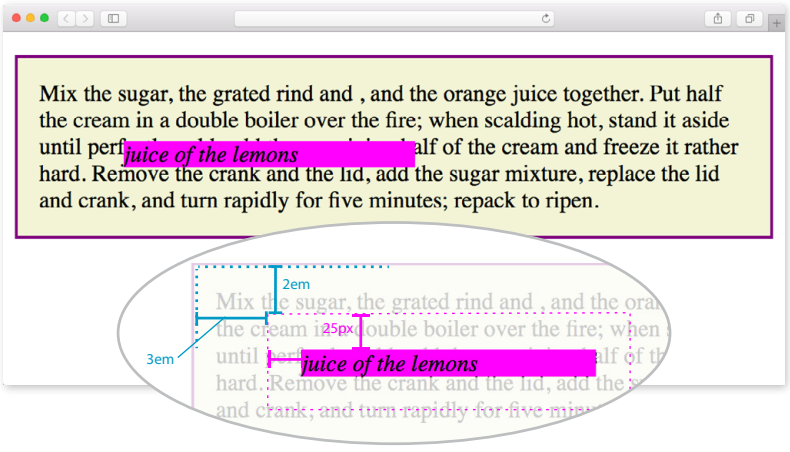


FIGURE 15-22. Adding a width and margins to the positioned element.

Here we can see that:

- The offset values apply to the outer edges of the element box (the outer margin edge) for absolutely positioned elements (see **Note**).
- Absolutely positioned elements always behave as block-level elements. For example, the margins on all sides are maintained, even though this is an inline element. It also permits a width to be set for the element.

NOTE

For relatively positioned elements, the offset is measured to the box itself (not the outer margin edge).

It is important to keep in mind that once you've positioned an element, it becomes the new containing block for all the elements it contains. Say you position a narrow `div` at the top-left corner of a page, creating a column. If you were to absolutely position an image within that `div` with offset values that place it in the top-right corner, it appears in the top-right corner of that `div`, not the entire page. Once the parent element is positioned, it acts as the containing block for the `img` and any other contained elements.

Specifying Position

Now that you have a better feel for the containing block concept, let's take some time to get better acquainted with the offset properties. So far, we've only seen an element moved a few ems down and to the right, but that's not all you can do, of course.

Pixel measurements

As mentioned previously, positive offset values push the positioned element box *away* from the specified edge and toward the center of the containing block. If there is no value provided for a side, it is set to `auto`, and the browser adds enough space to make the layout work. In this example, `div#B` is contained within `div#A`, which has been given the dimensions 600 pixels wide by 300 pixels high. I've used pixel lengths for all four offset properties to place

the positioned element (#B) at a particular spot in its containing element (#A) (FIGURE 15-23).

THE MARKUP

```
<div id="A">
  <div id="B"> </div>
</div>
```

THE STYLES

```
div#A {
  position: relative;   /* creates the containing block */
  width: 600px;
  height: 300px;
  background-color: #C6DE89; /* green */
}

div#B {
  position: absolute;
  top: 25px;
  right: 50px;
  bottom: 75px;
  left: 100px;
  background-color: steelblue;
}
```

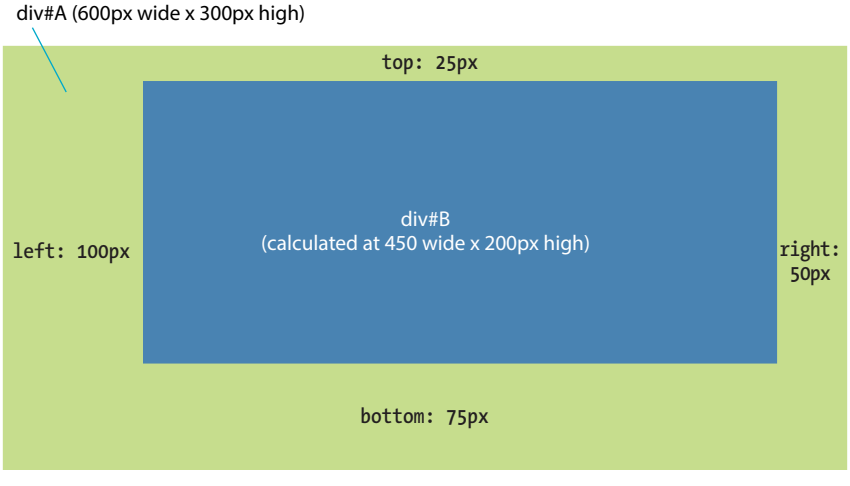


FIGURE 15-23. Setting offset values for all four sides of a positioned element.

Notice that by setting offsets on all four sides, I have indirectly set the dimensions of the positioned **div#B.** It fills the 450 × 200 pixel space that is left over within the containing block after the offset values are applied. If I had also specified a width and other box properties for **div#B**, there is the potential for conflicts if the total of the values for the positioned box and its offsets does not match the available space within the containing block.

The CSS specification provides a daunting set of rules for handling conflicts, but the upshot is that you should just be careful not to over-specify box properties and offsets. In general, a width (factoring in margins as well as padding and border if you are using the **content-box** box-sizing model) and one or two offset properties are all that are necessary to achieve the layout you're looking for. Let the browser take care of the remaining calculations.

Percentage values

You can also specify positions with percentage values. In the first example in FIGURE 15-24, the image is positioned halfway (50%) down the left edge of the containing block. In the second example on the right, the **img** element is positioned so that it always appears in the bottom-right corner of the containing block.

```
img#A {
  position: absolute;
  top: 50%;
  left: 0%;
}
img#B {
  position: absolute;
  bottom: 0%;
  right: 0%;
}
```

Although the examples here specify both a vertical and horizontal offset, it is common to provide just one offset for a positioned element—for example, to move it left or right into a margin using either **left** or **right** properties.

In EXERCISE 15-3, we'll make further changes to the Black Goose Bakery home page, this time using absolute positioning.

WARNING

Be careful when positioning elements at the bottom of the initial containing block (the **html** *element). Although you may expect it to be positioned at the bottom of the whole page, browsers actually place the element at the bottom of the browser window. Results may be unpredictable. If you want something positioned at the bottom of your page, put it in a containing block element at the end of the document source, and go from there.*

NOTE

The % symbol could be omitted for a 0 value, which essentially turns it into a 0 length but achieves an equivalent result.

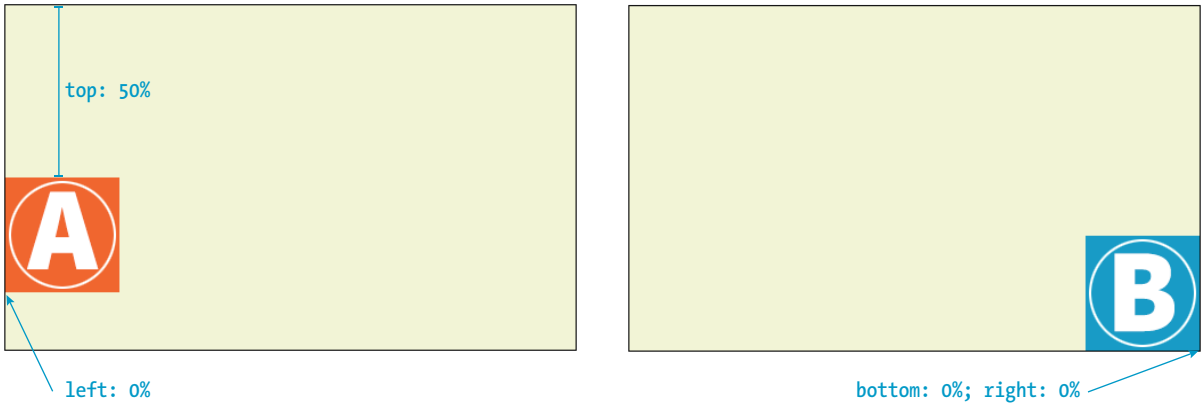


FIGURE 15-24. Using percentage values to position an element in a containing block.

EXERCISE 15-3. Absolute positioning

In this exercise, we'll use absolute positioning to add an award graphic to the home page. Open the version of the site you saved in EXERCISE 15-2.

1. Good news! Black Goose Bakery won the Farmers' Market Award, and we have the privilege of displaying an award medal on the home page. Because this is new content, we'll need to add it to the markup in *bakery.html*. Because it is nonessential information, add the image in a new **div** in the **footer** of the document:

```
<footer>
  <p>All content copyright &copy; 2017, Black
Goose Bistro.</p>
  <div id="award"><img src="images/award.png"
alt="Farmers' Market Award"></div>
</footer>
```

2. Just because the award is at the end of the source document doesn't mean it needs to display there. We can use absolute positioning to place the award in the top-left corner of the viewport by adding a new rule to the style sheet that positions the **div**, like so (I put mine in the **/* misc styles */** section):

```
#award {
  position: absolute;
  top: 30px;
  left: 50px;
}
```

Save the document and take a look (FIGURE 15-25). Resize the browser window very narrow, and you will see that the positioned award image overlaps the header content. Notice also that when you scroll the document, the image scrolls with the rest of the page. Try playing around with other offset properties to get a feel for positioning in the viewport (or the "initial containing block" to be precise).

P.S. I know that the navigation list still looks bad, but we'll fix it up in **Chapter 16**.

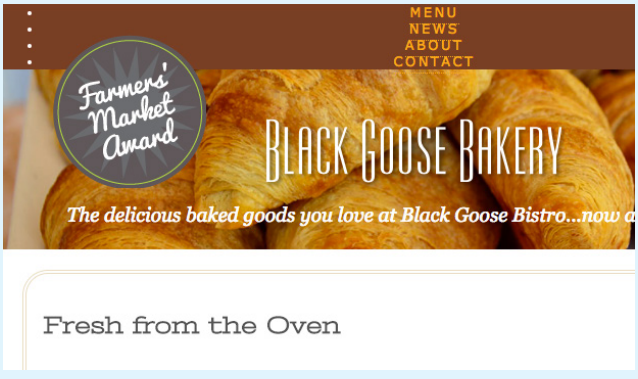


FIGURE 15-25. An absolutely positioned award graphic.

Stacking Order

Before we close the book on absolute positioning, there is one last related concept that I want to introduce. As we've seen, absolutely positioned elements overlap other elements, so it follows that multiple positioned elements have the potential to stack up on one another.

By default, elements stack up in the order in which they appear in the document, but you can change the stacking order with the **z-index** property (see **Note**). Picture the z-axis as a line that runs perpendicular to the page, as though from the tip of your nose, through this page, and out the other side.

NOTE

The **z-index** *property is also useful for items in a grid, which also have the potential to overlap, as discussed in* **Chapter 16**.

z-index

Values: *number* | auto

Default: auto

Applies to: positioned elements

Inherits: no

The value of the **z-index** property is a number (positive or negative). The higher the number, the higher the element will appear in the stack (that is, closer to your nose). Lower numbers and negative values move the element lower in the stack. Let's look at an example to make this clear (FIGURE 15-26).

Here are three paragraph elements, each containing a letter image (A, B, and C, respectively) that have been absolutely positioned in such a way that they overlap on the page. By default, paragraph C would appear on top because it appears last in the source. However, by assigning higher **z-index** values to paragraphs A and B, we can force them to stack in our preferred order.

Note that the values of **z-index** do not need to be sequential, and they do not relate to anything in particular. All that matters is that higher number values position the element higher in the stack.

THE MARKUP

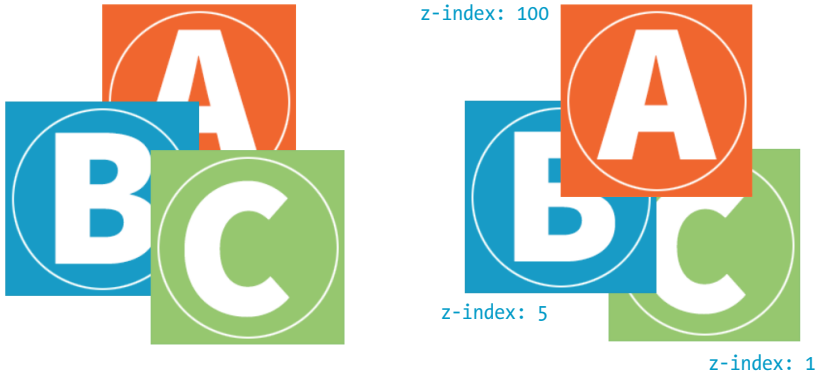
```
<p id="A"><img src="A.gif" alt="A"></p>
<p id="B"><img src="B.gif" alt="B"></p>
<p id="C"><img src="C.gif" alt="C"></p>
```

THE STYLES

```
#A {
  z-index: 100;
  position: absolute;
  top: 175px;
  left: 200px;
}

#B {
  z-index: 5;
  position: absolute;
  top: 275px;
  left: 100px;
}

#C {
  z-index: 1;
  position: absolute;
  top: 325px;
  left: 250px;
}
```



By default, elements later in the document source order stack on top of preceding elements.

You can change the stacking order with the z-index property. Higher values stack on top of lower values.

FIGURE 15-26. Changing the stacking order with the **z-index** property.

To be honest, the **z-index** property is not often required for most page layouts, but you should know it's there if you need it. If you want to guarantee that a positioned element always ends up on top, assign it a very high **z-index** value, such as 100 or 1000. If you want to make sure it's at the bottom, give it a negative value. The number itself doesn't actually matter.

FIXED POSITIONING

We've covered relative and absolute positioning, so now it's time to take on fixed positioning.

For the most part, fixed positioning works just like absolute positioning. The significant difference is that the offset values for fixed elements are *always* relative to the viewport, which means the positioned element stays put even when the rest of the page scrolls. By contrast, you may remember that when you scrolled the Black Goose Bakery page in EXERCISE 15-3, the award graphic scrolled along with the document—even though it was positioned relative to the initial containing block (equivalent to the viewport). Not so with fixed positioning, where the position is, well, *fixed*.

Fixed elements are often used for menus that stay in the same place at the top, bottom, or side of a screen so they are always available, even when the content scrolls (see **Warning**). Bear in mind that if you fix an element to the bottom of the viewport, you'll need to leave enough space at the end of the document so the content doesn't get hidden behind the fixed element. Fixed elements are also problematic when the document is printed because they will print on every page without reserving any space for themselves. It's best to turn off fixed elements when printing the document. (Targeting print with `@media` is addressed in **Chapter 17, Responsive Web Design**.)

Let's switch the award graphic on the Black Goose Bakery page to a fixed position in EXERCISE 15-4 to see the difference.

WARNING

The `position: fixed` *property causes some buggy behaviors on old versions of mobile Safari (5, 6, and 7) and Android (<4.4). Fortunately, these mobile browsers are nearly obsolete as of this writing, but it is a reminder to do thorough testing on a range of mobile devices if you have fixed elements.*

EXERCISE 15-4. Fixed positioning

This should be simple. Open the bakery style sheet as you left it in EXERCISE 15-3 and edit the style rule for the `#award div` to make it `fixed` rather than `absolute`:

```
#award {
  position: fixed;
  top: 30px;
  left: 50px;
}
```

Save the styles and open the page in a browser. When you scroll the page, you will see that the award now stays put where we positioned it in the browser window (FIGURE 15-27). You can see that fixed positioned elements have the potential to hide content as the page scrolls. Test well to see the potential pitfalls and weigh them against the benefits.

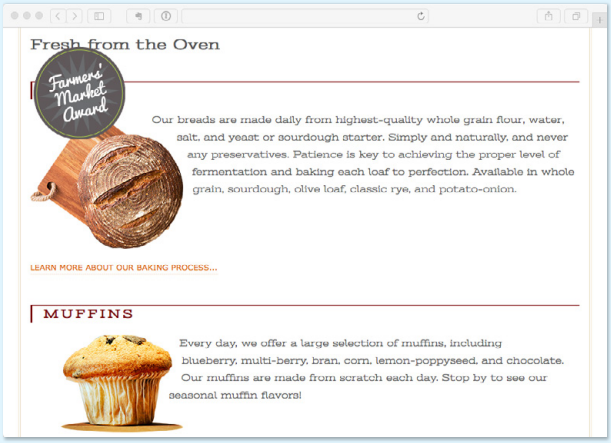


FIGURE 15-27. The award stays in the same place in the top-left corner of the browser when the document scrolls.

That does it for floating and positioning. In the next chapter, you'll learn about flexible boxes and grid layout, which are powerful tools for designing the overall structure of a page and specific page features. But first, try your hand at a few questions about floating and positioning.

TEST YOURSELF

Before we move on, take a moment to see how well you absorbed the principles in this chapter. You'll find the answers in **Appendix A**.

1. Which of the following is *not* true of floated elements?
   a. All floated elements behave as block elements.
   b. Floats are positioned against the padding edge of the containing element.
   c. The contents of inline elements flow around a float, but the element box is unchanged.
   d. You must provide a `width` property for floated block elements.

2. Which of these style rules is incorrect? Why?
   a. `img { float: left; margin: 20px;}`
   b. `img { float: right; width: 120px; height: 80px; }`
   c. `img { float: right; right: 30px; }`
   d. `img { float: left; margin-bottom: 2em; }`

3. How do you make sure a `footer` element always starts below any floated sidebars on the page?

4. Write the name of the positioning method or methods (static, relative, absolute, or fixed) that best matches each of the following descriptions.
   a. Positions the element relative to a containing block.
   b. Removes the element from the normal flow.
   c. Always positions the element relative to the viewport.
   d. The positioned element may overlap other content.

Continued...

e. Positions the element in the normal flow.

f. The space the element would have occupied in the normal flow is preserved.

g. The space the element would have occupied in the normal flow is closed up.

h. You can change the stacking order with **`z-index`**.

i. Positions the element relative to its original position in the normal flow.

CSS REVIEW: FLOATING AND POSITIONING PROPERTIES

Here is a summary of the properties covered in this chapter.

Property	Description
`clear`	Prevents an element from being laid out next to a float
`float`	Moves the element to the right or left and allows the following text to flow around it
`position`	Specifies the positioning method to be applied
`top`, `bottom`, `right`, `left`	Specifies the offset amount from each respective edge
`shape-outside`	Causes content to wrap around a shape instead of the float's bounding box.
`shape-margin`	Adds a margin to **`shape-outside`**
`shape-image-threshold`	Defines the alpha channel threshold used to create the wrap shape
`z-index`	Specifies the order of appearance within a stack of overlapping positioned elements

CHAPTER
16

CSS LAYOUT WITH FLEXBOX AND GRID

Get ready...this is a *whopper* of a chapter! In it, you will learn about two important CSS page layout tools:

- Flexbox for greater control over arranging items along one axis
- Grid for honest-to-goodness grid-based layouts, like those print designers have used for decades

Each tool has its special purpose, but you can use them together to achieve layouts we've only dreamed of until now. For example, you could create the overall page structure with a grid and use a flexbox to tame the header and navigation elements. Use each technique for what it's best suited for—you don't have to choose just one.

Now that browsers have begun to support these techniques, designers and developers have true options for achieving sophisticated layouts with baked-in flexibility needed for dealing with a wide array of screen sizes. Once old browsers fade from use, we can kiss our old float layout hacks goodbye (in the meantime, they make decent fallbacks).

You may notice that this chapter is big. *Really* big. That's because the specs are overflowing with options and new concepts that require explanation and examples. It's a lot to pack in your mind all at once, so I recommend treating it as two mini-chapters and spend some time getting up to speed with each technique individually.

IN THIS CHAPTER

FLEXIBLE BOXES WITH CSS FLEXBOX

The CSS Flexible Box Layout Module (also known as simply Flexbox) gives designers and developers a handy tool for laying out components of web pages such as menu bars, product listings, galleries, and much more.

According to the spec,

> *The defining aspect of flex layout is the ability to make the flex items "flex," altering their width/height to fill the available space in the main dimension.*

That means it allows items to stretch or shrink inside their containers, preventing wasted space and overflow—a real plus for making layouts fit a variety of viewport sizes. Other advantages include the following:

- The ability to make all neighboring items the same height
- Easy horizontal and vertical centering (curiously elusive with old CSS methods)
- The ability to change the order in which items display, independent of the source

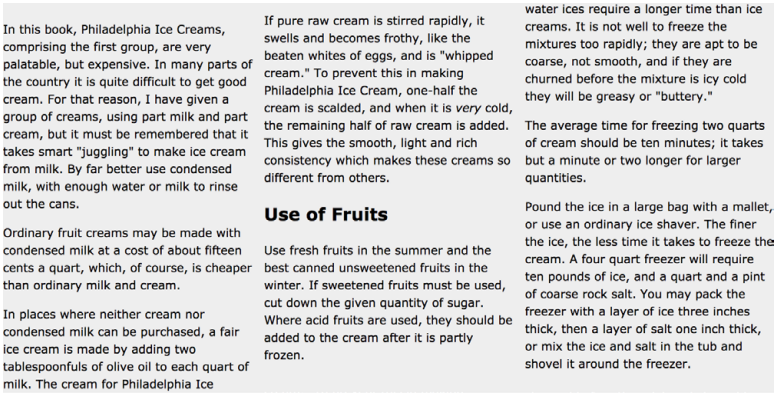
The Flexbox layout model is incredibly robust, but because it was designed for maximum flexibility, it takes a little time to wrap your head around it (at least it did for me). Here's how it helped me to think about it: when you tell an element to become a flexbox, all of its child elements line up on one axis, like beads on a string. The string may be horizontal, it may hang vertically, or it may even wrap onto multiple lines, but the beads are always on one string (or to use the proper term, one axis). If you want to line things up both horizontally *and* vertically, that is the job of CSS Grid, which I'll introduce in the next section of this chapter.

Before we dig in, I have a quick heads-up about browser support. All current browser versions support the latest W3C Flexible Box Layout Module spec; however, older browsers require prefixes and even different, outdated properties and values altogether. I'll be sticking with the current standard properties to keep everything simple while you learn this for the first time, but know

Multicolumn Layout

A third CSS3 layout tool you may want to try is multicolumn layout. The Multi-column Layout Module (*w3.org/TR/css-multicol-1*) provides tools for pouring text content into a number of columns, as you might see in a newspaper (FIGURE 16-1). It is designed to be flexible, allowing the widths and number of columns to automatically fit the available space.

This chapter is already big enough, so I've put this lesson in an article, **"Multicolumn Layout"** (PDF), available at *learningwebdesign.com/articles/*.



In this book, Philadelphia Ice Creams, comprising the first group, are very palatable, but expensive. In many parts of the country it is quite difficult to get good cream. For that reason, I have given a group of creams, using part milk and part cream, but it must be remembered that it takes smart "juggling" to make ice cream from milk. By far better use condensed milk, with enough water or milk to rinse out the cans.

Ordinary fruit creams may be made with condensed milk at a cost of about fifteen cents a quart, which, of course, is cheaper than ordinary milk and cream.

In places where neither cream nor condensed milk can be purchased, a fair ice cream is made by adding two tablespoonfuls of olive oil to each quart of milk. The cream for Philadelphia Ice

If pure raw cream is stirred rapidly, it swells and becomes frothy, like the beaten whites of eggs, and is "whipped cream." To prevent this in making Philadelphia Ice Cream, one-half the cream is scalded, and when it is *very* cold, the remaining half of raw cream is added. This gives the smooth, light and rich consistency which makes these creams so different from others.

Use of Fruits

Use fresh fruits in the summer and the best canned unsweetened fruits in the winter. If sweetened fruits must be used, cut down the given quantity of sugar. Where acid fruits are used, they should be added to the cream after it is partly frozen.

water ices require a longer time than ice creams. It is not well to freeze the mixtures too rapidly; they are apt to be coarse, not smooth, and if they are churned before the mixture is icy cold they will be greasy or "buttery."

The average time for freezing two quarts of cream should be ten minutes; it takes but a minute or two longer for larger quantities.

Pound the ice in a large bag with a mallet, or use an ordinary ice shaver. The finer the ice, the less time it takes to freeze the cream. A four quart freezer will require ten pounds of ice, and a quart and a pint of coarse rock salt. You may pack the freezer with a layer of ice three inches thick, then a layer of salt one inch thick, or mix the ice and salt in the tub and shovel it around the freezer.

FIGURE 16-1. An example of text formatted with the multicolumn properties.

that production-ready style sheets may require more code. I'll give you the nitty-gritty on browser support at the end of this section.

Setting Up a Flexbox Container

You've already learned about the block layout mode for stacking elements in the normal flow and the inline mode for displaying content within it horizontally. Flexbox is another layout mode with its own behaviors. To turn on flexbox mode for an element, set its **`display`** property to **`flex`** or **`inline-flex`** (see **Note**). It is now a flex container, and all of its direct child elements (whether they are **`div`**s, list items, paragraphs, etc.) *automatically* become flex items in that container. The flex items (the beads) are laid out and aligned along flex lines (the string).

FIGURE 16-2 shows the effect of simply adding **`display: flex`** to a **`div`**, thus turning on the Flexbox switch. I've added a blue border to the container to make its boundaries clear. To save space, I am not showing purely cosmetic styles such as colors and fonts.

NOTE

The **`inline-flex`** *value generates an inline-level flex container box. We'll be focusing on the more commonly used* **`flex`** *value in this chapter.*

THE MARKUP

```
<div id="container">
  <div class="box box1">1</div>
  <div class="box box2">2</div>
  <div class="box box3">3</div>
  <div class="box box4">4</div>
  <div class="box box5">5</div>
</div>
```

THE STYLES

```
#container {
  display: flex;
}
```

Flexbox Resources

You'll learn all the ins and outs of Flexbox in this chapter, but it is always good to get a few perspectives and hands-on tutorials online. If you do a web search, be sure to limit your findings to 2015 posts and later, or you may come across outdated advice based on earlier spec versions. Following are some of the sites that I've found most useful or most entertaining:

A Complete Guide to Flexbox (css-tricks.com/snippets/css/a-guide-to-flexbox/)

This summary of Flexbox features by Chris Coyier is one of the most popular Flexbox references out there. Many developers just keep it open in a browser when they do Flexbox work.

Flexbox Froggy (flexboxfroggy.com/)

Don't miss this online game for learning Flexbox by helping colorful frogs make it back to their lily pads.

What the Flexbox?! (flexbox.io/)

Wes Bos does a great job walking you through Flexbox properties as well as a few code projects in this free, 20-part video series.

Flexbox Playground (codepen.io/enxaneta/full/adLPwv/)

As the name says, this page by Gabi lets you play around with all of the Flexbox properties and values and see the results instantly. It's a nice way to get familiar with what Flexbox can do.

Flexy Boxes (the-echoplex.net/flexyboxes/)

This is another Flexbox playground and code generator.

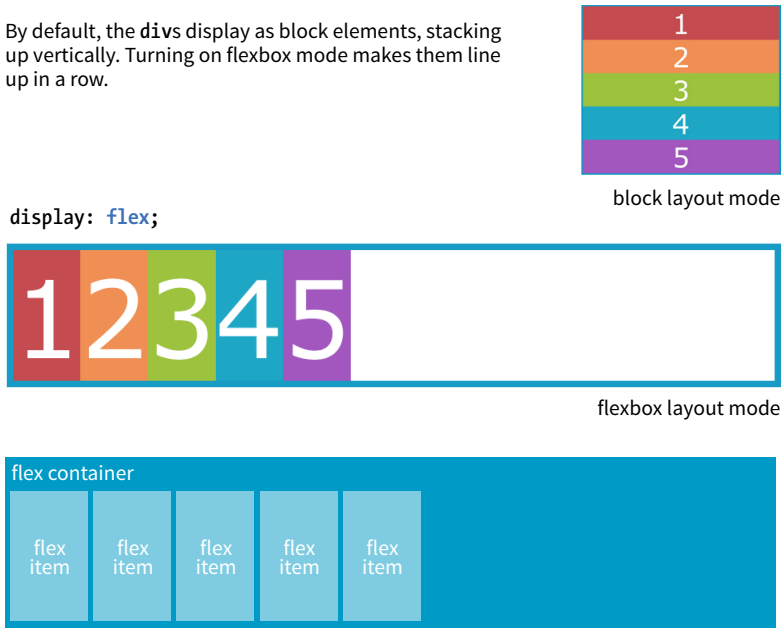


FIGURE 16-2. Applying the flex display mode turns the child elements into flex items that line up along one axis. You don't need to do anything to the child elements themselves.

■ FLEXBOX FUN FACTS

Here are a few things to know about Flexbox and flex item behavior:

- **`float`**, **`clear`**, multicolumn layout, and **`vertical-align`** do not work with elements in flexbox mode.
- Margins do not collapse in flexbox mode. The margin edges of items are placed at the start or end of the flex line and do not overlap the padding of the container. The margins on neighboring items add up.
- The spec recommends avoiding percentage values for margin and padding on flex items because of unpredictable results.

You can see that the items have lined up in a row from left to right, which is the default Flexbox behavior if your page is in English or another language written in rows from left to right. That is because the default flexbox direction matches the direction of the language the page is written in. It would go from right to left by default in Hebrew or Arabic or in columns if the page is set with a vertical writing direction. Because it is not tied to one default direction, the terminology for specifying directions tends to be a little abstract. You'll see what I mean when we talk about "flow" in the following section.

It is worth noting that you can turn any flex item into a flex container by setting its **`display`** to **`flex`**, resulting in a nested flexbox. In fact, you'll get to try that yourself in an upcoming exercise. Some Flexbox solutions use flexboxes nested several layers deep.

Controlling the "Flow" Within the Container

Once you turn an element into a flex container, there are a few properties you can set on that container to control how items flow within it. The flow refers to the direction in which flex items are laid out as well as whether they are permitted to wrap onto additional lines.

Specifying flow direction

You may be happy with items lining up in a row as shown in FIGURE 16-2, but there are a few other options that are controllled with the **flex-direction** property.

flex-direction

Values: row | column | row-reverse | column-reverse

Default: row

Applies to: flex containers

Inherits: no

The default value is **row**, as we saw in the previous example (see the "**Row and Column Direction**" sidebar). You can also specify that items get aligned vertically in a **column**. The other options, **row-reverse** and **column-reverse**, arrange items in the direction you would expect, but they start at the end and get filled in the opposite direction. FIGURE 16-3 shows the effects of each keyword as applied to our simple example.

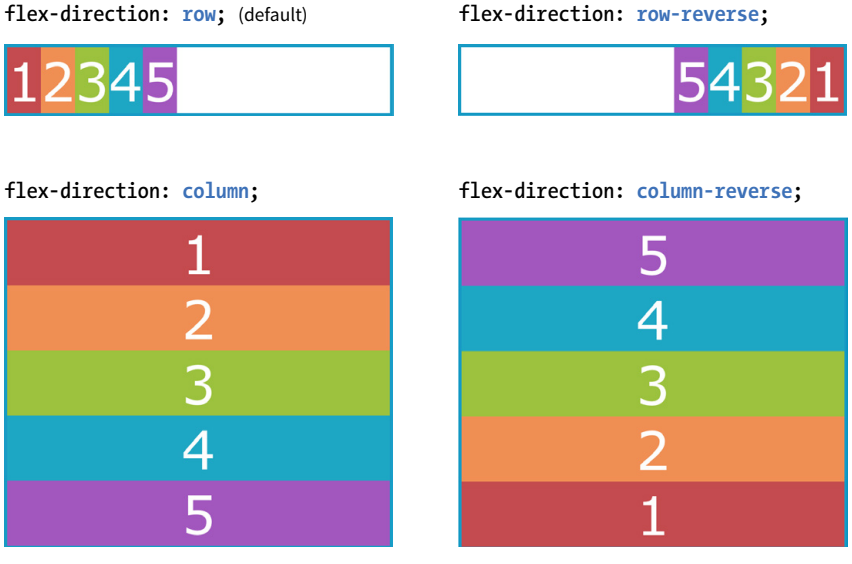


FIGURE 16-3. Examples of **flex-direction** values **row**, **row-reverse**, **column**, and **column-reverse**.

Now that you've seen Flexbox in action, it's a good time to familiarize yourself with the formal Flexbox terminology. Because the system is direction-agnostic, there are no references to "left," "right," "top," or "bottom" in the property values. Instead, we talk about the main axis and the cross axis. The main axis is the flow direction you've specified for the flex container. For primarily horizontal languages, when set to **row**, the main axis is horizontal; for **column**, the main axis is vertical (again, rows and columns are language-dependent, as explained earlier in the **"Row and Column Direction"** sidebar).

Row and Column Direction

In writing systems with horizontal lines of text, the **row** keyword lays items out horizontally, as we Westerners typically think of a "row." Bear in mind that in vertically oriented languages, **row** aligns items vertically, in keeping with the default direction of the writing system. Similarly, **column** results in horizontally aligned items in vertical languages.

This is a behavior worth knowing; however, because we are creating English language sites in this book, I'll be sticking with the assumptions that row = horizontal and column = vertical throughout this chapter for simplicity's sake.

The main axis is the flow direction you've specified for the flex container. The cross axis is perpendicular to the main axis.

The cross axis is whatever direction is perpendicular to the main axis (vertical for `row`, horizontal for `column`). The parts of a flex container are illustrated in FIGURE 16-4.

In addition to the axes, understanding the other parts of the Flexbox system makes the properties easier to learn. Both the main and cross axes have a start and an end, based on the direction in which the items flow. The main size is the width (or height if it's a column) of the container along the main axis, and the cross size is height (or width if it's a column) along the cross axis.

Wrapping onto multiple lines

If you have a large or unknown number of flex items in a container and don't want them to get all squished into the available space, you can allow them to break onto additional lines with the `flex-wrap` property.

DON'T WORRY

Keeping the main and cross axes straight as you switch between rows and columns can feel like mental gymnastics and is one of the trickier things about using Flexbox. With practice, you'll get used to it.

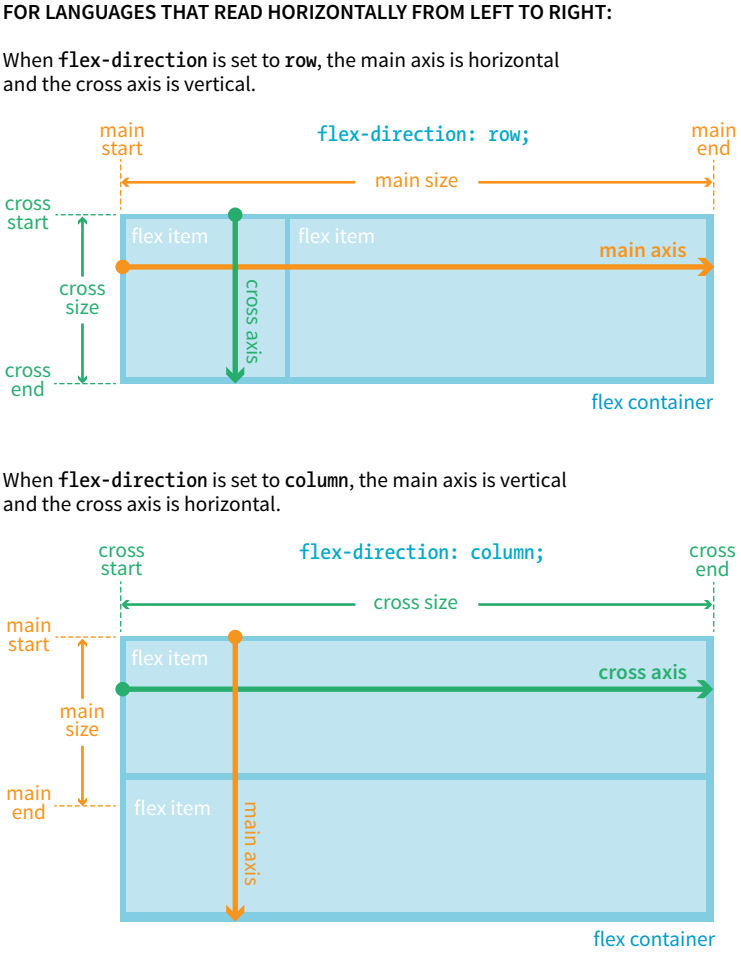


FIGURE 16-4. The parts of a flex container.

flex-wrap

Values: nowrap | wrap | wrap-reverse

Default: nowrap

Applies to: flex containers

Inherits: no

By default, items do the squish thing and do not wrap onto additional lines (**nowrap**). The **wrap** keyword turns on the ability to wrap onto multiple lines in the direction from cross start to cross end. For example, if the direction is row, then lines are positioned from the top down.

wrap-reverse breaks the elements onto multiple lines, but flows them in the opposite direction, from cross end to cross start (from the bottom up, in this case). It feels a little esoteric to me, but you never know when an occasion might arise to put it to use.

I've added more **div**s to our numbered flexbox example and I've given the flex items a width of 25% so that only four will fit across the width of the container. FIGURE 16-5 shows a comparison of the various wrap options when the **flex-direction** is the default **row**.

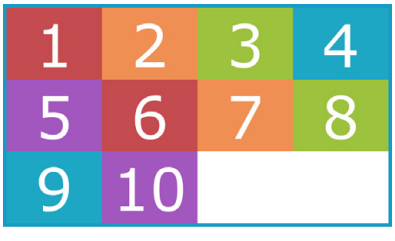
THE MARKUP

```
<div id="container">
  <div class="box box1">1</div>
  <!-- more boxes here -->
  <div class="box box10">10</div>
</div>
```

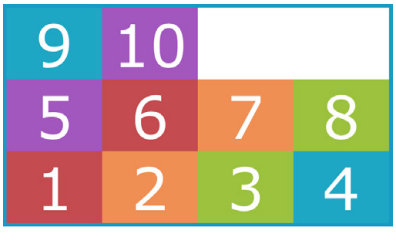
THE STYLES

```
#container {
  display: flex;
  flex-direction: row;
  flex-wrap: wrap;
}
.box {
 width: 25%;
}
```

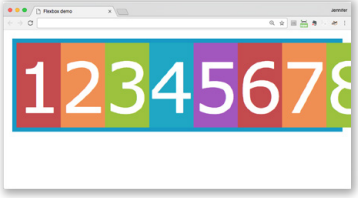
flex-wrap: wrap;



flex-wrap: wrap-reverse;



flex-wrap: nowrap; (default)



When wrapping is disabled, flex items squish if there is not enough room, and if they can't squish any further, may get cut off if there is not enough room in the viewport.

FIGURE 16-5. Comparing the effects of **nowrap**, **wrap**, and **wrap-reverse** keywords for **flex-wrap**.

By default, when the **flex-direction** is set to **column**, the container expands to contain the height of the items. In order to see wrapping kick in, you need to set a height on the container, as I've done here. FIGURE 16-6 shows how wrapping works for each of the **flex-wrap** keywords. Notice that the items are still 25% the width of their parent container, so there is space left over between the columns.

```
#container {
  display: flex;
  height: 350px;
  flex-direction: column;
  flex-wrap: wrap;
}
.box {
  width: 25%;
}
```

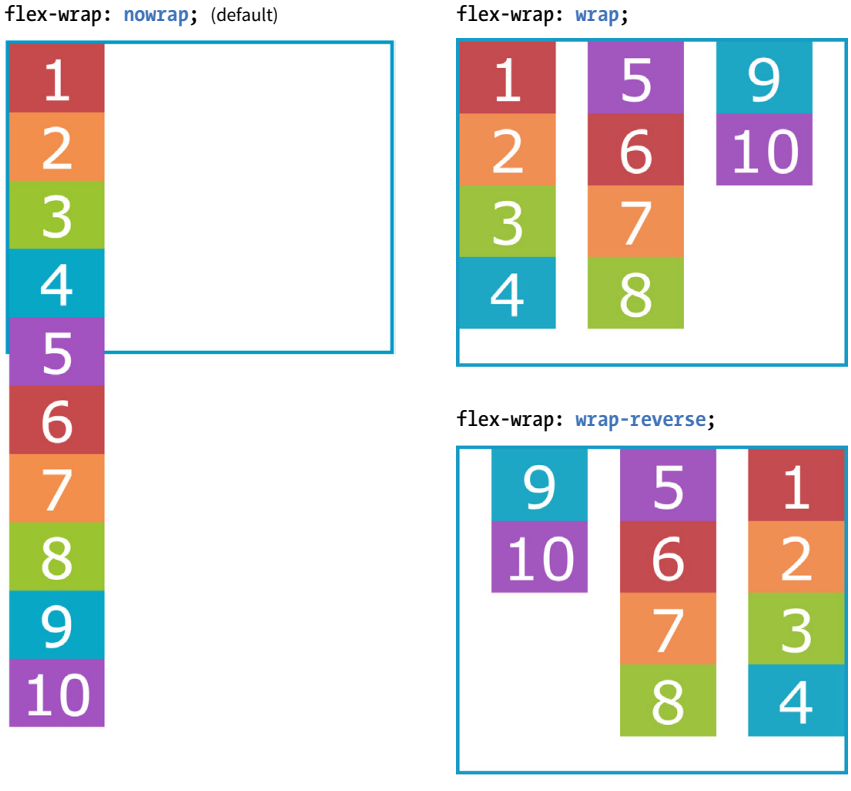


FIGURE 16-6. Comparing **nowrap**, **wrap**, and **wrap-reverse** when the items are in a column.

Putting it together with flex-flow

The shorthand property **flex-flow** makes specifying **flex-direction** and **flex-wrap** short and sweet. Omitting one value results in the default value for its respective property, which means you can use **flex-flow** for either or both direction and wrap.

flex-flow

Values: *flex-direction flex-wrap*

Default: `row nowrap`

Applies to: flex containers

Inherits: no

Using **flex-flow**, I could shorten the previous example (FIGURE 16-6) like so:

```
#container {
  display: flex;
  height: 350px;
  flex-flow: column wrap;
}
```

You've only scratched the surface of Flexbox, but you've got what it takes to whip that ugly **nav** menu on the bakery page into shape in EXERCISE 16-1.

EXERCISE 16-1. Making a navigation bar with Flexbox

Open the most recent version of the style sheet for the bakery home page in a text editor. If you need a fresh start, you will find an updated copy of *bakery-styles.css* in the materials for **Chapter 16**.

Note: Be sure to use one of the Flexbox-supporting browsers listed at the end of this section.

1. Open *bakery-styles.css* in a text editor and start by making the **ul** element in the **nav** element as neutral as possible:

   ```
   nav ul {
     margin: 0;
     padding: 0;
     list-style-type: none;
   }
   ```

   Turn that **ul** element into a flexbox by setting its **display** to **flex**. As a result, all of the **li** elements become flex items. Because we want rows and no wrapping, the default values for **flex-direction** and **flex-wrap** are fine, so the properties can be omitted:

   ```
   nav ul {
     …
     display: flex;
   }
   ```

   Save the document and look at it in a browser. You should see that the links are lined up tightly in a row, which is an improvement, but we have more work to do.

2. Now we can work on the appearance of the links. Start by making the **a** elements in the **nav** list items display as block elements instead of inline. Give them 1px rounded borders, padding within the borders (.5em top and bottom, 1em left and right), and .5em margins to give them space and to open up the brown navigation bar.

   ```
   nav ul li a {
     display: block;
     border: 1px solid;
     border-radius: .5em;
     padding: .5em 1em;
     margin: .5em;
   }
   ```

3. We want the navigation menu to be centered in the width of the **nav** section. I'm getting a little ahead here because we haven't seen alignment properties yet, but this one is fairly intuitive. Consider it a preview of what's coming up in the next section. Add the following declaration for the **nav ul** element:

   ```
   nav ul {
     …
     display: flex;
     justify-content: center;
   }
   ```

FIGURE 16-7 shows the way your navigation menu should look when you are finished.

IMPORTANT: We'll be using this version of the bakery site as the starting point for EXERCISE 16-6, so save it and keep it for later.

FIGURE 16-7. The list of links is now styled as a horizontal menu bar.

Controlling the Alignment of Flex Items in the Container

So far we've seen how to turn flexbox mode on, turning an element into a flex container and its children into flex items. We've also learned how to change the direction in which items flow, and allow them to wrap onto multiple lines. The remaining set of container properties affects the alignment of items along the main axis (**justify-content**) and cross axis (**align-items** and **align-content**).

Aligning on the main axis

By default, flex items are just as wide as they need to be to contain the element's content, which means the container may potentially have space to spare on the flex line. We saw this back in FIGURE 16-2. Also by default, the items flow in right next to each other from the "main start" (based on language direction and the direction of the flex line).

The **justify-content** property defines how extra space should be distributed around or between items that are inflexible or have reached their maximum size (see **Note**).

NOTE

You can also distribute extra space along the main axis by making the flex items themselves wider to fill the available space. That is the job of the **flex** *properties, which we'll look at in a moment.*

justify-content

Values: flex-start | flex-end | center | space-between | space-around

Default: flex-start

Applies to: flex containers

Inherits: no

Apply **justify-content** to the flex container element because it controls spacing within the container itself:

```
#container {
  display: flex;
  justify-content: flex-start;
}
```

FIGURE 16-8 shows how items align using each of the keyword values for **justify-content**. As you would expect, **flex-start** and **flex-end** position the line of items toward the start and end of the main axis, respectively, and **center** centers them.

NOTE

As new alignment keywords are added to the Grid Layout spec, they are available for Flexbox as well; however, because they are newer, they will be less well supported. Be sure to check the Flexbox spec for updates.

space-between and **space-around** warrant a little more explanation. When **justify-content** is set to **space-between**, the first item is positioned at the start point, the last item goes at the end point, and the remaining space is distributed evenly between the remaining items. The **space-around** property adds an equal amount of space on the left and right side of each item, resulting in a doubling up of space between neighboring items.

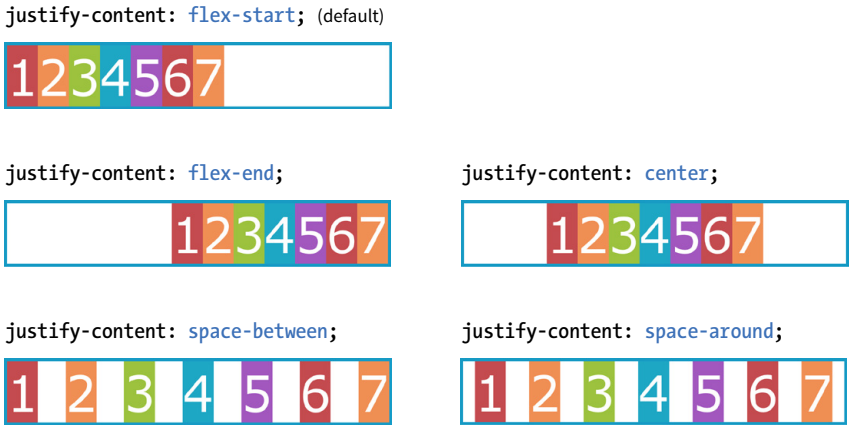


FIGURE 16-8. Options for aligning items along the main axis with **justify-content**.

NOTE

The **justify-content** *setting is applied* ***after*** *margins have been calculated on items and after the way that items have been set to "flex" has been accounted for. If the* **flex** *value for items allows them to grow to fill the container width, then* **justify-content** *is not applicable.*

When the direction is set to a column with a vertical main axis, the keywords work the same way; however, there needs to be an explicit container height with space left over in order for you to see the effect. I've changed the size of the text and set a height on the container element in FIGURE 16-9 to demonstrate the same keywords as applied to a vertical main axis.

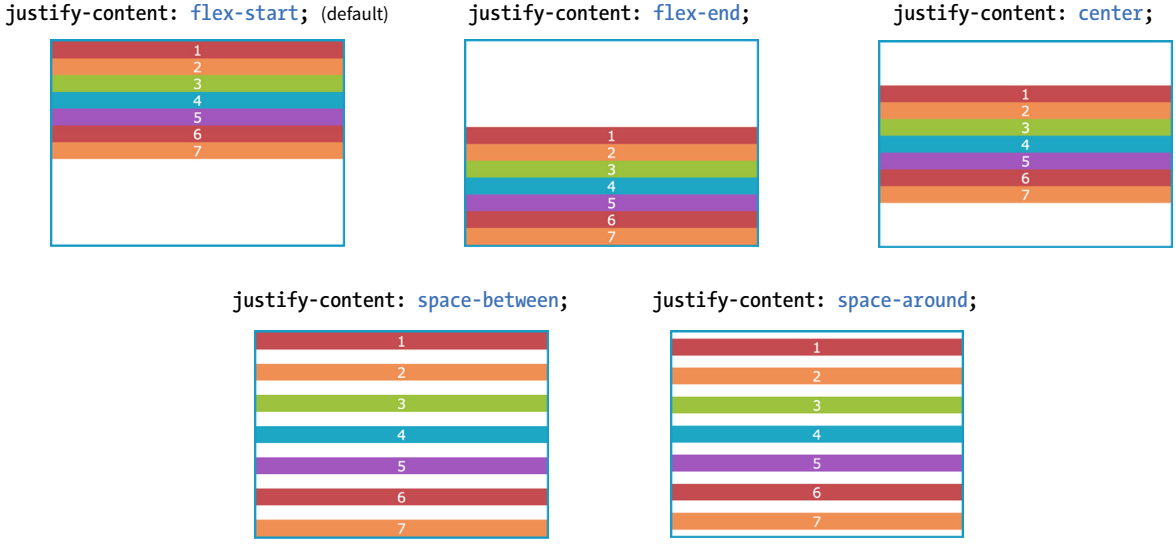


FIGURE 16-9. Options for aligning items along a vertical main axis (**flex-direction** set to **column**) with **justify-content**.

Aligning on the cross axis

That takes care of arranging things on the main axis, but you may also want to play around with alignment on the cross axis (up and down when the direction is **row**, left and right if the direction is **column**). Cross-axis alignment and stretching is the job of the **align-items** property.

align-items

Values: flex-start | flex-end | center | baseline | stretch

Default: stretch

Applies to: flex containers

Inherits: no

I've demonstrated the various keyword values for **align-items** as it applies to rows in FIGURE 16-10. In order to see the effect, you must specify the container height; otherwise, it expands just enough to contain the content with no extra space. I've given the container a height to show how items are positioned on the cross axis.

Like **justify-content**, the **align-items** property applies to the flex container (that can be a little confusing because "items" is in the name).

```
#container {
  display: flex;
  flex-direction: row;
  height: 200px;
  align-items: flex-start;
}
```

The **flex-start**, **flex-end**, and **center** values should be familiar, only this time they refer to the start, end, and center of the cross axis. The **baseline** value aligns the baselines of the first lines of text, regardless of their size. It may be a good option for lining up elements with different text sizes, such as headlines and paragraphs across multiple items. Finally, **stretch**, which is the default, causes items to stretch until they fill the cross axis.

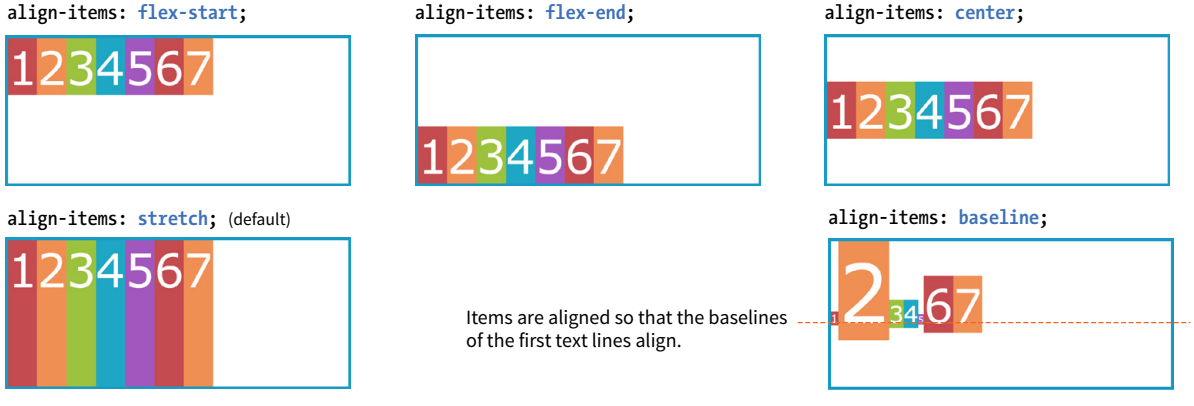


FIGURE 16-10. Aligning along the cross axis with **align-items**.

When the flex container's direction is set to **column**, **align-items** aligns items left and right. Look back at FIGURES 16-2 and 16-9 and you will see that when we set the items in a column and did not provide any alignment information, each item stretched to the full width of the cross axis because **stretch** is the default value.

If you'd like one or more items to override the cross-axis setting, use the **align-self** property on the individual item element(s). This is the first property we've seen that applies to an *item*, not the container itself. **align-self** uses the same values as **align-items**; it just works on one item at a time.

align-self

Values: flex-start | flex-end | center | baseline | stretch

Default: stretch

Applies to: flex items

Inherits: no

In the following code and FIGURE 16-11, the fourth box is set to align at the end of the cross axis, while the others have the default stretch behavior.

```
.box4 {
  align-self: flex-end;
}
```

FIGURE 16-11. Use **align-self** to make one item override the cross-axis alignment of its container.

Aligning multiple lines

The final alignment option, **align-content**, affects how multiple flex lines are spread out across the cross axis. This property applies only when **flex-wrap** is set to **wrap** or **wrap-reverse** and there are multiple lines to align. If the items are on a single line, it does nothing.

align-content applies only when there are multiple wrapped flex lines.

align-content

Values: flex-start | flex-end | center | space-around | space-between | stretch

Default: stretch

Applies to: multi-line flex containers

Inherits: no

All of the values you see in the property listing should look familiar, and they work the way you would expect. This time, however, they apply to how extra space is distributed around multiple lines on the cross axis, as shown in FIGURE 16-12.

Again, the **align-content** property applies to the flex container element. A height is required for the container as well, because without it the container would be just tall enough to accommodate the content and there would be no space left over.

```
#container {
  display: flex;
  flex-direction: row;
  flex-wrap: wrap;
  height: 350px;
  align-items: flex-start;
}

box {
  width: 25%;
}
```

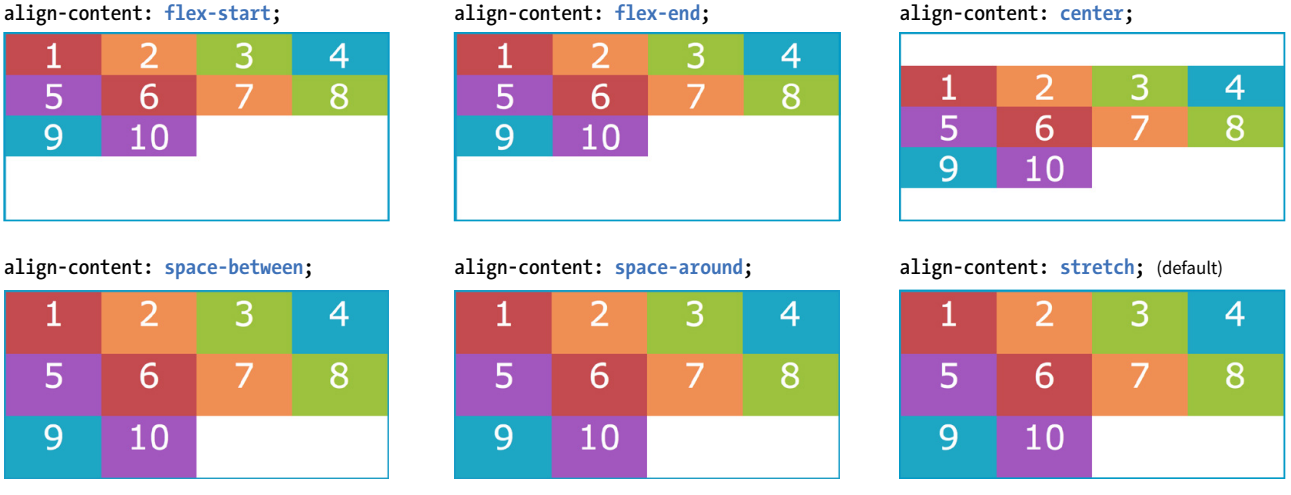


FIGURE 16-12. The **align-content** property distributes space around multiple flex lines. It has no effect when flex items are in a single line.

Aligning items with margins

As long as we're talking about alignment, there is one good trick I'd like to show you that will be useful when you start laying out components with Flexbox.

Menu bars are ubiquitous on the web, and it is common for one element of the bar, such as a logo or a search field, to be set off visually from the others. You can use a margin to put the extra container space on a specified side or

sides of a flex item, thus setting one item apart. This should be more clear with an example.

The menu in FIGURE 16-13 has a logo and four menu options. I'd like the logo to stay in the left corner but the options to stay over to the right, regardless of the width of the viewport.

THE MARKUP

```
<ul>
  <li class="logo"><img src="logo.png" alt="LoGoCo"></li>
  <li>About</li>
  <li>Blog</li>
  <li>Shop</li>
  <li>Contact</li>
</ul>
```

THE STYLES

```
ul {
  display: flex;
  align-items: center;
  background-color: #00af8f;
  list-style: none; /* removes bullets */
  padding: .5em;
  margin: 0;
}
li {
  margin: 0 1em;
}
li.logo {
  margin-right: auto;
}
```

FIGURE 16-13. Using a margin to adjust the space around flex items. In this example, the right margin of the logo item pushes the remaining items to the right.

Use margins to add space on the sides of particular flex items.

I've turned the unordered list (`ul`) into a flex container, so its list items (`li`) are now flex items. By default, the items would stay together at the start of the main axis (on the left) with extra space on the right. Setting the right margin on the logo item to `auto` moves the extra space to the right of the logo, pushing the remaining items all the way to the right (the "main end").

This technique applies to a number of scenarios. If you want just the last item to appear on the right, set its left margin to `auto`. Want equal space around the center item in a list? Set both its left and right margins to `auto`. Want to push a button to the bottom of a column? Set the top margin of the last item to `auto`. The extra space in the container goes into the margin and pushes the neighboring items away.

We've covered a lot of territory, so it's a good time to try out Flexbox in EXERCISE 16-2.

HEADS-UP

When you use `margin: auto` *on a flex item, the* `justify-content` *property no longer has a visual effect because you've manually assigned a location for the extra space on the main axis.*

EXERCISE 16-2. A flexible online menu

Now it's time for you to play around with Flexbox properties by using content a bit more complex than links in a menu bar. In this exercise, you'll format a simple online menu with a number of menu items. As always, the materials are available at *learningwebdesign.com/5e/materials*.

Open *flex-menu.html* in a text editor, and you'll see that it has all of the content ready to go as well as an internal style sheet with styles for the cosmetic aspects of the menu (colors, fonts, borders, spacing, etc.). Open the file in a browser, and the menu items should appear in a column because they are block elements. I put a border on the **#menu** wrapper **div** so you can visualize its boundaries.

1. First, we'll go for maximum impact with minimal effort by making the **#menu** wrapper **div** a flex container. There is already a rule for **#menu**, so add this declaration to it:

```
#menu {
  border: 3px solid #783F27;
  display: flex;
}
```

   Save and reload the page in the browser, and BAM!…they're in a row now! And because we haven't added any other flex properties, they are demonstrating default flexbox behavior (FIGURE 16-14):

   - Each item (defined by a **section** element) is the full height of the **#menu** container, regardless of its content.
   - The sections have their widths set to 240 pixels, and that measurement is preserved by default. Depending on how wide your browser window is set, you may see content extending beyond the container and getting clipped off, as shown in the figure.

2. By default, flex items appear in the writing direction (a row, left to right, in English). Add the **flex-direction** property to the existing **#menu** rule to try out some of the other values (**row-reverse**, **column**, **column-reverse**). The items are numbered to make their order more apparent.

```
flex-direction: row-reverse;
```

3. Set the **flex-direction** back to **row**, and let's play around with the cross-axis alignment by using the **align-items** property. Begin by setting it to **flex-start** (FIGURE 16-15). Save and reload, and see that the items all line up at the start of the cross axis (the top, in this case). Try some of the other values

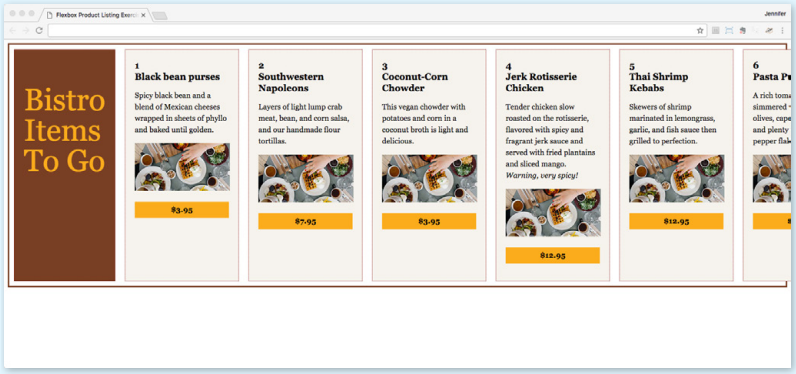


FIGURE 16-14. The bistro menu in default flexbox mode. By default, the items stay in one row even though there is not enough room for them and content gets clipped.

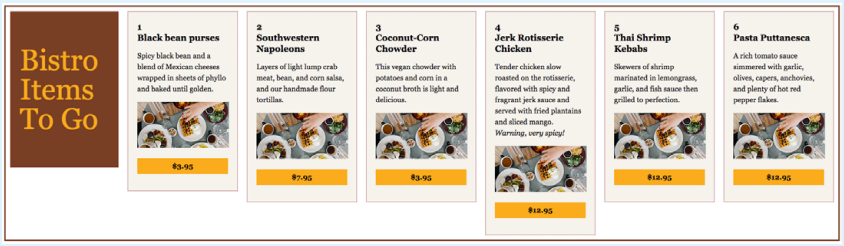


FIGURE 16-15. Using the **align-items** property to align the items at the start of the cross axis (**flex-start**).

for **align-items** (**flex-end**, **center**, **baseline**, and **stretch**) to get a feel for how each behaves.

```
align-items: flex-start;
```

4. When you are done experimenting, set **align-items** back to **stretch**. Instead of having all the items on one line and getting cropped by the edge of the browser, let's have them wrap onto multiple lines by using the **flex-wrap** property on the **#menu** container:

   ```
   flex-wrap: wrap;
   ```

   Save the file and look at it in the browser (FIGURE 16-16). Resize the browser window and watch the lines rewrap. Notice that each flex line is as tall as the tallest item in that row, but rows may have different heights based on item content.

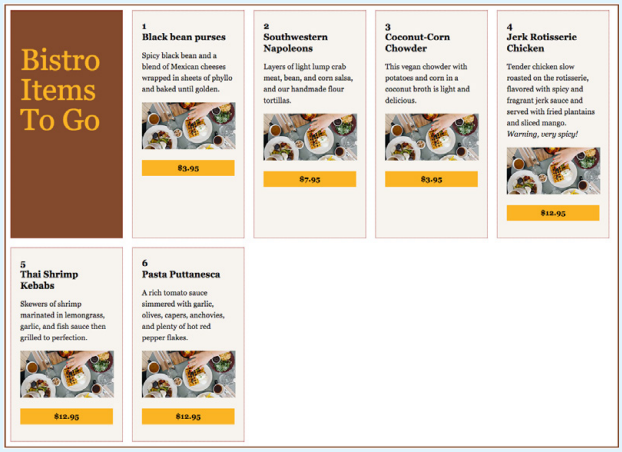


FIGURE 16-16. The menu with wrapping turned on.

5. If you'd like, you can replace the **flex-direction** and **flex-wrap** declarations with a single **flex-flow** declaration like so:

   ```
   flex-flow: row wrap;
   ```

6. By default, the items on each flex line are stacked toward the start of the main axis (the left). Try changing the main-axis alignment of items with the **justify-content** property (again, applied to the **#menu** flex container rule). I like how they look centered in the container, but check out the effect of the other values (**flex-start**, **flex-end**, **space-between**, **space-around**) as well.

   ```
   justify-content: center;
   ```

7. As a final tweak, let's make the price buttons line up at the bottom of each menu item, which is possible if each item is also a flex container. Here, I'm making each item a nested flex container by setting its **display** to **flex** and specifying the direction as **column** so they continue to stack up vertically. Now the **h2** and **p** elements become flex items within the **section** flex container.

   ```
   section {
     …
     display: flex;
     flex-direction: column;
   }
   ```

   When you reload the page in the browser, it looks about the same as when the sections were made up of block elements. The subtle difference is that now the neighboring margins between elements stack up and do not collapse.

   Now push the paragraphs containing the prices to the bottom using the **margin: auto** trick. Add this declaration to the existing style rule for elements with the class name "price."

   ```
   .price {
     …
     margin-top: auto;
   }
   ```

FIGURE 16-17 shows the final state of the "Bistro Items to Go" menu with nested flexboxes. We'll continue working on this file after we've learned the item-specific properties, so save it for later.

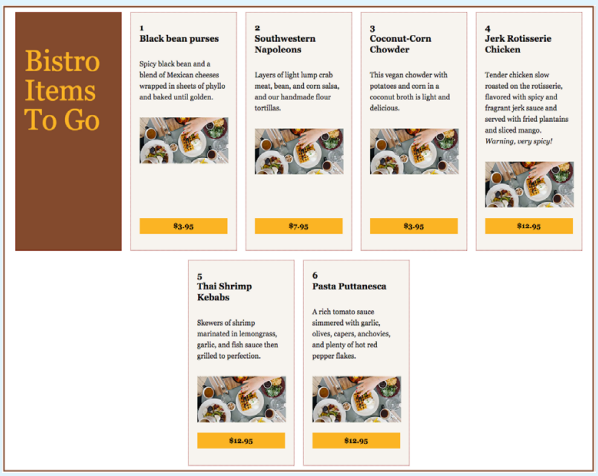


FIGURE 16-17. The menu so far with wrapping flex items and aligned prices.

Determining How Items "Flex" in the Container

One of the great marvels of the flexbox model is that items resize, or flex to use the formal term, to fit the available space. It's this flexibility that makes Flexbox such a powerful tool for designing for the wide array of screen and browser window sizes we encounter as web designers. The beauty is that the browser figures out the sizes on the fly, and that means less math for us! In this section, we'll get to know the flex properties.

Earlier, you learned about the **justify-content** property, which distributes extra space in the container between and around items along the main axis. The concept of flex is concerned with how space is distributed *within* items, growing or shrinking items as required to make them fit.

Flex is controlled with the **flex** property, which specifies how much an item can grow and shrink, and identifies its starting size. The full story is that **flex** is a shorthand property for **flex-grow**, **flex-shrink**, and **flex-basis**, but the spec strongly recommends that authors use the **flex** shorthand instead of individual properties in order to avoid conflicting default values and to ensure that authors consider all three aspects of **flex** for every instance.

The flex properties apply to flex items, not the container.

flex

Values:	none | '*flex-grow flex-shrink flex-basis*'
Default:	0 1 auto
Applies to:	flex items
Inherits:	no

The value for the **flex** property is typically three flex properties listed in this order:

```
flex: flex-grow flex-shrink flex-basis;
```

For the **flex-grow** and **flex-shrink** properties, the values 1 and 0 work like on/off switches, where 1 "turns on" or allows an item to grow or shrink, and 0 prevents it. The **flex-basis** property sets the starting dimensions, either to a specific size or a size based on the contents.

In this quick example, a list item starts at 200 pixels wide, is allowed to expand to fill extra space (1), but is not allowed to shrink (0) narrower than the original 200 pixels.

```
li {
  flex: 1 0 200px;
}
```

That should give you the general idea. In this section, we'll take a much closer look at growing, shrinking, and base size, in that order.

But first, it is important to note that **flex** and its component properties apply to flex *items*, not the container. Keeping track of which properties go on the container and which go on items is one of the tricks of using Flexbox. See the **"Flex Properties"** sidebar for a handy list of how the properties are divided.

Expanding items (flex-grow)

The first value in the **flex** property specifies whether (and in what proportion) an item may stretch larger—in other words, its **flex-grow** value (see **Note**). By default it is set to 0, which means an item is not permitted to grow wider than the size of its content or its specified width. Because items do not expand by default, the alignment properties have the opportunity to go into effect. If the extra space was taken up inside items, alignment wouldn't work.

flex-grow

Values: *number*

Default: 0

Applies to: flex items

Inherits: no

If you set the **flex-grow** value for all the items in a container to 1, the browser takes whatever extra space is available along the main axis and applies it equally to each item, allowing them all to stretch the same amount.

Let's take the simple box example from earlier in the chapter and see how it behaves with various flex settings applied. FIGURE 16-18 shows what happens when **flex-grow** is set to 1 for all box items (**flex-shrink** and **flex-basis** are left at their default values). Compare this to the same example with **flex-grow** set to the default 0 (this is the same behavior we observed in FIGURE 16-2).

THE MARKUP

```
<div id="container">
  <div class="box box1">1</div>
  <div class="box box2">2</div>
  <div class="box box3">3</div>
  <div class="box box4">4</div>
  <div class="box box5">5</div>
</div>
```

THE STYLES

```
.box {
  …
  flex: 1 1 auto;
}
```

flex: 0 1 auto; (prevents expansion)

flex: 1 1 auto; (allows expansion)

FIGURE 16-18. When **flex-grow** is set to 1, the extra space in the line is distributed into the items in equal portions, and they expand to fill the space at the same rate.

NOTE

flex-grow *is the individual property that specifies how an item may expand. Authors are encouraged to use the shorthand* **flex** *property instead.*

■ AT A GLANCE

Flex Properties

Now that you've been introduced to all the properties in the Flexible Box Module, it might be helpful to see at a glance which properties apply to containers and which are set on flex items.

Container Properties

Apply these properties to the flex container:

- display
- flex-flow
  - flex-direction
  - flex-wrap
- justify-content
- align-items
- align-content

Flex Item Properties

Apply these properties to flex items:

- align-self
- flex
  - flex-grow
  - flex-shrink
  - flex-basis
- order

If you specify a higher **flex-grow** integer to an item, it acts as a ratio that applies more space within that item. For example, giving "box4" the value **flex-grow: 3** means that it receives three times the amount of space than the remaining items set to **flex-grow: 1**. FIGURE 16-19 shows the result.

```
.box4 {
   flex: 3 1 auto;
}
```

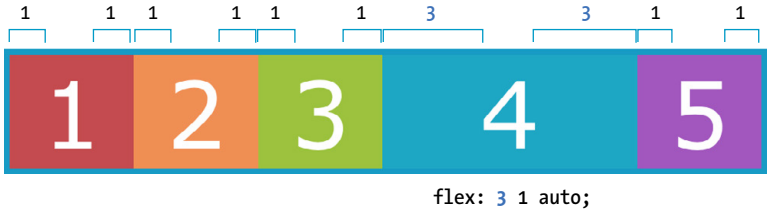


FIGURE 16-19. Assigning a different amount **flex-grow** to an individual item. Here "box4" was set to expand at three times the rate of the other items.

Notice that the resulting item is not three times as wide as the others; it just got three times the amount of space added to it.

If there's not much space left over on the line, there's a chance that each portion of space could be small enough that it would not add up to much difference. You may just need to play around with the **flex-grow** values and adjust the width of the browser until you get the effect you want.

Now that you have that concept down, shrinking should be straightforward because it is based on the same principle.

Squishing items (flex-shrink)

The second **flex** property value, **flex-shrink**, kicks in when the container is not wide enough to contain the items, resulting in a space deficit. It essentially takes away some space from within the items, shrinking them to fit, according to a specified ratio.

flex-shrink

Values: *number*

Default: 1

Applies to: flex items

Inherits: no

By default, the **flex-shrink** value is set to 1, which means if you do nothing, items shrink to fit at the same rate. When **flex-shrink** is 0, items are not permitted to shrink, and they may hang out of their container and out of view of the viewport. Finally, as in **flex-grow**, a higher integer works as a ratio. An item with a **flex-shrink** of 2 will shrink twice as fast as if it were set to 1. You

NOTE

flex-shrink *is the individual property that specifies how an item may contract. Authors are encouraged to use the shorthand* **flex** *property instead.*

will not generally need to specify a shrink ratio value. Just turning shrinking on (1) or off (0) should suffice.

Flex items stop shrinking when they reach their minimum size (defined by **min-width**/**min-height**). By default (when **min-width**/**min-height** is **auto**), this minimum is based on its **min-content** size. But it can easily be set to zero, or 12em, or any other length that seems useful. Watch for this effect when deeply nested items force a flex item to be wider than expected.

By default, items may shrink when the container is not wide enough (flex-shrink: 1).

You will see the **flex-shrink** property in action in FIGURE 16-20 in the next section.

Providing an initial size (flex-basis)

The third **flex** value defines the starting size of the item before any wrapping, growing, or shrinking occurs (**flex-basis**). It may be used instead of the **width** property (or **height** property for columns) for flex items.

flex-basis

Values:	*length* | *percentage* | content | auto
Default:	auto
Applies to:	flex items
Inherits:	no

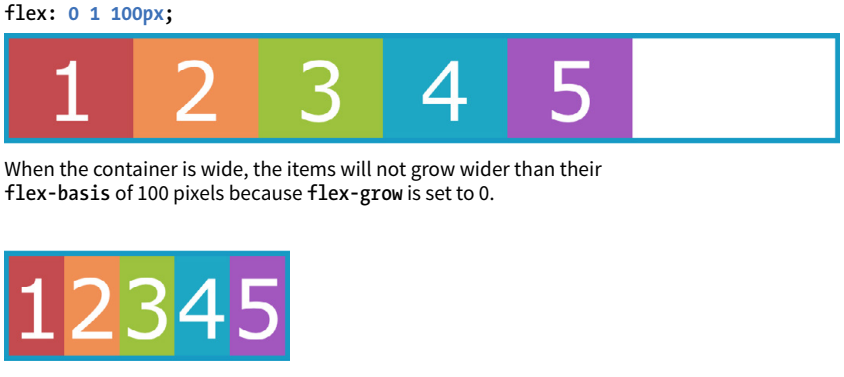
NOTE

flex-basis *is the individual property that sets the initial size of the item. Authors are encouraged to use the shorthand* **flex** *property instead.*

In this example, the **flex-basis** of the boxes is set to 100 pixels (FIGURE 16-20). The items are allowed to shrink smaller to fit in the available space (**flex-shrink: 1**), but they are not allowed to grow any wider (**flex-grow: 0**) than 100 pixels, leaving extra space in the container.

```
box {
  flex: 0 1 100px;
}
```

Flex settings override specified widths/heights for flex items.



When the container is wide, the items will not grow wider than their **flex-basis** of 100 pixels because **flex-grow** is set to 0.

When the container is narrow, the items are allowed to shrink to fit (**flex-shrink: 1**).

FIGURE 16-20. Using **flex-basis** to set the starting width for items.

By default, **flex-basis** is set to **auto**, which uses the specified **width**/**height** property values for the item size. If the item's main size property (**width** or **height**) is not set or is **auto** (its default), **flex-basis** uses the content width. You can also explicitly set **flex-basis** to be the width of the content with the **content** keyword; however, that value is poorly supported as of this writing.

In this example, the flex basis for the boxes is set to 100 pixels because the **auto** value uses the value set by **width**. Items are allowed to grow, taking up any extra space in the container, but they are not allowed to shrink.

```
box {
  width: 100px;
  flex: 1 0 auto;
}
```

When the browser goes about sizing a flex item, it consults the **flex-basis** value, compares it to the available space along the axis, and then adds or removes space from items according to their grow and shrink settings. It's important to note that if you allow an item to grow or shrink, it could end up being narrower or wider than the length provided by **flex-basis** or **width**.

Handy shortcut flex values

The advantage to using the **flex** property is that there are some handy shortcut values that cover typical Flexbox scenarios. Curiously, some of the shortcut values override the defaults of the individual properties, which was confusing to me at first, but in the end it results in more predictable behaviors. When creating a flexbox component, see if one of these easy settings will do the trick:

When creating a flexbox component, see if you can take advantage of one of the handy flex shortcuts.

flex: initial;

This is the same as **flex: 0 1 auto**. It prevents the flex item from growing even when there is extra space, but allows it to shrink to fit in the container. The size is based on the specified **width**/**height** properties, defaulting to the size of the content. With the **initial** value, you can use **justify-content** for horizontal alignment.

flex: auto;

This is the same as **flex: 1 1 auto**. It allows items to be fully flexible, growing or shrinking as needed. The size is based on the specified **width**/**height** properties.

flex: none;

This is equivalent to **flex: 0 0 auto**. It creates a completely inflexible flex item while sizing it to the **width** and **height** properties. You can also use **justify-content** for alignment when flex is set to **none**.

`flex: integer;`

This is the same as **`flex: integer 1 0px`**. The result is a flexible item with a flex basis of 0, which means it has absolute flex (see the sidebar "**Absolute Versus Relative Flex**") and free space is allocated according to the flex number applied to items.

How are you doing? Are you hanging in there with all this Flexbox stuff? I know it's a lot to take in at once. We have just one more Flexbox item property to cover before you get another chance to try it out yourself.

Absolute Versus Relative Flex

In FIGURE 16-19, we saw how extra space is assigned to items based on their flex ratios. This is called relative flex, and it is how extra space is handled whenever the **`flex-basis`** is set to any size other than zero (0), such as a particular **`width`**/**`height`** value or **`auto`**.

However, if you reduce the value of **`flex-basis`** to 0, something interesting happens. With a basis of 0, the items get sized proportionally according to the flex ratios, which is known as absolute flex. So with **`flex-basis: 0`**, an item with a **`flex-grow`** value of 2 *would* be twice as wide as the items set to 1. Again, this kicks in only when the **`flex-basis`** is 0.

In practice it is recommended that you always include a unit after the 0, such as **`0px`** or the preferred **`0%`**.

In this example of absolute flex, the first box is given a **`flex-grow`** value of 2, and the fourth box has a **`flex-grow`** value of 3 via the aforementioned shortcut **`flex: integer`**. In FIGURE 16-21, you can see that the resulting overall size of the boxes is in proportion to the **`flex-grow`** values because the **`flex-basis`** is set to 0.

```
.box {
  /* applied to all boxes */
  flex: 1 0 0%;
}
.box1 {
  flex: 2; /* shortcut value for flex: 2 1 0px */
}
.box4 {
  flex: 3; /* shortcut value for flex: 3 1 0px */
}
```

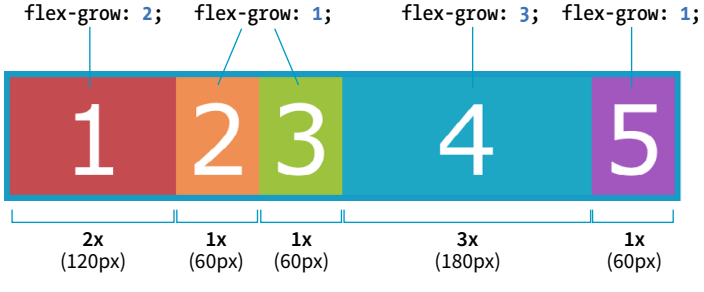


FIGURE 16-21. In absolute flex, boxes are sized according to the flex value ratios.

NOTE

I use Flexbox to format a responsive form in the "Styling Forms" section of **Chapter 19, More CSS Techniques**. *Flex properties allow form fields to adapt to the available width, while labels are set to always stay the same size. Wrapping allows form fields to move below their labels on smaller screens. You've probably got Flexbox in your head right now, so it might be worth taking a look ahead.*

Changing the Order of Flex Items

One of the killer features of Flexbox is the ability to display items in an order that differs from their order in the source (see the **"When to Reorder (and When Not To)"** sidebar). That means you can change the layout order of elements by using CSS alone. This is a powerful tool for responsive design, allowing content from later in a document to be moved up on smaller screens.

To change the order of items, apply the **order** property to the particular item(s) you wish to move.

order

Values: *integer*

Default: 0

Applies to: flex items and absolutely positioned children of flex containers

Inherits: no

The value of the **order** property is a positive or negative number that affects the item's placement along the flex line. It is similar to the **z-index** property in that the specific number value doesn't matter, only how it relates to other values.

By default, all items have an **order** value of zero (0). When items have the same **order** value, they are laid out in the order in which they appear in the HTML source. If they have *different* **order** values, they are arranged from the lowest **order** value to the highest.

Going back to our colorful numbered box example, I've given **box3** an **order** value of 1. With a higher order value, it appears after all the items set to 0 (the default), as shown in FIGURE 16-22.

```
.box3 {
  order: 1;
}
```

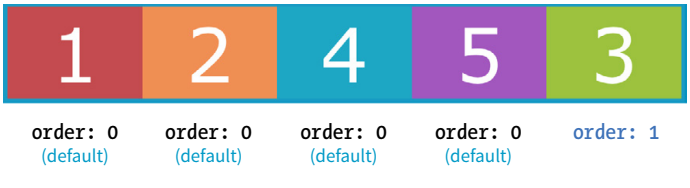


FIGURE 16-22. Changing the order of items with the **order** property. Setting **box3** to **order: 1** makes it display after the rest.

When multiple items share the same **order** value, that group of value-sharing items (called an ordinal group) sticks together and displays in source order. What happens if I give **box2** an **order** value of 1 as well? Now both **box2** and **box3** have an **order** value of 1 (making them an ordinal group), and they get

When to Reorder (and When Not to)

Keep in mind that although convenient, reordering is only a visual display sleight-of-hand and should be used with discretion. Some points to remember:

- Although elements display in a different order for visual browsers, alternative devices such as screen readers still read the content in the order in which it appears in the source (although it is not 100% reliable in the real world).
- Reorder the source if there is a logical (rather than visual) reason for the reordering.
- Don't use **order** because it is more convenient.
- Use **order** if the logical and visual order are *intended* to be disjointed.

displayed in source order after all the items with the lower **order** value of 0 (FIGURE 16-23).

```
.box2, .box3 {
  order: 1
}
```

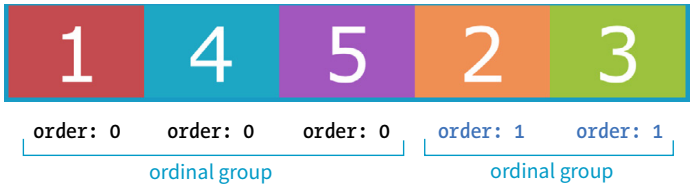


FIGURE 16-23. Setting **box2** to **order: 1** as well makes it display after the items with the default order of 0.

You can also use negative values for **order**. To continue with our example, I've given **box5** an order value of –1. Notice in FIGURE 16-24 that it doesn't just move back one space; it moves before all of the items that still have **order** set to 0, which is a higher value than –1.

```
.box5 {
  order: -1
}
```

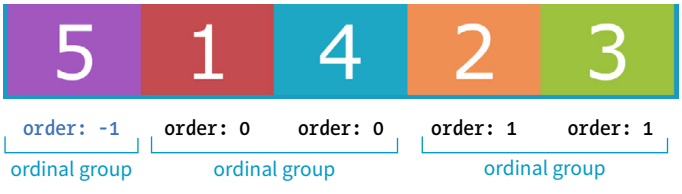


FIGURE 16-24. Negative values display before items with the default order of 0.

I've used simple values of 1 and –1 in my examples, but I could have used 10008 or –649, and the result would be the same; the order goes from least value to greatest value. Number values don't need to be in sequential order.

Now let's take a look at how we can use **order** for something more useful than moving little boxes around in a line. Here is a simple document with a header, a main section consisting of an article and two aside elements, and a footer:

```
<header>…</header>
<main>
  <article><h2>Where It's At</h2></article>
  <aside id="news"><h2>News></h2></aside>
  <aside id="contact"><h2>Contact</h2><aside>
</main>
<footer>…<footer>
```

In the following CSS, I've made the `main` element a flexbox container so the `article` and `aside` elements line up in a row, creating three columns (FIGURE 16-25). I set the `flex` factor for each item, allowing them to grow and shrink, and set their widths with `flex-basis`. Finally, I used the `order` property to specify the order in which I'd like them to appear. Notice that the Contact section is now first in the row, although it appears last in the source order. And, as an added bonus, all of the columns fill the height of the main container despite the amount of content in them.

```
main {
  display: flex;
}
article {
  flex: 1 1 50%;
  order: 2;
}
#news {
  flex: 1 1 25%;
  order: 3;
}
#contact {
  flex: 1 1 25%;
  order: 1;
}
```

NOTE

Although you can create a full-page layout with Flexbox, the task is more appropriately handled with Grid Layout, which we'll cover next. However, because Flexbox has better browser support than Grid Layout, it may be a suitable fallback. Flexbox is better suited for individual components on the page such as navigation, series of product "cards," or anything that you want to put in a line.

FIGURE 16-25. A columned layout using Flexbox.

That concludes our tour of Flexbox properties! In EXERCISE 16-3, you can put some of the item-level properties to use in the bistro menu. When you are finished, come back for some tips on dealing with varying browser support in the next section.

Browser Support for Flexbox

The current Flexible Box Layout Module became a stable Candidate Recommendation in 2012 (*www.w3.org/TR/css-flexbox-1/*). The good news is that all major desktop and mobile browsers have supported the standard

EXERCISE 16-3. Adjusting flex and order

The online menu is looking pretty good, but let's put a few finishing touches on it. Open the *flex-menu.html* file as you left it at the end of EXERCISE 16-2.

1. Instead of having lots of empty space inside the menu container, we'll make the items fill the available space. Because we want the items to be fully flexible, we can use the **auto** value for **flex** (the same as **flex: 1 1 auto;**). Add this declaration to the **section** rule to turn on the stretching behavior:

```
section {
  …
  flex: auto;
}
```

2. OK, one last tweak: let's make the photos appear at the top of each menu item. Because each section is a flex container, we can use the **order** property to move its items around. In this case, select the paragraphs with the "photo" class name and give it a value less than the default 0. This will make the photo display first in the line (FIGURE 16-26):

```
.photo {
  order: -1;
}
```

If you want to get fancy, you can set the width of the **img** elements to 100% so they always fill the width of the container. The little image I've provided gets quite blurry when it expands larger, so you can see how the responsive image techniques we covered in **Chapter 7, Adding Images**, might be useful here. It's not the best-looking web page in the world, but you got a chance to try out a lot of the Flexbox properties along the way.

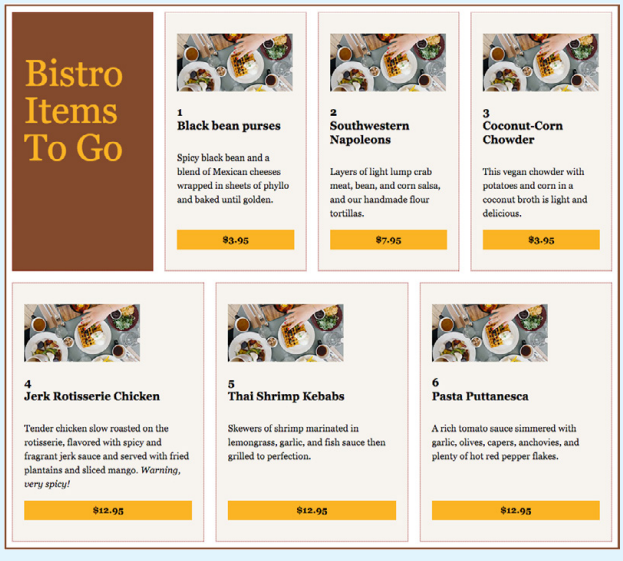


FIGURE 16-26. The final bistro menu with items flexing to fill the extra space and the photos moved to the top of each listing.

since 2015 and a few since as far back as 2013. That covers roughly 80–90% of users as of this writing according to *CanIUse.com*.

The Flexbox specification went through a lot of big changes in its path to stabilization, and along the way, some older browsers implemented those old specs. The three main releases are as follows:

Current version (2012)

Syntax example: `display: flex;`

Supported by: IE11+, Edge 12+, Chrome 21-28 (`-webkit-`), Chrome 29+, Firefox 22–27 (`-moz-`, no wrapping), Firefox 28+, Safari 6–8 (`-webkit-`), Safari 9+, Opera 17+, Android 4.4+, iOS 7–8.4 (`-webkit-`), iOS 9.2+

"Tweener" version (2011)

Syntax example: `display: flexbox;`

Supported by: IE10

Old version (2009)

Syntax example: `display: box;`

Supported by: Chrome <21, Safari 3.1–6, Firefox 2–21, iOS 3.2–6.1, Android 2.1–4.3

What you won't find in these listings is Internet Explorer 9 and earlier, which lack Flexbox support altogether.

Ensuring Flexbox works on the maximum number of browsers requires a gnarly stack of prefixes and alternative properties, the details of which are too complicated to dive into here. It's also not something you'd want to write out by hand anyway, but fortunately there are options.

WARNING

Be aware that although Autoprefixer makes adding prefixes easier, it does not guarantee that your flexboxes will work seamlessly in all browsers. There are behavior differences that can be unpredictable, so be sure to test on all of your target browsers.

You can use Autoprefixer to magically generate that gnarly stack for you automatically. As you're learning and practicing your CSS skills, you can convert your styles online at *autoprefixer.github.io*. Just paste in your styles, and it spits out the code (FIGURE 16-27) that you can add to your style sheet.

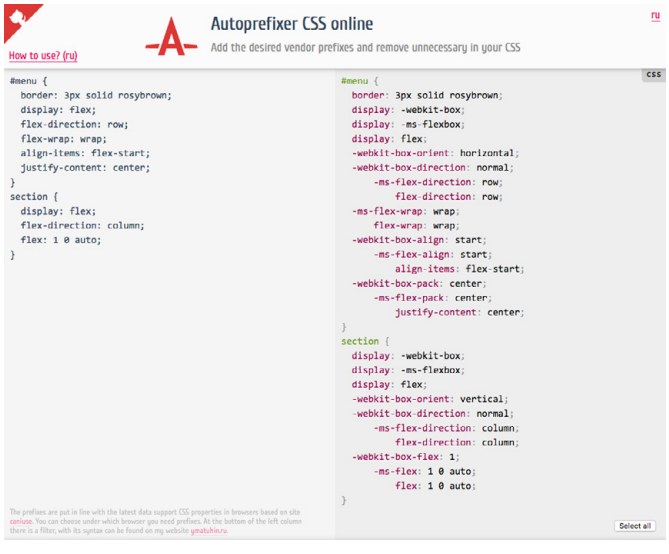


FIGURE 16-27. The Autoprefixer site converts standard Flexbox styles into all the styles needed for full browser support.

When you are ready to bring your workflow to a professional level, you can include Autoprefixer as part of a "build step" that automates a lot of the development gruntwork. If you are using a CSS preprocessor such as SASS, you can also use "mixins" to manage tedious prefixes. We'll look at build tools and preprocessors in **Chapter 20, Modern Web Development Tools**.

You may still want to provide fallback styles for non-supporting browsers (floats, inline blocks, and table display values are all options). If that is the case, you can use a feature detection technique to determine whether the browser supports Flexbox. If the browser fails the test, it gets a fallback set of

■ ONLINE RESOURCE

Flexbugs

There are some buggy implementations of Flexbox out there. Lucky for us, Philip Walton has been gathering all of these bugs in a GitHub repository called Flexbugs. To see the bugs and workarounds for them, visit *github.com/philipwalton/flexbugs*.

styles, while supporting browsers get the full Flexbox treatment. We'll take a look at feature detection in **Chapter 19**.

One big layout technique down, one big layout technique to go! Are you still with me? We've covered a lot of nitty-gritty details, and if you're like me, your head may be swimming. That's why I've included FIGURE 16-28. It has nothing to do with CSS layout, but I figured we could use a breather. In fact, why don't you put down this book and take a little walk before taking on grids?

FIGURE 16-28. This adorable red panda has nothing to do with CSS layout, but I figured we could use a breather before moving on to Grid Layout (photo by Teri Finn).

CSS GRID LAYOUT

At long last, we web designers and developers have a CSS module for using an underlying grid to achieve true page layout—and we only had to wait 25 years to get it! The CSS Grid Layout Module provides a system for laying out elements in rows *and* columns (remember that Flexbox lays out elements on one axis only) in a way that can remain completely flexible to fit a variety of screen sizes or mimic a print page layout. You can use grids to create the sort of web page layouts that are familiar today, or get more sophisticated

Re-creation of print jazz poster using grid

Re-creation of Die Neue Typography lecture invitation (1927) using grid

Overlap experiment with photos by Dorthea Lange

FIGURE 16-29. Examples of grid-based designs from Jen Simmons's "Experimental Layout Lab" page (*labs.jensimmons.com*).

with typography and whitespace as Jen Simmons has done in her Lab demos (FIGURE 16-29). You can also use a grid to format just a portion of a page, such as a gallery of images or products.

In this section, I will give you a good head start on using Grid Layout; however, I should note that there will be a few stones left unturned that you can explore on your own.

The Grid Layout Module is one of the more complex specs in CSS, the finer points of which could fill a book. In fact, Eric Meyer has written that book: *Grid Layout in CSS* (O'Reilly)(see **Note**). I found that Eric helped me make practical sense of the dense language of the spec itself (which you will also want to reference at *www.w3.org/TR/css-grid-1/*). I also highly recommend Grid expert Rachel Andrew's book, *The New CSS Layout* (A Book Apart) for a complete view of how we got to grid layouts and how to use them.

You will also find many great Grid resources online, which I will round up at the end of this section.

NOTE

CSS: The Definitive Guide, 4th edition *(O'Reilly), by Eric A. Meyer and Estelle Weyl, is a megavolume of everything you could ever want to know about CSS. It contains the entire* ***Grid Layout in CSS*** *book as a chapter.*

BROWSER SUPPORT NOTE

Internet Explorer versions 10 and 11 and MS Edge through 15 implemented an early draft of the Grid Layout Module, much of which has since been made obsolete. They should be treated as non-supporting browsers when it comes to the standard grid styles outlined in this chapter. However, if those Microsoft browsers are used by a significant share of your target audience, it is probably worth targeting them with an alternative version of your layout written in the older grid syntax they understand.

The Obligatory Talk About Browser Support

There's great and not-so-great news about browser support for Grid Layout. The great news is that Chrome 57+, Opera, Firefox 52+, Safari 10+, and iOS Safari 10+ all started supporting the Grid standard free and clear of browser prefixes in March 2017. Microsoft Edge added support in version 16 in 2017.

The not-so-great news is that in addition to lingering older versions of those browsers, no version of Internet Explorer supports the current Grid standard (see the **Browser Support Note**).

So, for the time being, you need to provide an alternative layout for non-supporting browsers by using Flexbox or old-fashioned floats (or the older Grid specification for IE and Edge <15), depending on the browsers you need to target. A good way to get your Grid-based layouts to the browsers that can handle them is to use a CSS Feature Query that checks for Grid support and provides the appropriate set of styles. Feature queries are discussed in detail in **Chapter 19**.

Be sure to check *CanIUse.com* for updated browser support information. Another good resource is the Browser Support page at the "Grid by Example" site, created by Rachel Andrew (*gridbyexample.com/browsers*), where she posts browser support news as well as known bugs.

How Grid Layout Works

The process for using the CSS Grid Layout Module is fundamentally simple:

1. **Use the `display` property to turn an element into a grid container.** The element's children automatically become grid items.
2. **Set up the columns and rows for the grid.** You can set them up explicitly and/or provide directions for how rows and columns should get created on the fly.
3. **Assign each grid item to an area on the grid.** If you don't assign them explicitly, they flow into the cells sequentially.

What makes Grid Layout complicated is that the spec provides *so* many options for specifying every little thing. All those options are terrific for customizing production work, but they can feel cumbersome when you are learning Grids for the first time. In this chapter, I'll set you up with a solid Grid toolbox to get started, which you can expand on your own as needed.

Grid Terminology

Before we dive into specific properties, you'll need to be familiar with the basic parts and vocabulary of the Grid system.

Starting with the markup, the element that has the **`display: grid`** property applied to it becomes the grid container and defines the context for grid formatting. All of its direct child elements automatically become grid items that end up positioned in the grid. If you've just read the Flexbox section of this chapter, this children-become-items scheme should sound familiar.

The key words in that previous paragraph are "direct child," as only those elements become grid items. Elements contained in those elements do not, so you cannot place them on the grid. You can, however, nest a grid inside another grid if you need to apply a grid to a deeper level.

The grid itself has a number of components, as pointed out in FIGURE 16-30.

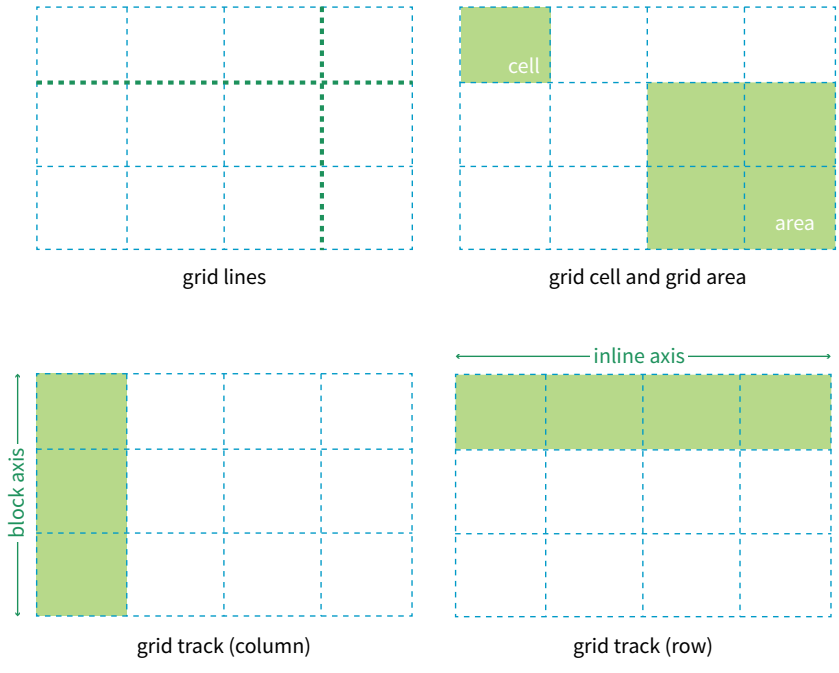


FIGURE 16-30. The parts of a CSS grid.

Grid line

The horizontal and vertical dividing lines of the grid are called grid lines.

Grid cell

The smallest unit of a grid is a grid cell, which is bordered by four adjacent grid lines with no grid lines running through it.

Grid area

A grid area is a rectangular area made up of one or more adjacent grid cells.

Grid track

The space between two adjacent grid lines is a grid track, which is a generic name for a grid column or a grid row. Grid columns are said to go along the block axis, which is vertical (as block elements are stacked) for languages written horizontally. Grid rows follow the inline (horizontal) axis.

It is worth pointing out that the structure established for the grid is independent from the number of grid items in the container. You could place 4 grid items in a grid with 12 cells, leaving 8 of the cells as "whitespace." That's the beauty of grids. You can also set up a grid with fewer cells than grid items,

and the browser adds cells to the grid to accommodate them. It's a wonderfully flexible system.

Without further ado, it's time to get into some code.

Declaring Grid Display

To turn an element into a grid container, set its **display** property to **grid** or **inline-grid** (see **Note**).

In this simple example, the **#layout div** becomes a grid container, and each of its children (**#one**, **#two**, **#three**, **#four**, and **#five**), therefore, is a grid item.

THE MARKUP

```
<div id="layout">
  <div id="one">One</div>
  <div id="two">Two</div>
  <div id="three">Three</div>
  <div id="four">Four</div>
  <div id="five">Five</div>
</div>
```

THE STYLES

```
#layout {
  display: grid;
}
```

That sets the stage (or to use the more accurate term, the context) for the grid. Now we can specify how many rows and columns we want and how wide they should be.

Setting Up the Grid

Because I don't want to have to figure out cells and spans in my head, I've made a quick sketch of how I'd like my final grid to look (FIGURE 16-31). A sketch is a good first step for working with grids. From the sketch, I can see that my layout requires three row tracks and three column tracks even though some of the content areas span over more than one cell. This is a pretty standard arrangement for a web page, and although I'm sticking with one-word content so we can focus on structure, you can imagine longer text content filling each area.

NOTE

You probably noticed that this page layout with its header, footer, and three columns looks like the one we made using Flexbox in FIGURE 16-25. And you're right! It just goes to show that there may be several solutions for getting to an intended result. Once Grid Layout becomes solidly supported, it will be the clear winner for creating flexible, whole-page layouts like this one.

NOTE

Inline grids function the same as block-level grids, but they can be used in the flow of content. In this section, I focus only on block-level grids.

As of this writing, work has begun on a Working Draft of CSS Grid Layout Module Level 2, which incudes a "subgrid" mode that allows a nested grid to inherit its grid structure from its parent.

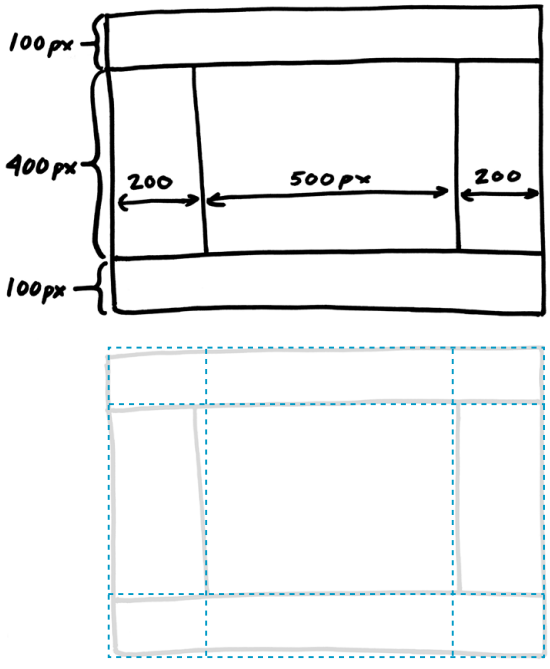


FIGURE 16-31. A rough sketch for my grid-based page layout. The dotted lines in the bottom image show how many rows and columns the grid requires to create the layout structure.

Defining grid tracks

To set up a grid in CSS, specify the height of each row and the width of each column (see **Note**) with the template properties, **`grid-template-rows`** and **`grid-template-columns`**, which get applied to the container element.

`grid-template-rows`
`grid-template-columns`

Values: none | *list of track sizes and optional line names*

Default: none

Applies to: grid containers

Inherits: no

The value of the **`grid-template-rows`** property is a list of the *heights* for each row track in the grid. The value of the **`grid-template-columns`** is a list of the *widths* for each column track. The number of track sizes determines the number of rows or columns. For example, if you provide four lengths for **`grid-template-columns`**, you get a grid that is initially divided into four columns.

You can also include names for the grid lines between tracks, which we'll get to in a moment, but for now, let's start off as simply as possible.

NOTE

Like the Flexbox Module, the Grid Layout Module is dependent on the direction of the language in which the page is written. In this book, I will base grid terminology on the left-to-right, top-to-bottom writing direction.

Grid track sizes

In the following example, I've added template properties to divide the **#layout** container into three columns and three rows with the sizes I designated in my original sketch (FIGURE 16-31):

```
#layout {
  display: grid;
  grid-template-rows: 100px 400px 100px;
  grid-template-columns: 200px 500px 200px;
}
```

Let's see what happens if I do a quick check of the grid so far in the browser. FIGURE 16-32 shows that by default, the grid items flow in order into the available grid cells. I've added background colors to the items so their boundaries are clear, and I used Firefox CSS Grid Inspector (right) to reveal the entire grid structure.

Because there are only five child elements in the **#layout div**, only the first five cells are filled. This automatic flowing behavior isn't what I'm after for this grid, but it is useful for instances in which it is OK for content to pour into a grid sequentially, such as a gallery of images. Soon, we will place each of our items on this grid deliberately, but first, let's look at the template property values in greater depth.

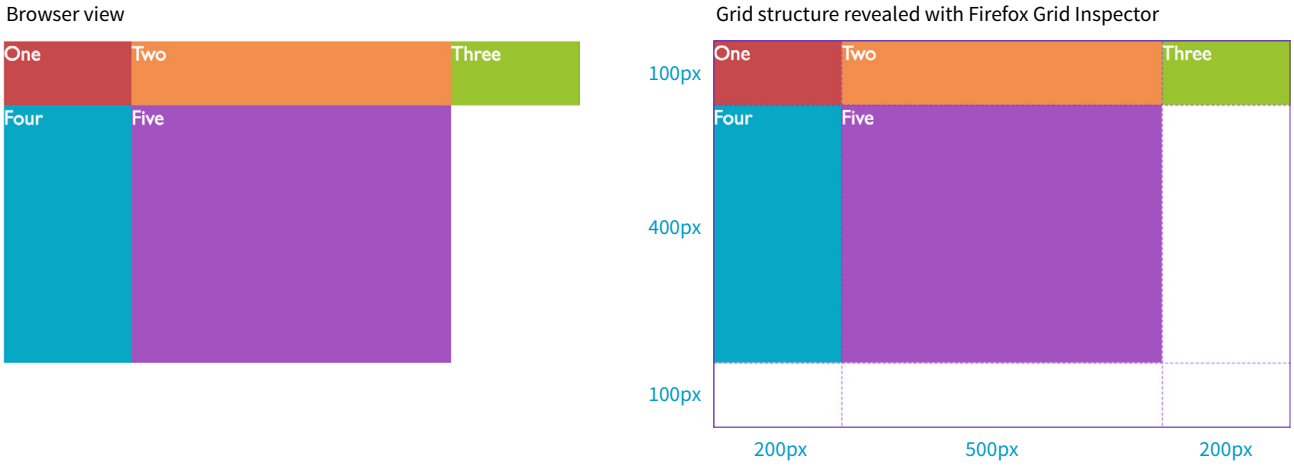


FIGURE 16-32. By default, grid items flow into the grid cells by rows.

Grid line numbers and names

When the browser creates a grid, it also automatically assigns each grid line a number that you can reference when positioning items. The grid line at the start of the grid track is 1, and lines are numbered sequentially from there. FIGURE 16-33 shows how the grid lines are numbered for our sample grid.

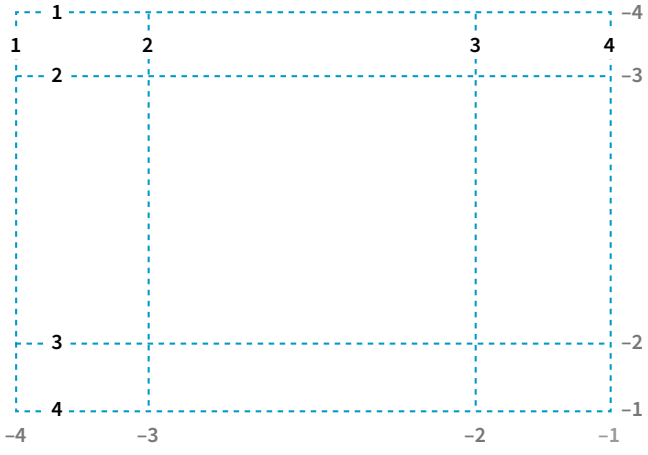


FIGURE 16-33. Grid lines are assigned numbers automatically.

The lines are numbered from the end of tracks as well, starting with –1, and numbers count back from there (–2, –3, etc.), as shown by the gray numbers in FIGURE 16-33. Being able to target the end of a row or column without counting lines (or even knowing how many rows or columns there are) is a handy feature. You'll come to love that –1.

But if you don't like to keep track of numbers, you can also assign names to lines that may be more intuitive. In the following example, I've assigned names that correspond to how I will be using the grid in the final page. Line names are added within square brackets in the position they appear relative to the tracks.

```
#layout {
  display: grid;
  grid-template-rows: [header-start] 100px [content-start] 400px
[footer-start] 100px;
  grid-template-columns: [ads] 200px [main] 500px [links] 200px;
}
```

Based on this example, the grid line at the top of the grid can now be referred to as "header-start," "1," or "–4." I could also name the line that comes after the first row track "header-end" even though I've already named it "content-start." To give a line more than one name, just include all the names in the brackets, separated by spaces:

```
grid-template-rows: [header-start] 100px [header-end content-start]
400px [footer-start] 100px;
```

It is common for each grid line to end up with multiple names and numbers, and you can choose whichever is the easiest to use. We'll be using these numbers and names to place items on the grid in a moment.

■ COOL TOOL TIP

Firefox CSS Grid Inspector and Layout Panel

Firefox 52+ includes a great developer tool called the CSS Grid Inspector that overlays a representation of the grid structure for elements with their **`display`** set to **`grid`**. It's what I used for the right screenshot in FIGURE 16-32. To get to it, open the Inspector (**Tools → Web Developer → Inspector**). Find an element that is a grid and click the # icon, and you'll see the grid overlaid on the page.

You can also click the Layout tab to access the Layout Panel, which lists all the grid containers on the page and provides tools for analyzing grid lines and areas. It also has a box-model properties component so you can easily see the dimensions, padding, border, and margins for every grid-related element, and more. These visual tools make it easier to tweak your designs.

As this book goes to press, the news is that similar grid layout development tools are coming to Chrome and Safari. The future looks bright for grid designers!

Specifying track size values

I provided all of the track sizes in my example in specific pixel lengths to make them easy to visualize, but fixed sizes are one of many options. They also don't offer the kind of flexibility required in our multi-device world. The Grid Layout Module provides a *whole bunch* of ways to specify track sizes, including old standbys like lengths (e.g., pixels or ems) and percentage values, but also some newer and Grid-specific values. I'm going to give you quick introductions to some useful Grid-specific values: the **`fr`** unit, the **`minmax()`** function, **`auto`**, and the content-based values **`min-content`**/**`max-content`**. We'll also look at functions that allow you to set up a repeating pattern of track widths: the **`repeat()`** function with optional **`auto-fill`** and **`auto-fit`** values.

Fractional units (flex factor)

The Grid-specific fractional unit (**`fr`**) allows developers to create track widths that expand and contract depending on available space. To go back to the example, if I change the middle column from **`500px`** to **`1fr`**, the browser assigns all leftover space (after the 200-pixel column tracks are accommodated) to that column track (FIGURE 16-34).

```
#layout {
  display: grid;
  grid-template-rows: 100px 400px 100px;
  grid-template-columns: 200px 1fr 200px;
}
```

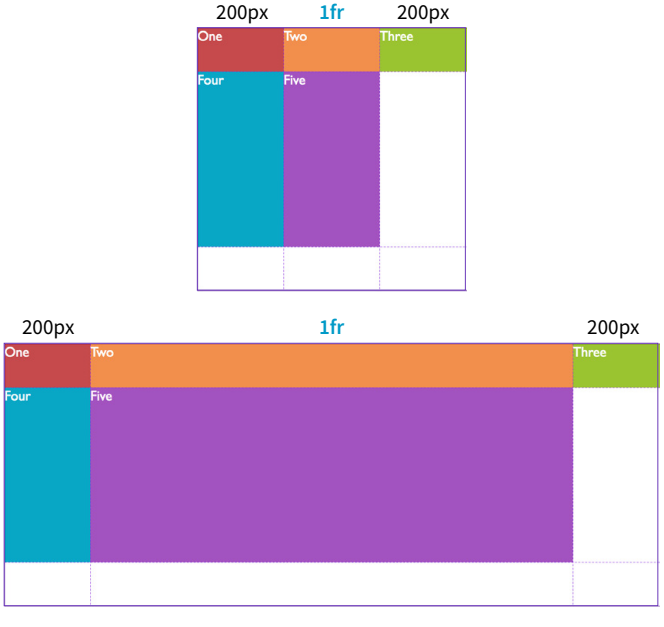


FIGURE 16-34. When the middle column has a track size of 1fr, it takes up the remaining space in the browser window and flexes to adapt to the browser width.

■ AT A GLANCE

Track Size Values

The Grid specification provides the following values for the **`grid-template-*`** properties:

- Lengths (such as **`px`** or **`em`**)
- Percentage values (%)
- Fractional units (**`fr`**)
- **`auto`**
- **`min-content`**, **`max-content`**
- **`minmax()`**
- **`fit-content()`**

The **fr** unit is great for combining fixed and flexible track widths, but I could also use all **fr** units to give all the columns proportional widths. In this example, all of the column widths flex according to the available browser width, but the middle column will always be twice the width of the side columns (see **Note**).

```
grid-template-columns: 1fr 2fr 1fr;
```

NOTE

Technically, the browser adds up the **fr** *units (4 in our example), divides the left-over space into that many portions, and then assigns the portions based on the number of units specified.*

Minimum and maximum size range

You can constrict the size range of a track by setting its minimum and maximum widths using the **minmax()** function in place of a specific track size.

```
grid-template-columns: 200px minmax(15em, 45em) 200px;
```

WARNING

fr *units are not permitted as the minimum value in a* **minmax()** *statement.*

This rule sets the middle column to a width that is at least 15em but never wider than 45em. This method allows for flexibility but allows the author to set limits.

Content-based sizing

The **min-content**, **max-content**, and **auto** values size the track based on the size of the content within it (FIGURE 16-35).

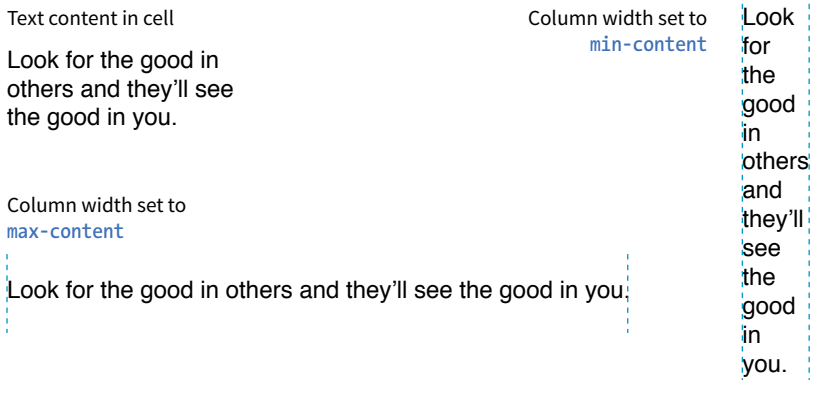


FIGURE 16-35. The **min-content** and **max-content** track sizing values.

The **min-content** value is the *smallest* that track can get without overflowing (by default, unless overridden by an explicit **min-width**). It is equivalent to the "largest unbreakable bit of content"—in other words, the width of the longest word or widest image. It may not be useful for items that contain normal paragraphs, but it may be useful in some cases when you don't want the track larger than it needs to be. This example establishes three columns, with the right column sized just wide enough to hold the longest word or image:

```
grid-template-columns: 50px 1fr min-content;
```

The **max-content** property allots the maximum amount of space needed for the content, even if that means extending the track beyond the boundaries of the grid container. When used as a column width, the column track will be

as wide as the widest content in that track *without* line wrapping. That means if you have a paragraph, the track will be wide enough to contain the text set on one line. This makes **max-content** more appropriate for short phrases or navigation items when you don't want their text to wrap (**auto** may work better because it allows wrapping if there's not enough room).

Using the **auto** keyword for a track size is basically like handing the keys over to the browser. In general, it causes the track to be sized large enough to accommodate its content, while taking into consideration what other restrictions are in place.

In the **minmax()** function, the **auto** keyword behaves very similarly to either **min-content** or **max-content**, depending on whether you put it in the minimum or maximum slot. As a keyword on its own, it functions similarly to **minmax(min-content, max-content)**, allowing the track to squeeze as narrow as it can without anything overflowing, but grow to fit its content without wrapping if there's enough space.

Unlike **max-content**, an **auto** maximum allows **align-content** and **justify-content** to stretch the track beyond the size of the content. As a minimum, it has a few more smarts than **min-content**—for example, using a specified **min-width** or **min-height** on an item (if any) instead of its **min-content** size, and ignoring the contents of any grid items with scrollbars.

If you want to size a track based on its content, but you're not sure which keyword to use, start with **auto**.

If you want to size a track based on its content, but you're not sure which keyword to use, start with auto.

Repeating track sizes

Say you have a grid that has 10 columns with alternating column widths, like so:

```
grid-template-columns: 20px 1fr 20px 1fr 20px 1fr 20px 1fr 20px 1fr
20px 1fr;
```

That's kind of a bummer to have to type out (I know, I just did it), so the fine folks at the W3C have provided a nice shortcut in the form of the **repeat()** function. In the previous example, the pattern "20px 1fr" repeats five times, which can be written as follows:

```
grid-template-columns: repeat(5, 20px 1fr);
```

Much better, isn't it? The first number indicates the number of repetitions, and the track sizes after the comma provide the pattern. You can use the **repeat()** notation in a longer sequence of track sizes—for example, if those 10 columns are sandwiched between two 200-pixel-wide columns at the start and end:

```
grid-template-columns: 200px repeat(5, 20px 1fr) 200px;
```

You can also provide grid line names before and/or after each track size, and those names will be repeated in the pattern:

```
grid-template-rows: repeat(4, [date] 5em [event] 1fr);
```

auto-fill and auto-fit

In the previous **repeat()** examples, we told the browser how many times to repeat the provided pattern. You can also let the browser figure it out itself based on the available space by using the **auto-fill** and **auto-fit** values instead of an integer in **repeat()**.

For example, if I specify

```
grid-template-rows: repeat(auto-fill, 10em);
```

and the grid container is 35em tall, then the browser creates a row every 10 ems until it runs out of room, resulting in three rows. Even if there is only enough content to fill the first row, all three rows are created and the space is held in the layout.

The **auto-fit** value works similarly, except any tracks that do not have content get dropped from the layout. If there is leftover space, it is distributed according to the vertical (**align-content**) and horizontal (**justify-content**) alignment values provided (we'll discuss alignment later in this section).

WARNING

You can only use one auto-repeat for a given declaration, and you cannot use it with **fr** *units. You also cannot put content-based size keywords inside an* **auto-fill** *or* **auto-repeat** *notation. Note that you can use* **minmax()** *notation inside an* **auto-repeat**, *and you can use it with* **fr***s or content-based keywords (***auto***,* **min-content**, **max-content***) if they're in the max position with a min length.*

Defining grid areas

So far we've been exploring how to divide a grid container into row and column tracks by using the **grid-template-columns** and **grid-template-rows** properties, and we've looked at many of the possible values for track dimensions. We've learned that you can assign names to individual grid lines to make them easy to refer to when placing items on the grid.

You can also assign names to *areas* of the grid, which for some developers is an even more intuitive method than calling out specific lines. Remember that a grid area is made up of one or more cells in a rectangle (no L-shapes or other non-rectangular collections of cells). Naming grid areas is a little funky to implement, but provides nice shortcuts when you need them.

To assign names to grid areas, use the **grid-template-areas** property.

grid-template-areas

Values:	none | *series of area names*
Default:	none
Applies to:	grid containers
Inherits:	no

The value of the property is a list of names provided for every cell in the grid, listed row by row, with each row in quotation marks. When neighboring cells share a name, they form a grid area with that name (see **Bonus Grid Line Names** sidebar).

In the following example, I've given names to areas in the example grid we've been working on so far (FIGURE 16-36). Notice that there is a cell name for each of the nine cells as they appear in each row. The row cell lists don't need

Bonus Grid Line Names

When you give an area a name with **grid-template-areas**, as an added bonus, you get a set of automatically generated grid line names to go with it. For example, when you name an area "main", the left and top grid lines of that area are automatically named "main-start," and the right and bottom grid lines are named "main-end." You can use those line names when positioning items.

The inverse is true as well. If you explicitly assign line names "portal-start" and "portal-end" around an area, you can use the area name "portal" to assign content to that area later, even if you haven't defined it with **grid-template-areas**. You can keep this shortcut in mind when naming grid lines, but it is not required.

This is a prime example of the flexibility and complexity of the Grid Layout Module.

to be stacked as I've done here, but many developers find it helpful to line up the cell by names using character spaces to better visualize the grid structure.

```
#layout {
  display: grid;
  grid-template-rows: [header-start] 100px [content-start] 400px
[footer-start] 100px;
  grid-template-columns: [ads] 200px [main] 1fr [links] 200px;
  grid-template-areas:
    "header  header  header"
    "ads     main    links"
    "footer  footer  footer";
}
```

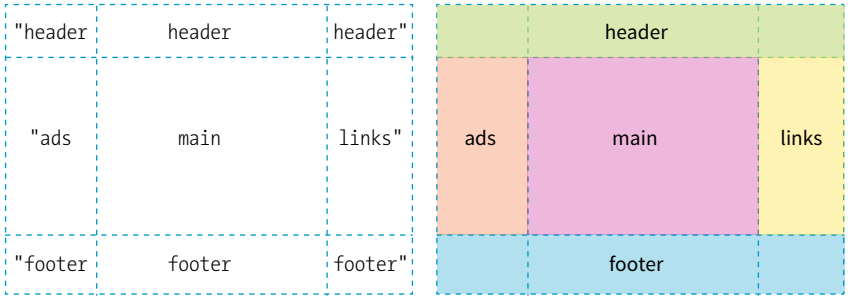


FIGURE 16-36. When neighboring cells have the same name, they form a named area that can be referenced later.

If there are three columns in the grid, there must be three names provided for each row. If you want to leave a cell unnamed, type one or more periods (.) in its place as a space holder so that every cell is still accounted for. Again, a sketch of your grid with the areas identified will make it easier to plan out the **grid-template-areas** value.

Be aware that the track sizes are still coming from the **grid-template-columns** and **grid-template-rows** properties. The **grid-template-areas** property simply assigns names to the areas, making it easier to plop items in them later.

The grid shorthand property

Use the **grid** shorthand property to set values for **grid-template-rows**, **grid-template-columns**, and **grid-template-areas** with one style rule. Bear in mind that any properties you do not use will be reset to their defaults, as is the case for all shorthands.

grid

Values: none | *row info / column info*

Default: none

Applies to: grid containers

Inherits: no

HEADS-UP

Be sure that you place the cell names in a way that forms rectangles when they combine to identify a named area. No L-shapes or fragments.

In **grid**, the row values and column values are separated by a slash, with the row values appearing first:

```
grid: rows / columns
```

It's easier to grasp without the clutter of line and area names, so here is the shorthand declaration for our example grid with just the row and column track information:

```
#layout {
  display: grid;
  grid: 100px 400px 100px / 200px 1fr 200px;
}
```

To include custom line names, add the names in brackets around their respective tracks, as we saw in the earlier named line example.

Including area names looks a little convoluted at first, but if you remember that you list cell names row by row, it makes sense that they appear with the other row information, before the slash. The complete order goes as follows:

```
[start line name] "area names" <track size> [end line name]
```

The line names and area names are optional. Repeat this for each row in the grid, simply listing them one after another with no special character separating rows. You may find it helpful to stack them as I've done in the following example to help keep each row distinct. When the rows are done, add a slash, and list the column track information after it. Here's a complete example of our grid written with the **grid** shorthand:

```
#layout {
  display: grid;
  grid:
    [header-start]  "header  header  header" 100px
    [content-start] "ads     main    links"  400px
    [footer-start]  "footer  footer  footer" 100px
    /[ads] 200px [main] 1fr [links] 200px; }
```

This expands to the following:

```
#layout {
  display: grid;
  grid-template-rows: [header-start] 100px [content-start] 400px
[footer-start] 100px;
  grid-template-columns: [ads] 200px [main] 1fr [links] 200px;
  grid-template-areas:
    "header  header  header"
    "ads     main    links"
    "footer  footer  footer" }
```

There is also a **grid-template** property that works exactly like **grid**, but it may be used only with explicitly defined grids (as opposed to implicit grids, which I cover later). The Grid Layout spec strongly recommends that you use the **grid** shorthand instead of **grid-template** (see **Note**) unless you specifically want the cascading behavior of **grid-template**.

I'm thinking that it's a good time for you to put all of these grid setup styles to use in EXERCISE 16-4.

NOTE

The Grid experts I've talked to don't tend to use **grid** *or* **grid-template** *except for the simplest of grid structures. The code becomes overly complex, and one small slip can make the whole grid fall apart. For complicated grid structures, stick to separate properties for defining rows, columns, and areas.*

EXERCISE 16-4. Setting up a grid

In this exercise, we'll set up the grid template for the page shown in FIGURE 16-37. We'll place the grid items into the grid in EXERCISE 16-5, so for now just pay attention to setting up the rows and columns.

This page is similar to the bakery page we've been working on, but it has a few more elements and whitespace to make things interesting. The starter document, *grid.html*, is provided with the exercise materials at *learningwebdesign.com/5e/materials*. Open it in a text editor, and you'll see that all of the styles affecting the appearance of each element are provided.

FIGURE 16-37. The Breads of the World page that we will create using Grid Layout.

1. Start by turning the containing element, the **`#layout div`**, into a grid container by setting its display mode to "grid":

```
#layout {
  …
  display: grid;
}
```

2. FIGURE 16-37 shows the row and column tracks required to accommodate the content in the desired layout. Start by defining the rows as specified in the sketch, using the **`grid-template-rows`** property. There should be six values, representing each of the six rows. (Spoiler alert: we'll be tweaking these values when we get to the next exercise. This is just a starting point.)

```
#layout {
  …
  display: grid;
  grid-template-rows: 3em 20px 150px 300px 5em;
}
```

3. Do the same for the seven columns. Because I want the text column to grow and shrink with the available space, I've specified its width in fractional units (**`1fr`**). The remaining columns create 150px-wide cells for three images and 20px of space before them.

NOTE

You will need to use a browser that supports grids for this exercise. I am using Firefox in order to take advantage of the Grid Inspector tool. Supporting browsers are listed earlier in this section. See the ***"Firefox Grid Inspector and Layout Panel"*** *sidebar for instructions on how to open the tool.*

EXERCISE 16-4. Continued

You can write them all out like this:

```
grid-template-columns: 1fr 20px 150px 20px 150px 20px 150px;
```

However, because the last six columns are a repeating pattern, it would be easier to use the **`repeat()`** function to repeat the spaces and figure columns three times:

```
grid-template-columns:  1fr repeat(3, 20px 150px);
```

4. Finally, let's assign names to the grid lines that border the grid area where the **`main`** content element should appear. The names give us some intuitive options for placing that item later. The main area starts at the third row track, so assign the name "main-start" to the grid line between the second and third row track measurements:

   ```
   grid-template-rows: 3em 20px [main-start] 150px 300px 5em;
   ```

   The main area extends into the last row track, so assign the name "main-end" to the last grid line in the grid (after the last row track):

   ```
   grid-template-rows: 3em 20px [main-start] 150px 300px 5em [main-end];
   ```

5. Now do the same for the grid lines that mark the boundaries of the column track where the main content goes:

   ```
   grid-template-columns:  [main-start] 1fr [main-end] repeat(3, 20px
   150px);
   ```

I've saved my work and looked at it in Firefox with the Grid Inspector turned on (FIGURE 16-38). Because I haven't specified where the grid items go, they flowed into the cells sequentially, making the mess you see in the figure. However, the grid overlay reveals that the structure of the grid looks solid. Save the file and hold on to it until the next exercise.

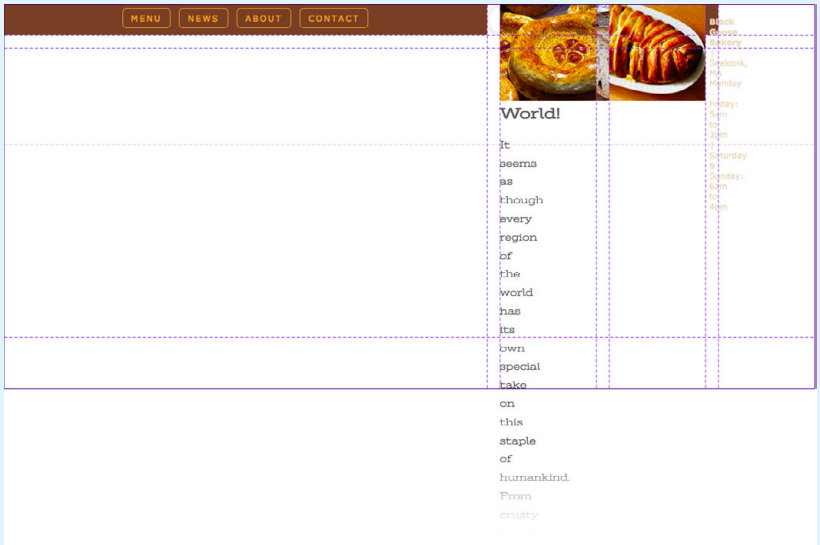


FIGURE 16-38. The grid items are not placed correctly yet, but the Firefox Grid Inspector shows that the grid is set up correctly.

Placing Grid Items

Now that we've covered all the ins and outs of setting up a grid, including giving ourselves handy line and area names, we can move on to assigning items to areas on the grid.

As we saw in FIGURES 16-32 and 16-38, without any explicit placement instruction, grid items flow into the available grid cells sequentially. That's fine for some use cases, but let's tell our grid items where to go!

Positioning using lines

One method for describing a grid item's location on the grid is to specify the four lines bordering the target grid area with four properties that specify the start and end row lines and the start and end column lines. Apply these properties to the individual grid item element you are positioning.

`grid-row-start`
`grid-row-end`
`grid-column-start`
`grid-column-end`

Values: auto | *grid line* | span *number* | span *'line name'* | *number 'line name'*

Default: auto

Applies to: grid items

Inherits: no

This set of properties provides a straightforward way to describe an element's position on the grid by identifying either the name or number of the grid line on each border. As an alternative, you can provide just one line identifier and tell the item to "span" a certain number of cells. By default, an item occupies one track width, which is what you get with the **auto** keyword.

Getting back to our five-item example, I would like the first item to go in the top row and span across all three columns (FIGURE 16-39).

One way to do this is to use the four line start/end properties and identify lines by their numbers like so:

```
#one {
  grid-row-start: 1;
  grid-row-end: 2;
  grid-column-start: 1;
  grid-column-end: 4;
}
```

Take a moment to compare this to the position of the **#one div** back in FIGURE 16-36. For **grid-row-start**, the 1 value refers to the first (top) line of the grid container. For **grid-column-start**, 1 refers to the first line on the left edge of the container, and the value 4 for **grid-column-end** identifies the fourth and last line on the right edge of the container.

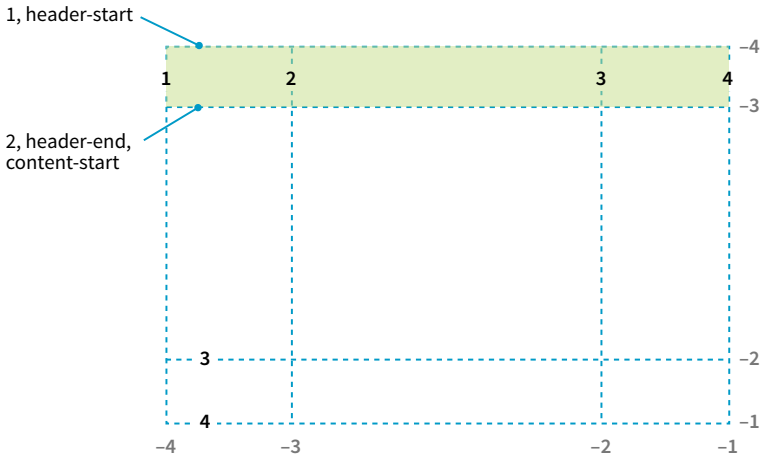


FIGURE 16-39. Positioning a grid item across the top row track in our sample grid.

Here's one more for good measure. This style declaration positions the **#four** item element in the right side column as shown in FIGURE 16-36:

```
#four {
  grid-row-start: 2;
  grid-row-end: 3;
  grid-column-start: 3;
  grid-column-end: 4;
}
```

Remember how grid lines are also numbered in the opposite direction starting at –1? We can use that here. I could specify the **grid-column-end** for **#one** as –1, and it would be the same as 4. In fact, this method has the advantage of guaranteeing to span to the end of the track and avoids miscounting.

I could also use the named lines I set up here. These row values are interchangeable with the previous example:

```
#one {
  grid-row-start: header-start;
  grid-row-end: header-end;
  …
}
```

If I omit the end line declaration, the row would be one track high (the default). That's what I want here, so omitting the end declaration altogether is one more way to achieve the effect I want.

Ready for yet another option? I can tell the item what line to start on, but instead of providing an end line, I can use the **span** keyword to specify how many tracks to span over. In this example, the item starts at the left edge of the track (line 1) and spans over three columns, effectively ending at line 4.

NOTE

If you omit a start or end line, the area will be one track wide (the default, **auto***).*

■ HOT TIP

If you need to span to the last grid line in a row or column, use the value –1 and save yourself some counting. Also, even if the number of rows or columns changes down the line, –1 will always select the last line, so you won't need to renumber.

```
#one {
  …
  grid-column-start: 1;
  grid-column-end: span 3;
}
```

Spans can work in reverse as well. If you provide only an end line, the span searches toward the start of the track. The following styles have the same effect as our previous examples because they define the target area by its end line at the far right of the grid and span back three columns to the beginning:

```
#one {
  …
  grid-column-start: span 3;
  grid-column-end: -1;
}
```

If four declarations feels like too many, use the shorthand **grid-row** and **grid-column** properties instead.

grid-row
grid-column

Values: *start line* / *end line*

Default: see individual properties

Applies to: grid items

Inherits: no

These properties combine the ***-start** and ***-end** properties into a single declaration. The start and end line values are separated by a slash (/). With the shorthand, I can shorten my example to the following two declarations. Any of the methods for referring to lines work in the shorthand values.

```
#one {
  grid-row: 1 / 2;
  grid-column: 1 / span 3;
}
```

Positioning by area

The other way to position an item on a grid is to tell it go into one of the named areas by using the **grid-area** property.

grid-area

Values: *area name* | *1 to 4 line identifiers*

Default: see individual properties

Applies to: grid items

Inherits: no

The **grid-area** property points to one of the areas named with **grid-template-areas**. It can also point to an area name that is implicitly created when you name lines delimiting an area with the suffixes "-start" and "-end". With

this method, I can drop all of the grid items into the areas I set up with my template earlier (FIGURE 16-40):

```
#one { grid-area: header; }
#two { grid-area: ads; }
#three { grid-area: main }
#four { grid-area: links; }
#five { grid-area: footer; }
```

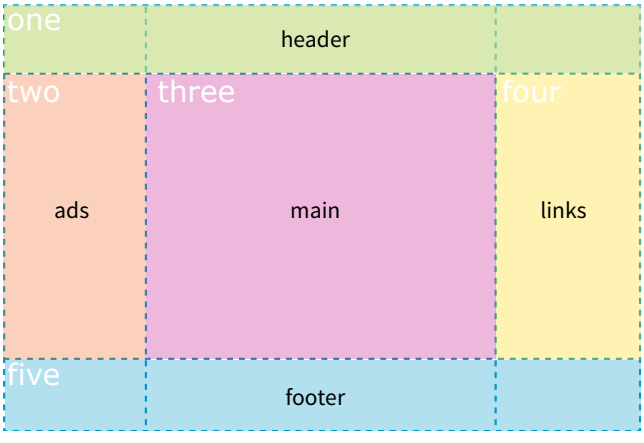


FIGURE 16-40. Assigning grid items by area names.

How easy was *that*?! One benefit of using areas is that you can change the grid, and as long as you provide consistently named grid areas, the items will end up in the right place. There's no need to renumber lines in the style sheet.

You can also use **grid-area** to provide a list of four grid lines that define an area, separated by slashes. The order in which they appear is "row-start," "column-start," "row-end," "column-end" (counterclockwise from the top). There are a lot of rules for what happens when you omit values, but I'm not going to get into all those finer points here. The **grid-area** declaration for the first grid item could be written like this to achieve the same result as previous examples:

```
#one {
  grid-area: 1 / 1 / 2 / span 3;
  /* row-start / column-start / row-end / column-end */
}
```

As you can see, the Grid Layout Module gives you a variety of ways to set up a grid and a variety of ways to place items on it. In fact, the spec includes a few more uses of **span** that you can explore. Choose the methods that work best for the grid you are designing or that work best for your brain.

Now let's finish up the grid we've been working on in EXERCISE 16-5.

EXERCISE 16-5. Placing items on a grid

Now that we have the grid set up for the Breads of the World page, we can place items into the correct grid areas by using line numbers and names.

I'm going to go through them quickly, but feel free to save the file and look at the page in a grid-supporting browser at any step along the way. Refer to the finished layout in FIGURE 16-41 for the final item positions and line number hints.

1. Open *grid.html* in your text editor if it isn't open already. We'll start by placing the **nav** element into the first row of the grid, using the four grid line properties:

```
nav {
  grid-row-start: 1;
  grid-row-end: 2;
  grid-column-start: 1;
  grid-column-end: 8; /* you could also use -1 */
}
```

2. Now place the figures in their positions on the grid. Start by putting the third figure (**#figC**) in its place in the far-right column by using the shorthand **grid-row** and **grid-column** properties. It goes between the 3rd and 4th row grid lines and extends from the 7th to 8th column lines. For columns, instead of 7 and 8, use the negative value for the last line and span it one space to the left to get to the starting point:

```
#figC {
  grid-row: 3 / 4;
  grid-column: span 1 / -1;
}
```

   Now position the **#figA** and **#figB** elements by using the **grid-area** property with line values. Remember that the values go in the order top, left, bottom, right (counterclockwise around the area).

```
#figA {
  grid-area: 3 / 3 / 4 / 4;
}
#figB {
  grid-area: 3 / 5 / 4 / 6;
}
```

3. We gave the grid lines around the main area names, so let's use them to place the **main** grid item:

```
main {
  grid-row: main-start / main-end;
  grid-column: main-start / main-end;
}
```

   Do you remember that when you name lines around an area ***-start** and ***-end**, it creates an implicitly named area *****? Because we named the lines according to this syntax, we could also place the **main** element with **grid-area** like this:

```
main {
  grid-area: main;
}
```

4. Finally, we can put the footer into its place. It starts at the last row grid line and spans back one track. For columns, it starts at the third line and goes to the last. Here is one way to write those instructions. Can you come up with others that achieve the same result?

```
footer {
  grid-row: 5 / 6;
  grid-column: 3 / -1;
}
```

Save your file and look at it in the browser. You may spot a problem, depending on the width of your browser window. When

FIGURE 16-41. The final Breads of the World grid layout.

EXERCISE 16-5. Continued

the browser is wide, the layout works fine, but when it is made narrower, the text in the **main** element overflows its cell. That's because the 300-pixel height we gave that row is not sufficient to hold the text when it breaks onto additional lines or is resized larger.

5. We can fix that by changing the measurement of the fifth row track to **auto**. In that way, the height of that row will always be at least big enough to hold the content. The **min-content** value would work as well, but **auto** is always the first value to try:

```
#layout {
  display: grid;
  grid-template-rows: 3em 20px
[main-start] 150px auto 5em
[main-end];
…
}
```

If you reload the page in the browser, the text is always contained in its grid area, regardless of the width of the window. Everything should fall into place nicely, as shown in FIGURE 16-41.

You now have your first grid layout under your belt. This exercise gives you only a taste of what Grid Layout can do, but we've covered the fundamentals of setting up a grid and placing items in it. You're off to a great start!

Now you know the basics of creating an explicit grid and placing items on it. There are a few more grid-related topics that are important to be familiar with: implicit grids, gutter spaces, and grid alignment. I have space for only a basic introduction to each topic, but when you start implementing grid layouts on your own, you can do the deep dive required to meet your needs.

Implicit Grid Behavior

So far, we've been focusing on ways to define an explicit grid and place items on it deliberately. But along the way, we've encountered a few of the Grid system's automatic, or implicit, behaviors. For example, without explicit placement instructions, grid items flow into the grid sequentially, as we saw in FIGURE 16-32. I also pointed out how creating a named area implicitly generates grid lines with the "-start" and "-end" suffixes, and vice versa.

Another implicit Grid behavior is the creation of row and column tracks on the fly to accommodate items that don't fit in the defined grid. For example, if you place an item outside a defined grid, the browser automatically generates tracks in the grid to accommodate it. Similarly, if you simply have more items than there are cells or areas, the browser generates more tracks until all the items are placed.

By default, any row or column automatically added to a grid will have the size **auto**, sized just large enough to accommodate the height or width of the contents. If you want to give implicit rows and columns specific dimensions, such as to match a rhythm established elsewhere in the grid, use the **grid-auto-*** properties.

`grid-auto-rows`
`grid-auto-columns`

Values:	*list of track sizes*
Default:	auto
Applies to:	grid containers
Inherits:	no

The **grid-auto-row** and **grid-auto-columns** properties provide one or more track sizes for automatically generated tracks and apply to the grid container. If you provide more than one value, it acts as a repeating pattern. As just mentioned, the default value is **auto**, which sizes the row or column to accommodate the content.

In this example, I've explicitly created a grid that is two columns wide and two columns high. I've placed one of the grid items in a position equivalent to the fifth column and third row. My explicit grid isn't big enough to accommodate it, so tracks get added according to the sizes I provided in the **grid-auto-*** properties (FIGURE 16-42).

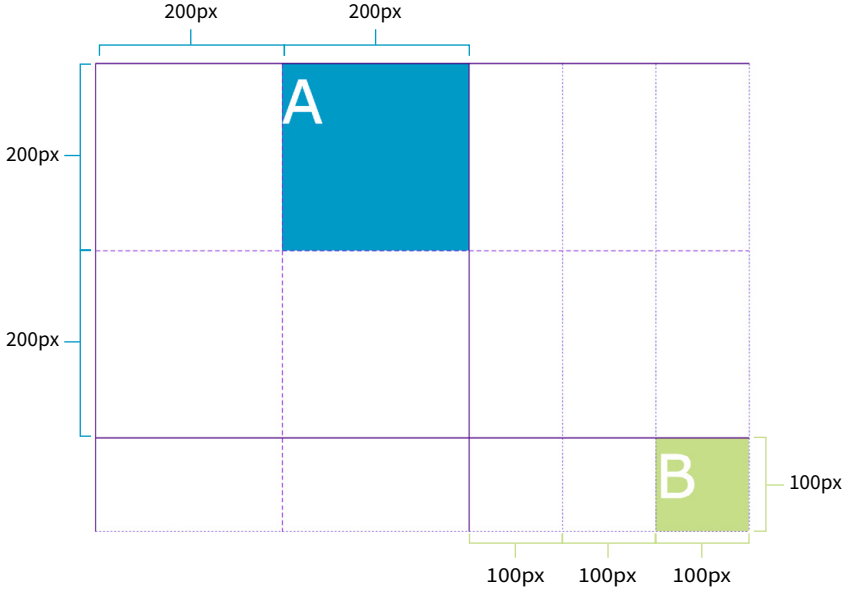
THE MARKUP

```
<div id="littlegrid">
  <div id="A">A</div>
  <div id="B">B</div>
</div>
```

THE STYLES

```
#littlegrid {
  display: grid;
  grid-template-columns: 200px 200px;
  grid-template-rows: 200px 200px;
  grid-auto-columns: 100px;
  grid-auto-rows: 100px;
}
#A {
  grid-row: 1 / 2;
  grid-column: 2 / 3;
}
#B {
  grid-row: 3 / 4;
  grid-column: 5 / 6;
}
```

The grid has two explicitly defined rows and columns at 200 pixels wide each.



Rows and column tracks are added automatically as needed. They are sized as specified by `grid-auto-rows` and `grid-auto-columns` (100 pixels).

FIGURE 16-42. Browsers generate rows and columns automatically to place grid items that don't fit the defined grid.

Hopefully, that example helped you form a mental model for automatically generated rows and columns. A more common use of auto-generated tracks is to tile images, product listings, and the like into columns, letting rows be

The Grid Item Shuffle

So far, you've seen grid items flowing into a grid sequentially and get placed into their own little areas on a grid explicitly. There are a few properties that may be useful for tweaking the position of grid items.

Changing the Order

As in Flexbox, you can apply the **`order`** property to a grid item element to change the order in which it appears when it is rendered. Keep in mind that the **`order`** property does not change the order in which it is read by an assistive device. See the section **"Changing the Order of Flex Items"** earlier in this chapter for more information on how to use **`order`**.

Stacking Order

It is possible to position items in a grid in a way that causes them to overlap. When more than one item is assigned to a grid area, items that appear later in the source are rendered on top of items earlier in the source, but you can change the stacking order by using the **`z-index`** property. Assigning a higher **`z-index`** value to earlier item elements makes them render above items that appear later. See the section **"Stacking Order"** in **Chapter 15, Floating and Positioning**, for details on using **`z-index`**.

created as needed. These styles set up a grid with explicit columns (as many as will fit the width of the viewport, no narrower than 200px) and as many 200px-high rows as needed:

```
grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));
grid-auto-rows: 200px;
```

You can also control the manner in which items automatically flow into the grid with the **grid-auto-flow** property.

Flow direction and density

grid-auto-flow

Values: row or column | dense *(optional)*

Default: row

Applies to: grid containers

Inherits: no

Use **grid-auto-flow** to specify whether you'd like items to flow in by row or column. The default flow follows the writing direction of the document (left-to-right and top-to-bottom for English and other left-to-right languages).

In this example, I've specified that I'd like grid items to flow in by columns instead of the default rows:

```
#listings {
  display: grid;
  grid-auto-flow: column;
}
```

By default, items are placed in the first area in which they fit. Cells that are too small to accommodate the content will be skipped over until a cell large enough is found for placement. If you include the optional **dense** keyword for the **grid-auto-flow** property, it instructs the browser to fill the grid as densely as possible, allowing the items to appear out of sequence in order to fill the available space:

```
#listings {
  display: grid;
  grid-auto-flow: dense rows;
}
```

The example on the left of FIGURE 16-43 shows the default flow method. Look closely and you'll see that the grid items are in order. When there isn't enough room for the whole item, it moves down and to the left until it fits (similar to floats). This method may leave empty cells as shown in the figure. By comparison, the dense flow example on the right is all filled in, and if you look at the numbering, you can see that putting items wherever they fit makes them end up out of order. Note that dense flow doesn't always result in a completely filled-in grid like the figure, but it is likely to have fewer holes and be more compact than the default mode.

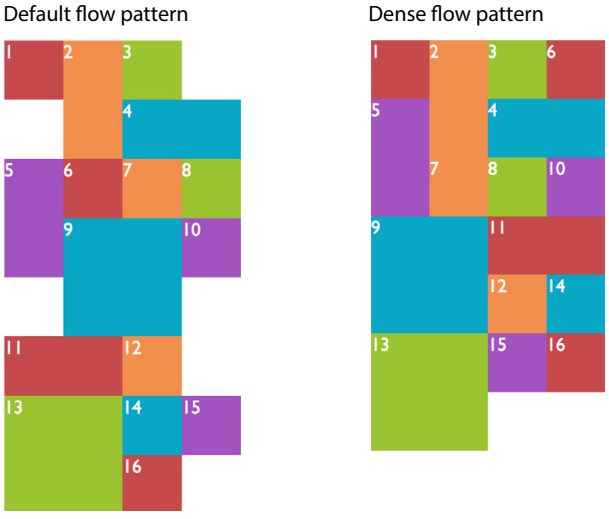


FIGURE 16-43. Comparison of default and dense auto-flow modes.

The grid shorthand property revisited

Earlier we saw the **grid** shorthand property used to provide track sizes as well as area names. In that section, we were dealing with explicit grids, but **grid** can be used with implicit grid properties as well.

Adding the **auto-flow** keyword to either the row or track information indicates that the tracks on that axis should be automatically generated at the provided dimension.

Say we want to establish columns explicitly, but let rows generate automatically as needed. The **grid** shorthand for this common scenario is shown here:

```
grid: auto-flow 12em / repeat(5, 1fr);
```

Remember that the **grid** shorthand syntax lists row information first, then a slash, then the column information. Here, the rule says to create rows automatically that are 12 ems high and create 5 columns at 1fr each. When **auto-flow** is applied to rows, the **grid-auto-flow** is set to **row**.

In this example, the resulting grid will have two 300px rows, but 100px-wide columns will be generated on the fly as grid items are added:

```
grid: 300px 300px / auto-flow 100px;
```

With **auto-flow** applied to columns, the **grid-auto-flow** is set to **column**.

It is important to keep in mind that because **grid** is a shorthand property, any omitted value will be reset to its default. Therefore, if you've also used **grid** to set up explicit rows and columns, those will be essentially lost if a **grid** shorthand with implicit grid instructions appears later in the style sheet.

Spacing and Alignment

The remaining properties defined in the Grid Layout Module relate to spacing and alignment. You can add space between tracks and adjust alignment of the grid and its items by using many of the same methods you learned for Flexbox.

Spacing between tracks (gutters)

NOTE

These property names will be changing to **row-gap**, **column-gap**, *and* **gap**. *Until browsers start supporting the new syntax, you can still use the* **grid-*** *prefixed versions, which will continue to be supported for backward compatibility.*

`grid-row-gap`
`grid-column-gap`

Values: *length (must not be negative)*
Default: 0
Applies to: grid containers
Inherits: no

`grid-gap`

Values: *grid-row-gap grid-column-gap*
Default: 0 0
Applies to: grid containers
Inherits: no

Setting a length value for **grid-row-gap** adds space between the row tracks of the grid, and **grid-column-gap** adds space between (you guessed it) column tracks. The effect is as if the grid lines have a width; however, the gap width is applied only to lines between tracks, not outside the first and last lines in the grid. (Spacing on the outside edges can be controlled with padding.) You can use the **grid-gap** shorthand to specify gap widths for rows and columns in one go, with rows first, as usual.

In this example, I've added 20px space between rows and 50px space between columns by using the **grid-gap** shorthand (FIGURE 16-44).

```
div#container {
  border: 2px solid gray;
  display: grid;
  grid: repeat(4, 150px) / repeat(4, 1fr);
  grid-gap: 20px 50px;
 }
```

Box Alignment

It's no coincidence that Flexbox and Grid share alignment properties and values. They are all standardized in their own spec called the CSS Box Alignment Module, Level 3, which serves as a reference to a number of CSS modules. You can check it out at *www.w3.org/TR/css-align/*.

Grid and item alignment

You can align grid items in their cells with the same alignment vocabulary used for Flexbox items (see the **"Box Alignment"** sidebar). I'm going to touch on these quickly, but you can play around with them on your own.

FIGURE 16-44. Grid gaps add gutter spaces between tracks.

Aligning individual items

justify-self

Values: start | end | center | left | right | self-start | self-end | stretch | normal | auto

Default: auto (looks at the value for justify-items, which defaults to normal)

Applies to: grid items

Inherits: no

align-self

Values: start | end | center | left | right | self-start | self-end | stretch | normal | auto

Default: auto (looks at the value for align-items)

Applies to: grid items

Inherits: no

When a grid item doesn't fill its entire grid area, you can specify how you'd like it to be aligned in that space. Specify the horizontal (inline) alignment with the **justify-self** property. **align-self** specifies alignment on the vertical (block) axis. These properties apply to the grid item element, which makes sense because you want the item to align itself.

FIGURE 16-45 shows the effects of each keyword value. For items with their size set to **auto** (or in other words, not explicitly set with **width** and **height** properties), the default is **stretch**. This is what we've seen in all the previous

NOTE

The **self-start** *and* **self-end** *values look at the writing direction of the content of the item and use its start or end edge for alignment. For example, if an item is in Arabic, its* **self-start** *edge is on the right, and it would be aligned to the right. The* **start** *and* **end** *values consider the writing direction of the grid container. The* **left** *and* **right** *keywords are absolute and would not change with the writing system, but they correspond to* **start** *and* **end** *in left-to-right languages.*

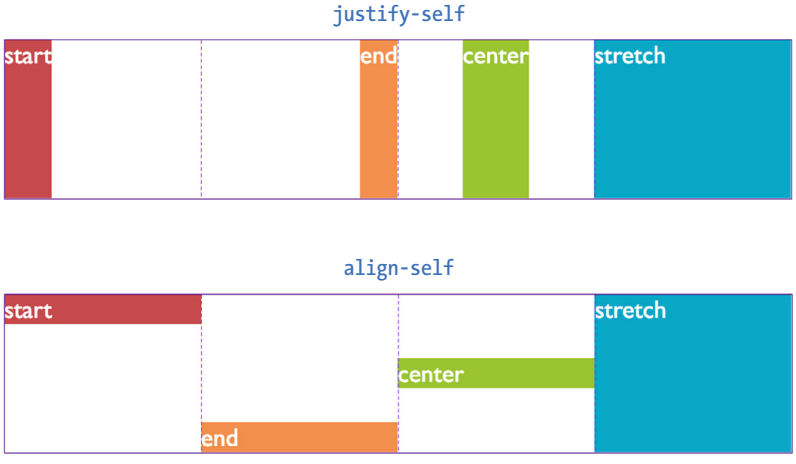


FIGURE 16-45. Values for **justify-self** and **align-self** for aligning a grid item within its respective grid area. These values have the same use in the **justify-items** and **align-items** properties that are used to align all the items in the grid.

> **TIP**
>
> If you want a grid item to stay centered in its grid area, set both **align-self** and **justify-self** to **center**.

grid examples. If the grid item has a width and height specified, those dimensions are preserved and the default is **start**.

After reading about Flexbox, you should find these familiar—for example, the use of "start" and "end" to keep the system language direction-agnostic.

Aligning all the items in a grid

justify-items

Values: start | end | center | left | right | self-start | self-end | stretch | normal

Default: normal (stretch for non-replaced elements; start for replaced elements)

Applies to: grid containers

Inherits: no

align-items

Values: start | end | center | left | right | self-start | self-end | stretch | normal

Default: normal (stretch for non-replaced elements; start for replaced elements)

Applies to: grid containers

Inherits: no

To align all of the items in a grid in one fell swoop, use the **justify-items** property for horizontal/inline axis alignment and **align-items** for vertical/block axis. Apply these properties to the grid container element so it affects all of the items in the grid. The keywords do the same things shown in

Speaking of Spacing, What About Margins?

You can add margins to a grid item as you can for any other element. It is useful to know that the item's margin box will be anchored to the cell or grid area, and the margin space is preserved.

You can use margins to move the item around in the grid area. For example, setting the left margin to "auto" pushes the item to the right, as we saw in earlier Flexbox examples. Setting the left and right margins to "auto" (as long as item has a specified width) centers it horizontally. In Grid, you can also set the top and bottom margins to "auto" and, as long as there's a specified height, it centers vertically. Of course, you have the grid item alignment properties to achieve these effects as well.

FIGURE 16-43; just picture it happening consistently across the entire grid. Keep in mind that these settings will be overridden by the `*-self` properties.

Aligning tracks in the grid container

There may be instances in which the tracks of your grid do not fill the entire area of their grid container—for example, if you've specified track widths and heights in specific pixel measurements. You can decide how the browser should handle leftover space within the container by using the **`justify-content`** (horizontal/inline axis) and **`align-content`** (vertical/block axis) properties.

Alignment properties apply to the grid container.

justify-content

Values: `start` | `end` | `left` | `right` | `center` | `stretch` | `space-around` | `space-between` | `space-evenly`

Default: `start`

Applies to: grid containers

Inherits: no

align-content

Values: `start` | `end` | `left` | `right` | `center` | `stretch` | `space-around` | `space-between` | `space-evenly`

Default: `start`

Applies to: grid containers

Inherits: no

NOTE

The **`stretch`** *keyword works only when the track width or height is set to* **`auto`**.

In FIGURE 16-46, the grid container is indicated with a gray outline. The rows and columns of the drawn grid do not fill the whole container, so something has to happen to that extra space. The **`start`**, **`end`**, and **`center`** keywords move the whole grid around within the container by putting the extra space after, before, or equally on either side, respectively. The **`space-around`** and **`space-between`** keywords distribute space around tracks as discussed in the Flexbox section. The **`space-evenly`** keyword adds an equal amount of space at the start and end of each track and between items.

WARNING

When you distribute space around and between tracks, it adds to whatever **`gap`** *settings you may have.*

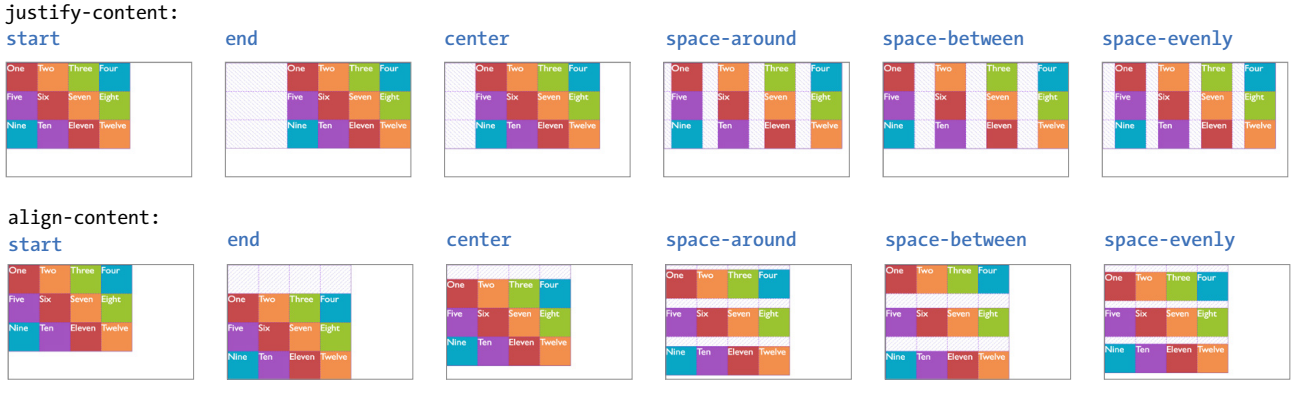


FIGURE 16-46. The **`justify-content`** and **`align-content`** properties distribute extra space in the container.

Before we close out this discussion of Grid Layout, let's give the Black Goose Bakery page a nice two-column layout in EXERCISE 16-6.

EXERCISE 16-6. A grid layout for the bakery page

The Black Goose Bakery page has come a long way. You've added padding, borders, and margins. You've floated images, positioned an award graphic, and created a navigation bar by using Flexbox. Now you can use your new grid skills to give it a two-column layout that would be appropriate for tablets and larger screens (FIGURE 16-47).

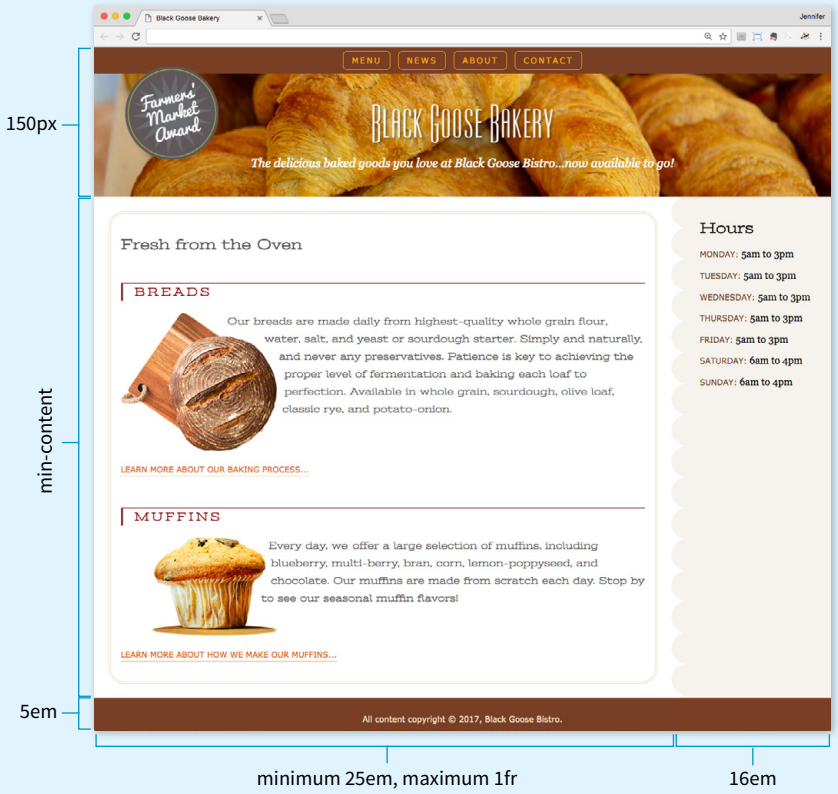


FIGURE 16-47. The Black Goose Bakery page with a two-column grid layout.

Start by opening the bakery file as you left it in EXERCISE 16-1.

1. We need to add a bit of markup that encloses everything in the body of the document in an element that will serve as the grid container. Open the HTML document *bakery.html*, add a **`div`** around all of the content elements (from **`header`** to **`footer`**), and give it the **`id`** "container". Save the HTML file.

```
<body>
  <div id="container">
    <header>…</header>
    <main>…</main>
    <aside>…</aside>
    <footer>…</footer>
  </div>
</body>
```

In the style sheet (*bakery-styles.css*), add a new style to make the new **div** display as a grid:

```
#container {
  display: grid;
}
```

2. First we'll work on the rows. FIGURE 16-47 shows that we need three rows to create the layout. Set the height of the first row track to **auto** so it will observe the height settings on the elements within it and automatically accommodate the content. The second row has a lot of text, so use the **auto** track value again to guarantee the track will expand at least as much as necessary to fit the text. For the third row, a height of 5em should be sufficient to fit the few lines of text with a comfortable amount of space:

```
#container {
  display: grid;
  grid-template-rows: auto auto 5em;
}
```

3. Now we can set up the column tracks. It looks like we'll need only two: one for the main content and one for the Hours sidebar. I've used the **minmax()** value so I can ensure the text column never gets narrower than 25em, but it can expand to fill the available space in the browser (**1fr**). The Hours column feels right to me at 16em. Feel free to try other values.

```
#container {
  display: grid;
  grid-template-rows: auto auto 5em;
  grid-template-columns: minmax(25em, 1fr) 16em;
}
```

4. Next, name the areas in the grid so we can place the items in it easily and efficiently. Use the **grid-template-areas** property to name the cells in the grid:

```
#container {
  display: grid;
  grid-template-rows: auto auto 5em;
  grid-template-columns: minmax(25em, 1fr) 16em;
  grid-template-areas:
    "banner banner"
    "main   hours"
    "footer footer";
}
```

5. With everything set up, it'll be a breeze to put the content items into their proper places. Create a style rule for each grid item and tell it where to go with **grid-area**:

```
header {
  grid-area: banner;
}
main {
  grid-area: main;
}
aside {
  grid-area: hours;
}
footer {
  grid-area: footer;
}
```

Pretty easy, right? Now would be a good time to save the file and take a look at it in the browser (if you haven't already). The 2.5% margins that we had set on the **main** element earlier give it some nice breathing room in its area, so let's leave that alone. However, I'd like to remove the right margin and border radius we had set on the **aside** so it fills the right column. I'm going to just comment them out so that information is still around if I need to use it later:

```
aside {
  …
  /* border-top-right-radius: 25px; */
  /* margin: 1em 2.5% 0 10%; */
}
```

That does it! Open the *bakery.html* page in a browser that supports CSS grids, and it should look like the screenshot in FIGURE 16-47.

Now the bakery page has a nice two-column layout using a simple grid. In the real world, this would be just one layout in a set that would address different screen sizes as part of a responsive design strategy. We'll be talking about responsive design in the next chapter. And because grids are not supported by Internet Explorer, Edge, and older browsers, you would also create fallback layouts using Flexbox or floats depending on how universally you need the layout to work. I don't mean to kill your buzz, but I do want you to be aware that although this exercise let you sharpen your skills, it's part of a much broader production picture.

Note: For float- and position-based layout techniques that could be used as fallbacks, get the article **"Page Layout with Floats and Positioning"** (PDF) at learningwebdesign.com/articles/.

Online Grid Resources

As you continue your Grid Layout adventure, I'm sure you'll find plenty of excellent resources online, as more are popping up all the time. I'd like to point you to a few of the most complete and authoritative resources that I found helpful as I learned about grids myself.

***Rachel Andrew's "Grid By Example" site** (gridbyexample.com)*

Rachel Andrew, one of the first champions of Grid Layout, has assembled an incredible collection of articles, free video tutorials, browser support information, and more. You can also try searching the web for her excellent conference talks on the topic.

***Jen Simmons' "Experimental Layout Lab"** (labs.jensimmons.com)*

Jen Simmons, who works for Mozilla Foundation, shows off what Grid Layout can do in her Experimental Layout Lab. It's definitely worth a visit for the cool examples of Grid and other emerging CSS technologies as well as exercises that let you code along.

You can find Jen's many articles on CSS Grid at *jensimmons.com/writing*. I also recommend her roundup of resources for learning Grid Layout at *jensimmons.com/post/feb-27-2017/learn-css-grid*. See also Jen's YouTube video series called "Layout Land" (*youtube.com*, search for "Layout Land Jen Simmons").

***Grid Garden by Thomas Park** (cssgridgarden.com)*

If you enjoyed the Flexbox Froggy game created by Thomas Park, you will have fun playing his Grid Garden game for getting familiar with CSS Grid Layout.

AT A GLANCE

Grid Property Roundup

Here's a nice, handy list of the Grid properties, organized by whether they apply to the container or to individual grid items.

Grid Container Properties

```
display: grid | inline-grid
grid
  grid-template
    grid-template-rows
    grid-template-columns
    grid-template-areas
  grid-auto-rows
  grid-auto-columns
  grid-auto-flow
grid-gap
  grid-row-gap
  grid-column-gap
justify-items
align-items
justify-content
align-content
```

Grid Item Properties

```
grid-column
  grid-column-start
  grid-column-end
grid-row
  grid-row-start
  grid-row-end
grid-area
justify-self
align-self
order (not part of Grid Module)
z-index (not part of Grid Module)
```

TEST YOURSELF

We covered lots of ground in this chapter. See how you do on this quiz of some of the highlights. As always, **Appendix A** has the answers.

Flexbox

1. How do you turn an element into a flex item?

2. Match the properties with their functions.

a. `justify-content`	1. Distribute space around and between flex lines on the cross axis.
b. `align-self`	2. Distribute space around and between items on the main axis.

c. `align-content` 3. Position items on the cross axis.

d. `align-items` 4. Position a particular item on the cross axis.

3. How is **`align-items`** different from **`align-content`**? What do they have in common?

4. Match the properties and values to the resulting effects.

a. `flex: 0 1 auto;` 1. Items are completely inflexible, neither shrinking nor growing.

b. `flex: none;` 2. Item will be twice as wide as others with `flex: 1` and may also shrink.

c. `flex: 1 1 auto;` 3. Items are fully flexible.

d. `flex: 2 1 0px;` 4. Items can shrink but not grow bigger.

5. Match the following **`flex-flow`** declarations with the resulting flexboxes (FIGURE 16-48).

a. `flex-flow: row wrap;`

b. `flex-flow: column nowrap;`

c. `flex-flow: row wrap-reverse;`

d. `flex-flow: column wrap-reverse;`

e. `flex-flow: row nowrap;`

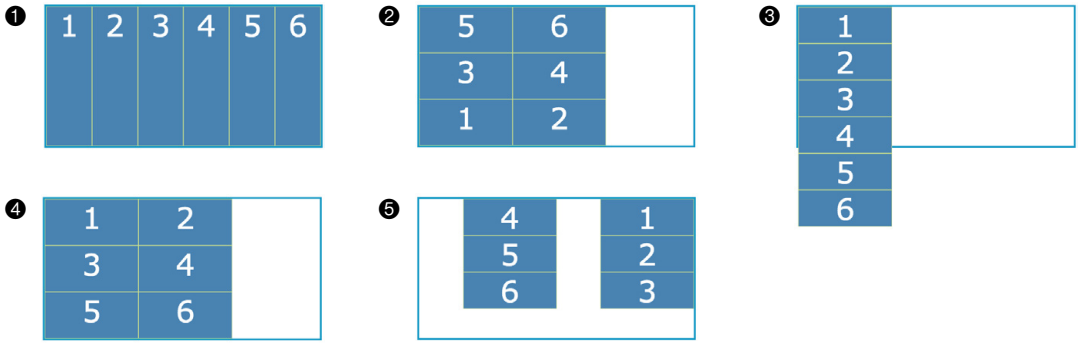


FIGURE 16-48. Various **`flex-flow`** settings.

6. Write style rules for displaying the flexbox items in the order shown in FIGURE 16-49.

Source HTML

```
<div id="container">
 <div class="box box1">1</div>
 <div class="box box2">2</div>
 <div class="box box3">3</div>
 <div class="box box4">4</div>
 <div class="box box5">5</div>
 <div class="box box6">6</div>
 <div class="box box7">7</div>
</div>
```

After reordering

FIGURE 16-49. Write styles to put items in the shown order.

Grid Layout

7. What is the key difference between Grid Layout and Flexbox? Name at least one similarity (there are multiple answers).

8. Create the grid template for the layout shown in FIGURE 16-50 by using **`grid-template-rows`** and **`grid-template-columns`**.

   Write it again, this time using the **`grid`** shorthand property.

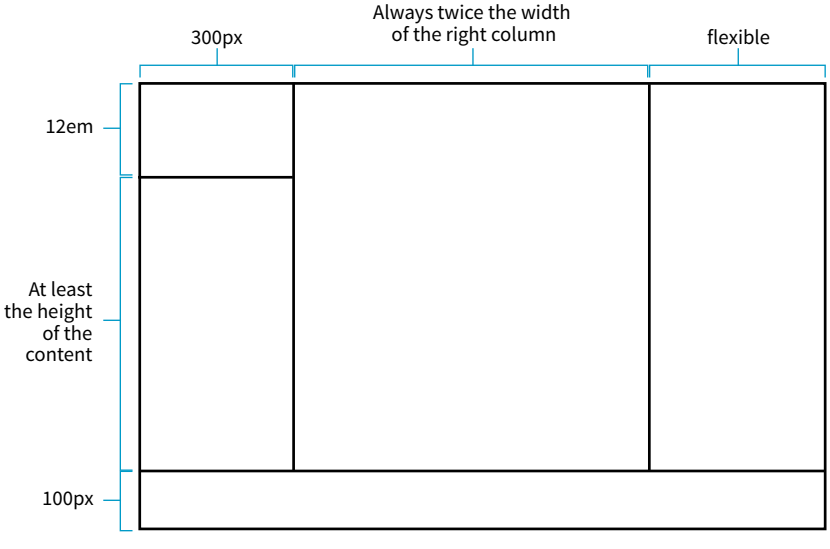


FIGURE 16-50. Create the grid template for this grid structure.

9. Match the following style declarations with the lettered grid items in FIGURE 16-51. In addition to automatic numbering, some of the grid lines have been named, as labeled.

a. _____

```
grid-row-start: 1;
grid-row-end: 3;
grid-column-start: 3;
grid-column-end: 7;
```

b. _____

```
grid-area: 2 / 2 / span 4 / 3;
```

c. _____

```
grid-area: bowie;
```

d. _____

```
grid-row: -2 / -1;
grid-column: -2 / -1;
```

e. _____

```
grid-row-start: george;
grid-row-end: ringo;
grid-column-start: paul;
grid-column-end: john;
```

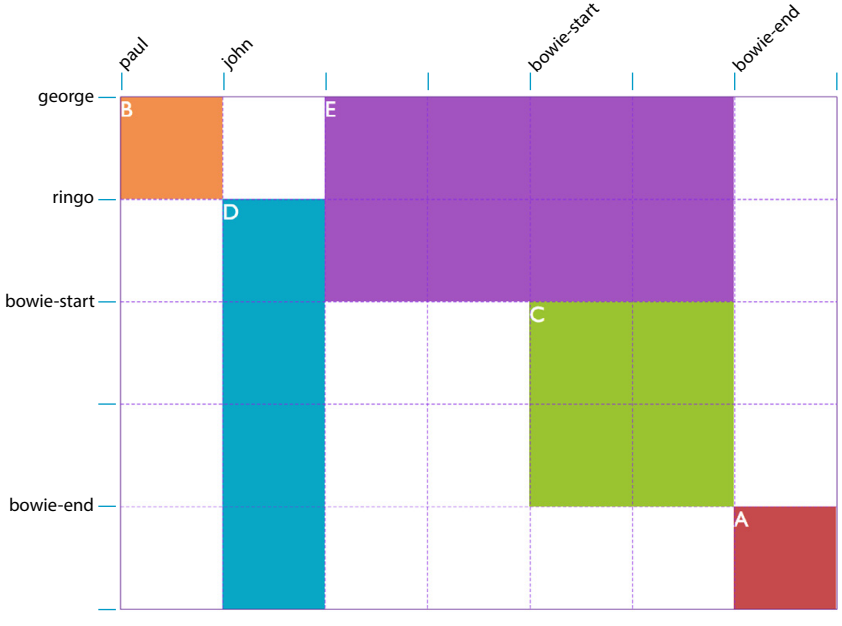


FIGURE 16-51. Match the style examples to the items in this grid.

10. Write a style rule that adds 1em space between columns in a grid container named `#gallery`.

11. Match the tasks with the declarations.

    a. `justify-self: end;`

    b. `align-items: end;`

    c. `align-content: center;`

    d. `align-self: stretch;`

    e. `justify-items: center;`

    _____ Make a particular item stretch to fill its container.

    _____ Position an image on the right edge of its grid area (in a left-to-right reading language).

    _____ Center the whole grid vertically in its container.

    _____ Push all of the images in a grid to the bottom of their respective cells.

    _____ Center all items in their areas horizontally.

CSS REVIEW: LAYOUT PROPERTIES

Here are the properties covered in this chapter, sorted into Flexbox and Grid sections and whether they apply to the container or item.

Flexbox Properties

Flex container properties	
`display: flex`	Turns on flexbox mode and makes the element a flex container
`flex-direction`	Indicates the direction in which items are placed in the flex container
`flex-wrap`	Specifies whether the flex items are forced onto a single line or wrapped onto multiple lines
`flex-flow`	Shorthand property for **`flex-direction`** and **`flex-wrap`**
`justify-content`	Specifies how space is distributed between and around items on the main axis
`align-content`	Aligns flex lines within the flex container when there is extra space on the cross axis
`align-items`	Specifies how the space is distributed around items on the cross axis

Flex item properties

`align-self`	Specifies how one item is aligned on the cross axis (overrides **`align-items`**)
`flex`	Shorthand property for **`flex-grow`**, **`flex-shrink`**, and **`flex-basis`**; specifies how items alter their dimensions to fit available space
`flex-basis`	Indicates the initial main size of a flex item
`flex-grow`	Specifies how much a flex item is permitted to grow when there is extra space in the container
`flex-shrink`	Specifies how much a flex item is permitted to shrink when there is not enough room in the container
`order`	Indicates the order used to lay out items in their container

Grid Properties

Grid container properties

`display: grid | inline-grid`	Sets the display mode of an element to a grid context
`grid-template`	Shorthand property for specifying **`grid-template-areas`**, **`grid-template-rows`**, and **`grid-template-columns`**
`grid-template-areas`	Assigns names to areas in the grid
`grid-template-columns`	Specifies track sizes for the columns in explicit grids
`grid-template-rows`	Specifies track sizes for the rows in explicit grids
`grid-auto-columns`	Specifies track sizes for automatically generated columns
`grid-auto-flow`	Indicates the direction and density in which items flow automatically into a grid
`grid-auto-rows`	Specifies track sizes for automatically generated rows
`grid`	Shorthand property for specifying **`grid-template-rows`**, **`grid-template-columns`**, and **`grid-template-areas`**; or **`grid-auto-flow`**, **`grid-auto-rows`**, and **`grid-auto-columns`**
`grid-gap`	Shorthand property for **`grid-row-gap`** and **`grid-column-gap`**
`grid-column-gap`	Specifies the width of the gutter between columns
`grid-row-gap`	Specifies the width of the gutter between rows
`justify-items`	Indicates alignment of all the grid items along the inline axis within their respective areas
`justify-content`	Indicates alignment of the grid tracks along the inline axis in its container

`align-items`	Indicates alignment of all the items in a grid along the block axis within their respective grid areas
`align-content`	Indicates alignment of the grid tracks along the block axis in the container

Grid item properties

`grid-column`	Shorthand property for specifying **`grid-column-start`** and **`grid-column-end`**
`grid-column-end`	Denotes the end line of the column in which an item is to be placed
`grid-column-start`	Denotes the start line of the column in which an item is to be placed
`grid-row`	Shorthand property for specifying **`grid-row-start`** and **`grid-row-end`**
`grid-row-end`	Denotes the end line of the row in which an item is to be placed
`grid-row-start`	Denotes the start line of the row in which an item is to be placed
`grid-area`	Assigns a grid item to a named area or an area described by its four boundary grid lines
`align-self`	Indicates alignment of a single item along the block axis within its grid area
`justify-self`	Indicates alignment of a single grid item along the inline axis within its area
`order`	Specifies the order in which to display the item relative to other items in the source
`z-index`	Specifies the stacking order of an item relative to other items when there is overlap

CHAPTER
17

RESPONSIVE WEB DESIGN

IN THIS CHAPTER

What RWD is and why it's important

Fluid layouts

Media queries

Design strategies and patterns

Testing options

I first introduced you to the concept of Responsive Web Design way back in **Chapter 3, Some Big Concepts You Need to Know**, and we've been addressing ways to keep all screen sizes in mind throughout this book. In this chapter, we get to do a deeper dive into responsive strategies and techniques.

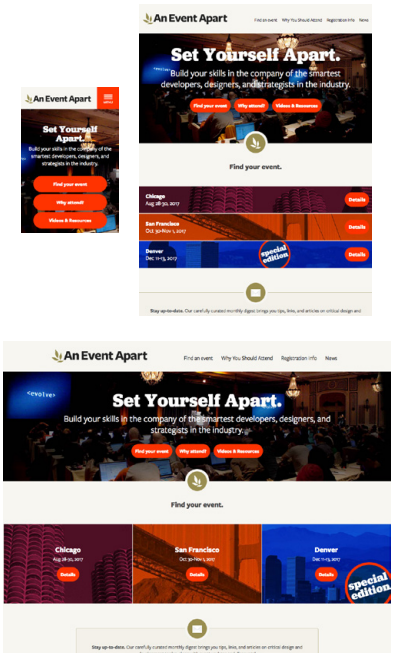
Just to recap, Responsive Web Design (or RWD) is a design and production approach that allows a website to be comfortably viewed and used on all manner of devices. The core principle is that all devices get the same HTML source, located at the same URL, but different styles are applied based on the viewport size to rearrange components and optimize usability. FIGURE 17-1 shows examples of responsive sites as they might appear on a smartphone, tablet, and desktop, but it is important to keep in mind that these sites are designed to work well on the continuum of every screen width in between.

WHY RWD?

Since the iPhone shook things up in 2007, folks now view the web on phones of all sizes, tablets, "phablets," touch-enabled laptops, wearables, televisions, video game consoles, refrigerators, and who knows what else that may be coming down the line.

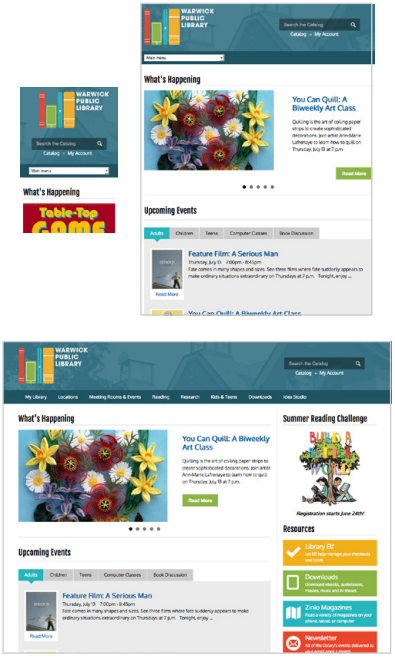
In 2016, mobile internet usage surpassed desktop usage—an important milestone. The percentage of web traffic that comes from devices other than desktop browsers is steadily increasing. For roughly 10% of Americans, a smartphone or tablet is their *only* access to the internet because of lack of access to a computer or high-speed WiFi at work or home.* Younger users may be mobile-only by choice. Furthermore, the vast majority of us access the

* Pew Research Center, "Smartphone Use in 2015," *www.pewinternet.org/2015/04/01/us-smartphone-use-in-2015/*.



An Event Apart
aneventapart.com

The Boston Globe
bostonglobe.com



Warwick Public Library
warwicklibrary.org

FIGURE 17–1. Examples of responsive sites that adapt to fit small, medium, and large screens and all sizes in between.

NOTE

Each site in FIGURE 17–1 *has one morphing design, not three distinct layouts. Some sites do have a limited number of layouts aimed at specific devices, which is an approach known as* Adaptive Design.

web from a number of platforms (phone, tablet, computer) over the course of the day. And guess what—we expect to have a similar experience using your content or service regardless of how we access your site.

That's where RWD fits in. With one source, you ensure that mobile visitors receive the same content as other visitors (although it might be organized differently). Users are not penalized with stripped-down content or features just because they are using a smartphone. And for visitors who might start using your site on one device and finish it on another, you can ensure a consistent experience.

Responsive design is becoming the default way to build a website that meets the demands of our current multidevice environment.

In fact, for many web designers, "responsive design" is now just "web design." Instead of a niche approach, it is becoming the default way to build a website that meets the demands of our current multidevice environment.

THE RESPONSIVE RECIPE

The deluge of web-enabled mobile devices initially sent shockwaves through the web design community. Accustomed to designing exclusively for large desktop screens, we were unclear about how we could accommodate screens that fit in the palm of your hand.

One solution was to rely on the phone's built-in web display functionality. By default, mobile devices display an entire web page shrunken down to fit on whatever screen real estate is available. Users can pinch to zoom into the details and scroll around to various parts of the page. While that technically works, it is far from an optimal experience. Another approach was to create a separate mobile site just for small screens and people "on the go." There are still many companies and services that use dedicated mobile ("m-dot") sites—Twitter and Facebook come to mind—but in general, m-dot sites are going away in favor of RWD. Google is helping the process along by favoring responsive sites with single URLs over *m.* or *mobile.* versions.

NOTE

Mobile sites were discussed in the sidebar **"M-dot Sites"** *in* **Chapter 3**.

In 2010, Ethan Marcotte gave name to another, more flexible solution in his article "Responsive Web Design" (*alistapart.com/article/responsive-web-design*), which has since become a cornerstone of modern web design. In this chapter, I will follow the "ingredients" for RWD that Ethan outlines in his book *Responsive Web Design* (A Book Apart).

The technique has three core components:

A flexible grid

Rather than remaining at a static width, responsive sites use methods that allow them to squeeze and flow into the available browser space.

Flexible images

Images and other embedded media need to be able to scale to fit their containing elements.

CSS media queries

Media queries give us a way to deliver sets of rules only to devices that meet certain criteria, such as width and orientation.

To this list of ingredients, I would add the viewport `meta` element, which makes the width of the web page match the width of the screen. That's where we'll begin our tour of the mechanics of RWD.

Setting the Viewport

To fit standard websites onto small screens, mobile browsers render the page on a canvas called the viewport and then shrink that viewport down to fit the width of the screen (device width). For example, on iPhones, mobile Safari sets the viewport width to 980 points (see **Note**), so a web page is rendered as though it were on a desktop browser window set to 980 pixels wide. That rendering gets shrunk down to the width of the screen (ranging from 320 to 414 points, depending on the iPhone model), cramming a lot of information into a tiny space.

NOTE

iOS layouts are measured in points, a unit of measurement that is independent from the number of pixels that make up the physical screen. Points and device pixels are discussed in more detail in **Chapter 23, Web Image Basics**.

Mobile Safari introduced the viewport `meta` element, which allows us to define the size of that initial viewport. Soon, the other mobile browsers followed suit. The following `meta` element, which goes in the `head` of the HTML

document, tells the browser to set the width of the viewport equal to the width of the device screen (**width=device-width**), whatever that happens to be (FIGURE 17-2). The **initial-scale** value sets the zoom level to 1 (100%).

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

With the viewport **meta** element in place, if the device's screen is 320 pixels wide, the rendering viewport on that device will also be 320 pixels across (not 980) and will appear on the screen at full size. That is the width we test for with media queries, so setting the viewport is a crucial first step.

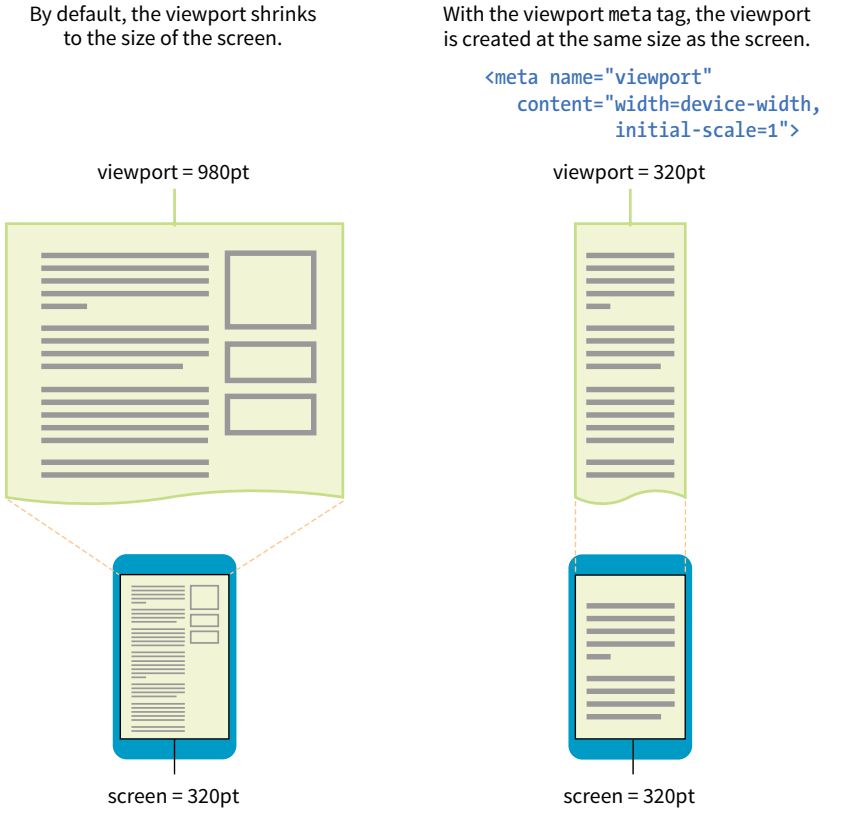


FIGURE 17-2. The viewport **meta** element matches the resolution of the device's browser viewport to the resolution of its screen.

Flexible Grids (Fluid Layouts)

In the Flexbox and Grid discussions in the previous chapter, we saw examples of items expanding and contracting to fill the available space of their containers. That fluidity is exactly the sort of behavior you need to make content neatly fit a wide range of viewport sizes. Fluid layouts (or "flexible grids," as Ethan Marcotte calls them in his article and book) are the foundation of responsive design.

FUN FACT

Media queries are always at work, even after the page has initially loaded. If the viewport changes—for example, if a user turns a phone from portrait to landscape orientation or resizes a desktop browser window—the query runs again and applies the styles appropriate for the new width.

WARNING

The viewport **meta** *element also allows the* **maximum-scale** *attribute. Setting it to 1 (***maximum-scale=1***) prevents users from zooming the page, but it is strongly recommended that you avoid doing so because resizing is important for accessibility and usability.*

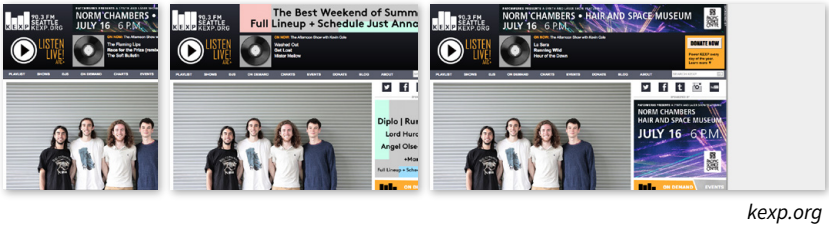
In a fluid layout, the page area and its grid resize proportionally to fill the available width of the screen or window (FIGURE 17-3, top). That is easily accomplished with `fr` and `minmax()` units in CSS Grid layouts and with `flex` property settings in Flexbox. If you need to also target older browsers that don't support CSS layout standards, you can use percentage values for horizontal measurements so elements remain proportional at varying sizes (see the sidebar **"Converting Pixels to Percentages"**).

In the past, when we knew that everyone was looking at our sites on desktop monitors, fixed-width layouts were the norm. (Ahh, those simple pre-mobile days when we only needed to deal with radically incompatible browser support!) As the name implies, fixed-width layouts are created at a specific pixel width (FIGURE 17-3, bottom), with 960 pixels being quite fashionable (see **Note**). Specifying all measurements in pixel values gave designers control over the layout as they might have in print, and ensured that users across all platforms and browsers got similar, if not the same, rendering of the page.

Fluid layouts fill the viewport proportionally.

w3c.org

Fixed layouts stay the same size and may get cut off or leave extra space.



kexp.org

FIGURE 17-3. Fluid and fixed layout examples.

It didn't take long to realize, however, that it would be impossible to create separate fixed-width designs tailored to every device size. Clearly, fluidity has the advantage. It is based on the intrinsic nature of the normal flow, so we're working *with* the medium here rather than against it. When the layout reflows to fill the available width, you don't need to worry about horizontal scrollbars or awkward empty space in the browser.

On the downside, fluid layouts may allow text line lengths to become uncomfortably long, so that is something to watch out for. We'll go into more detail on layouts later in this chapter.

Converting Pixels to Percentages

To convert measurements in your layout from pixels to percentages, use the following formula:

```
target ÷ context = result
```

The target is the size of the element you are resizing. The context is the size of the containing element. The result is a percentage you can use in in your style rules. Don't worry about rounding long decimal strings down. Browsers know what to do with them, and the extra precision doesn't hurt.

NOTE

Designers landed on 960 pixels wide as a standard page width because it filled the standard desktop width at the time (1,028 pixels) and it was easily divided into an equal columns. Page layout systems based on 12-column grids within the 960-pixel page were also popular.

PERSONAL ANECDOTE

When I got started in web design in 1993, the most common PC monitor size was a measly 640 × 480 pixels, unless you were a fancy-pants designer type with a 800 × 600 screen. My earliest designs had a fixed width of an adorable 515 pixels.

Making Images Flexible

Every now and then a solution is simple. Take, for example, the style rule required to make images scale down to fit the size of their container:

```
img {
  max-width: 100%;
}
```

That's it! When the layout gets smaller, the images in it scale down to fit the width of their respective containers. If the container is larger than the image—for example, in the tablet or desktop layouts—the image does not scale larger; it stops at 100% of its original size (FIGURE 17-4). When you apply the **max-width** property, you can omit the **width** and **height** attributes in the **img** elements in the HTML document. If you do set the **width** attribute, be sure the **height** attribute is set to **auto**; otherwise, the image won't scale proportionately.

```
img { max-width: 100%, }
```

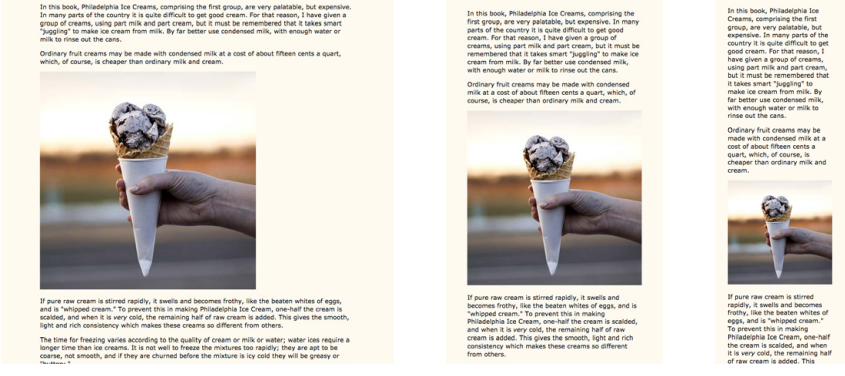


FIGURE 17-4. Setting the **max-width** of inline images allows them to shrink to fit available space but not grow larger than actual size.

Responsive images

But wait—things are never that simple, right? If you think back to our discussion of responsive images in **Chapter 7, Adding Images**, you'll remember that there is some elbow grease required to avoid serving unnecessarily large images to small devices as well as making sure large, high-density monitors get high-resolution images that shine. Choosing the best image size for performance is part of the responsive design process, but we won't be concentrating on that in this chapter. We've got bigger fish to fry!

Other embedded media

Videos and other embedded media (using **object** or **embed** elements) also need to scale down in a responsive environment. Unfortunately, videos do not retain their intrinsic ratios when the width is scaled down, so there are a

few more hoops to jump through to get good results. Thierry Koblentz documents one strategy nicely in his article "Creating Intrinsic Ratios for Video" at *www.alistapart.com/articles/creating-intrinsic-ratios-for-video*. There is also a JavaScript plug-in called FitVids.js (created by Chris Coyier and the folks at Paravel) that automates Koblentz's technique for fluid-width videos. It is available at *fitvidsjs.com*.

Media Query Magic

Now we get to the real meat of responsive design—media queries! Media queries apply different styles based on characteristics of the browser: its width, whether it is vertically or horizontally oriented, its resolution, and more. They are what make it possible to send a one-column layout to small screens and a multicolumn layout to larger screens on the fly.

The query itself includes a media type followed by a particular feature and a value for which to test. The criteria are followed by a set of curly brackets that contain styles to apply if the test is passed. The structure of a media query as used within a style sheet goes like this:

```
@media type and (feature: value) {
  /* styles for browsers that meet this criteria */
}
```

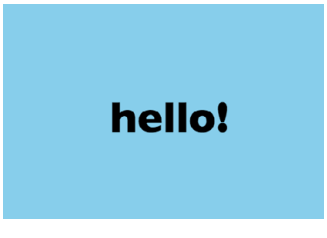
Let's clarify that with an example. The following media queries look at whether the viewport is on a screen and in **`landscape`** (horizontal) or **`portrait`** (vertical) orientation. When the query detects that the viewport is in landscape mode, the background color of the page is "skyblue"; when it is in portrait orientation, the background is "coral" (FIGURE 17-5). If this were displayed on a smartphone that tips from vertical to horizontal and back again, the colors would change as it tilts. This isn't a very practical design choice, but it does provide a very simple illustration of media queries at work.

```
@media screen and (orientation: landscape) {
  body {
     background: skyblue;
     }
}
@media screen and (orientation: portrait) {
  body {
    background: coral;
    }
}
```

HEADS UP

Having style declaration curly brackets nested inside media query curly brackets can get a little confusing. Be sure that you have the right number of curly brackets and nest them properly. Careful indenting is helpful. Many code-editing programs also use color coding to help you keep them straight.

When the viewport is in portrait mode, the background color is "coral."



When the viewport is in landscape mode, the background color is "skyblue."

FIGURE 17-5. Changing the background color based on the orientation of the viewport with media queries.

Media types

Media types, as included in the first part of a query, were introduced in CSS2 as a way to target styles to particular media. For example, this **@media** rule delivers a set of styles only when the document is printed (it does not test for any specific features or values):

```
@media print {
  /* print-specific styles here */
}
```

The most current defined media types are **all**, **print**, **screen**, and **speech** (see **Note**). If you are designing for screen, the media type is optional, so you can omit it as shown in the syntax example shown here, but including it doesn't hurt. I'll be including the screen **media** type for the sake of clarity in my examples.

```
@media (feature: value) {
  }
```

NOTE

CSS2 also defined **aural**, **handheld**, **braille**, **embossed**, **projection**, **tty**, *and* **tv**, *but they have been deprecated in the latest Media Queries Level 4 spec (currently a Working Draft) and are discouraged from use.*

Media feature queries

CSS3 media queries take targeting one step further by letting us test for a particular *feature* of a viewport or device. We saw an example of testing the orientation of a device in FIGURE 17-5. The most common feature to test for is the viewport width. You can also test for a minimum width (**min-width**) and maximum width (**max-width**).

Here is a simple example that displays headline fonts in a fancy cursive font only when the viewport is 40em or wider—that is, when there is enough space for the font to be legible. Viewports that do not match the query (because they are narrower than 40em) use a simple serif face.

```
h1 {
  font-family: Georgia, serif;
  }
@media screen and (min-width: 40em) {
  h1 {
    font-family: 'Lobster', cursive;
    }
}
```

The complete list of device features you can detect with media queries appears in TABLE 17-1.

■ AUTHORING TIP

Minimum-width queries are your go-to for creating mobile-first responsive design.

TABLE 17-1. Media features you can evaluate with media queries

Feature	Description
`width`	The width of the display area (viewport). Also **`min-width`** and **`max-width`**.
`height`	The height of the display area (viewport). Also **`min-height`** and **`max-height`**.
`orientation`	Whether the device is in **`portrait`** or **`landscape`** orientation.
`aspect-ratio`	Ratio of the viewport's width divided by height (width/height). Example: **`aspect-ratio: 16/9`**.
`color`	The bit depth of the display; for example, **`color: 8`** tests for whether the device has at least 8-bit color.
`color-index`	The number of colors in the color lookup table.
`monochrome`	The number of bits per pixel in a monochrome device.
`resolution`	The density of pixels in the device. This is increasingly relevant for detecting high-resolution displays.
`scan`	Whether a **`tv`** media type uses progressive or interlace scanning. (Does not accept **`min-`**/**`max-`** prefixes.)
`grid`	Whether the device uses a grid-based display, such as a terminal window. (Does not accept **`min-`**/**`max-`** prefixes.)

Deprecated features

The following features have been deprecated in Media Queries Level 4 Working Draft and are discouraged from use.

`device-width`	The width of the device's rendering surface (the whole screen). (Deprecated in favor of **`width`**.)
`device-height`	The height of the device's rendering surface (the whole screen). (Deprecated in favor of **`height`**.)
`device-aspect-ratio`	Ratio of the whole screen's (rendering surface) width to height. (Deprecated in favor of **`aspect-ratio`**.)

New in Media Queries Level 4

These features have been added in the Working Draft of MQ4. Some may gain browser support, and some may be dropped from future drafts. I include them here to show you where the W3C sees media queries going. For details, see *drafts.csswg.org/mediaqueries-4*.

`update-frequency`	How quickly (if at all) the output device modifies the appearance of the content.
`overflow-block`	How the device handles content that overflows the viewport along the block axis.
`overflow-inline`	Whether the content that overflows the viewport along the inline axis can be scrolled.
`color-gamut`	The approximate range of colors that are supported by the user agent and output device.
`pointer`	Whether the primary input mechanism is a pointing device and how accurate it is.
`hover`	Whether the input mechanism allows the user to hover over elements.
`any-pointer`	Whether any available input mechanism is a pointing device, and how accurate it is.
`any-hover`	Whether any available input mechanism allows hovering.

How to use media queries

You can use media queries within a style sheet or to conditionally load external style sheets. Media queries may not be used with inline styles.

Within a style sheet

The most common way to utilize media queries is to use an **`@media`** ("at-media") rule right in the style sheet. The examples in this chapter so far are all **`@media`** rules.

When you use media queries within a style sheet, the order of rules is very important. Because rules later in the style sheet override the rules that come before them, your media query needs to come *after* any rules with the same declaration.

The strategy is to specify the baseline styles that serve as a default, and then override specific rules as needed to optimize for alternate viewing environments. In RWD, the best practice is to set up styles for small screens and browsers that don't support media queries, and then introduce styles for increasingly larger screens later in the style sheet.

That's exactly what I did in the headline font-switching example earlier. The **`h1`** sets a baseline experience with a local serif font, and then gets enhanced for larger screens with a media query.

With external style sheets

For large or complicated sites, developers may choose to put styles for different devices into separate style sheets and call in the whole *.css* file when certain conditions are met. One method is to use the **`media`** attribute in the **`link`** element to conditionally load separate *.css* files. In this example, the basic styles for a site are requested first, followed by a style sheet that will be used only if the device is more than 1,024 pixels wide (and if the browser supports media queries):

```
<head>
  <link rel="stylesheet" href="styles.css">
  <link rel="stylesheet" href="2column-styles.css" media="screen and
(min-width:1024px)">
</head>
```

Some developers find this method helpful for managing modular style sheets, but it comes with the disadvantage of requiring extra HTTP requests for each additional *.css* file. Be sure to provide only as many links as necessary (perhaps one for each major breakpoint), and rely on **`@media`** rules within style sheets to make minor adjustments for sizes in between.

Similarly, you can carry out media queries with **`@import`** rules that pull in external style sheets from within a style sheet. Notice that the word "media" does not appear in this syntax, only the type and query.

```
<style>
  @import url("/default-styles.css");
  @import url("/wide-styles.css") screen and (min-width: 1024px);
  /* other styles */
</style>
```

Browser support

We can't close out a discussion of media queries without a nod to browser support. The good news is that media queries are supported by virtually all desktop and mobile browsers in use today. The big exceptions are Internet Explorer versions 8 and earlier, which have no support. Because of the staying power of the Windows XP operating system, IE8 continues to show up in browser use statistics (at 1–2% as I write, ahead of IE 9 and 10). If your site has hundreds of thousands of users, that 1% ends up being a significant number of broken experiences.

If you expect to have visitors using old versions of IE, you have a couple of options. First, you could use the Respond.js polyfill, which adds support for **`min-width`** and **`max-width`** to non-supporting browsers. It was created by Scott Jehl and is available at *github.com/scottjehl/Respond*.

The other option is to create a separate style sheet with a no-frills desktop layout and deliver it only to users with IE8 or earlier by using a conditional comment. Other browsers ignore the content of this IE-specific comment:

```
<!-- [if lte IE 8]>
    <link rel="stylesheet" href="/path/IE_fallback.css">
<![endif]-->
```

Depending on your site statistics and when you are reading this, you may not need to worry about media query support at all. Lucky you!

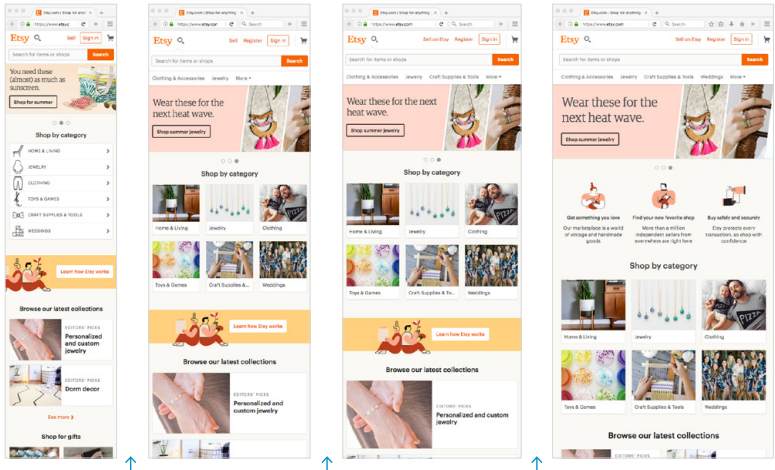
CHOOSING BREAKPOINTS

A breakpoint is a width at which you introduce a style change.

A breakpoint is the point at which we use a media query to introduce a style change. When you specify **`min-width: 800px`** in a media query, you are saying that 800 pixels is the "breakpoint" at which those particular styles should be used. FIGURE 17-6 shows some of the breakpoints at which Etsy.com makes both major layout changes and subtle design tweaks on its home page.

Choosing breakpoints can be challenging, but there are a few best practices to keep in mind.

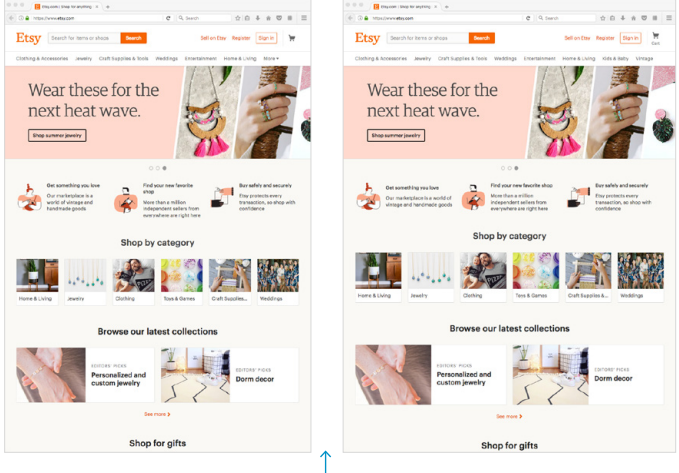
When RWD was first introduced, there were only a handful of devices to worry about, so we tended to base our breakpoints on the common device sizes (320 pixels for smartphones, 768 pixels for iPads, and so on), and we created a separate design for each breakpoint. It didn't take long until we had to deal with device widths at nearly every point from 240 to 3,000+ pixels. That device-based approach definitely didn't scale.



At the **480-pixel breakpoint**, the category navigation changes from a list to photos. "Register" is added to the top navigation bar.

At **501 pixels**, "Sell" becomes "Sell on Etsy" (a very subtle adjustment). You can also see more links in the navigation bar under the search field.

At **640 pixels**, the "How Etsy Works" images and messages move above the categories. In smaller views, they were accessible via the "Learn how Etsy works" link in a yellow bar.



At **901 pixels**, the search input form moves into the top header.

At **981 pixels**, the word "Cart" appears under the shopping cart icon. We now see the full list of navigation options in the header (no "More" link). At this point, the layout expands to fill larger windows until it reaches its maximum width of 1400 pixels. Then margins add space equally on the left and right to keep the layout centered.

FIGURE 17-6. A series of breakpoints used by Etsy's responsive site (2017).

Module-Based Breakpoints

A better approach is to create breakpoints for the individual parts of a page rather than switching out the entire page at once (although for some pages that may work just fine). A common practice is to create the design for narrow screens first, and then resize the browser wider and pay attention to the point at which each part of the page starts to become unacceptable. The navigation might become too awkward and need a breakpoint at 400 pixels wide, but the one-column layout might be OK until it reaches 800 pixels, at which point a two-column design could be introduced.

It is common to create breakpoints for each component of the page rather than changing the entire page at once.

In his book *Responsive Design: Patterns & Principles* (A Book Apart), Ethan Marcotte calls this "content out" design and puts it like this:

> *For me, that "content out" process begins by looking at the smallest version of a piece of content, then expanding that element until its seams begin to show and it starts to lose its shape. Once that happens, that's an opportunity to make a change—to introduce a breakpoint that reshapes the element and preserves its integrity.*

If you find that you have a lot of breakpoints within a few pixels or ems of one another, grouping them together may streamline your style sheet and process. And it doesn't hurt to keep the screen sizes of the most popular devices in mind in case nudging your breakpoint down a little helps improve the experience for a whole class of users. The site Screen Sizes (*screensiz.es*) lists the dimensions of a wide range of popular devices. A web search will turn up similar resources.

Em-Based Breakpoints

The examples in this section have been based on breakpoints with pixel measurements. An alternative, and many would say better, method is to use ems instead of pixels in the media query. Remember that an em is equal to the current font size of an element. When used in a media query, an em is based on the base font size for the document (16 pixels by default, although that can be changed by the user or the page author).

Em-based media queries keep the layout proportional to the font size.

Pixel-based media queries don't adapt if the user changes their font size settings, which people do in order to be able to read the page more easily. But em-based media queries respond to the size of the text, keeping the layout of the page in proportion.

For example, say you have a layout that switches to two columns when the page reaches 800 pixels. You've designed it so the main column has an optimum text line length when the base font size is the default 16 pixels. If the user changes their base font size to 32 pixels, that double-sized text will pour into a space intended for text half its size. Line lengths would be awkward.

Using em-based queries, if the query targets browsers wider than 50em, when the base font size is 16 pixels, the switch happens at 800 pixels (as designed).

However, should the base font size change to 32 pixels, the two-column layout would kick in at 1,600 pixels (50em × 32px = 1,600px), when there is plenty of room for the text to fill the main column with the same line lengths as the original design.

This example used a whole-page layout switch, but media queries on individual components (as discussed earlier) can use ems as well. In the next section, I'll introduce some of the aspects of web pages that require attention when you're choosing breakpoints.

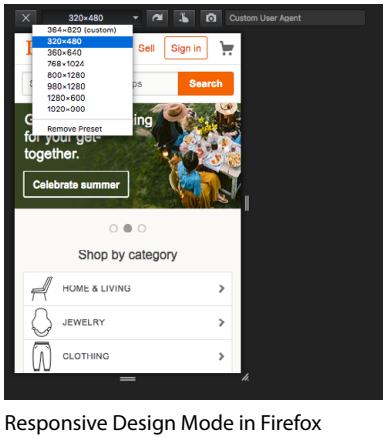
How Wide Is the Viewport?

I've suggested making your browser wider until you see that a breakpoint is needed, but how do you know how wide the window is? There are a number of tools that provide window measurements.

Firefox, Chrome, and Safari all have tools that can show you how a page looks at specific viewport dimensions. In Responsive Mode (or View), you get a resizable window-in-a-window that can be set to standard device sizes or resized manually. The pixel dimensions are displayed as the viewport resizes.

In Firefox, access Responsive Design Mode (FIGURE 17-7) via the Web Developer Tools (**Tools → Web Developer → Responsive Design Mode**). Safari's Responsive Design Mode is accessible via **Develop → Enter Responsive Design Mode**. Chrome offers a Device Toolbar (**View → Developer → Developer Tools**, and then look for the Toggle Device Toolbar icon on the left of the menu bar). They all work about the same, but you may find you prefer one browser's user interface over another.

To find out how your browser window or device responds to media queries, go to MQTest.io (*mqtest.io*) by Viljami Salminen. In addition to viewport width and height, it reports on other device features such as device-pixel-ratio, aspect ratio, and more.



Responsive Design Mode in Firefox shows you the exact pixel dimensions of its viewport. It has shortcuts to resize it to common device dimensions. Chrome and Safari have similar responsive views.

MQTest.io is a web page that reports on how your browser responds to media queries, including width.

FIGURE 17-7. Checking viewport size in Firefox's Responsive Design Mode and MQTest.io.

DESIGNING RESPONSIVELY

We've covered the RWD nuts and bolts—now let's talk about some of the decisions designers make when creating responsive sites. I am only going to be able to scratch the surface here, but you'll find much deeper explorations on these themes in the books and articles listed along the way. For now, I just want to raise your awareness of responsive strategies. At the end of the section, you will use some of these strategies to make the Black Goose Bakery page responsive.

We've seen a few examples in our exercises of content looking wonky when the browser gets very narrow or very wide. A three-column layout just doesn't fit, and text in an image may become unreadable when it is scaled down to fit a 320-pixel-wide screen. At the other end of the spectrum, the line length in single-column layouts becomes too long to read comfortably when the viewport fills a high-resolution desktop monitor. For many aspects of a web page, one size does not fit all. As designers, we need to pay attention to where things fall apart and set breakpoints to "preserve the integrity" of the elements (as Ethan Marcotte so nicely puts it).

In the broadest of strokes, the tricky bits to keep optimized over a wide range of viewport sizes include the following:

- Content hierarchy
- Layout
- Typography
- Navigation
- Images
- Special content such as tables, forms, and interactive features

Content Hierarchy

Content is king on the web, so it is critical that content is carefully considered and organized before any code gets written. These are tasks for Information Architects and Content Strategists who address the challenges of organizing, labeling, planning, and managing web content.

Organization and hierarchy across various views of the site are a primary concern, with a particular focus on the small-screen experience. It is best to start with an inventory of potential content and pare it down to what is most useful and important for all browsing experiences. Once you know what the content modules are, you can begin deciding in what order they appear on various screen sizes.

Keep in mind that you should strive for content parity—that is, the notion that the same content is accessible regardless of the device used to access the

site. It might be that visitors need to follow a slightly different navigational path to that information, but dropping portions of your site on small screens because you think mobile users won't need it is false thinking. People do bop between devices mid-task, and you want to be sure they have everything they need.

This is a woefully brief introduction to what is perhaps the most important first step of creating a site, but it is just outside the focus of this book. To get up to speed properly with content strategy, particularly as it applies to RWD, I recommend the following books:

- *Content Strategy for the Web, 2nd Edition,* by Kristina Halvorson and Melissa Rich (New Riders)
- *Content Strategy for Mobile* by Karen McGrane (A Book Apart)

Layout

Rearranging content into different layouts may be the first thing you think of when you picture responsive design, and with good reason. The layout helps form our first impression of a site's content and usability.

As mentioned earlier, responsive design is based on fluid layouts that expand and contract to fill the available space in the viewport. One fluid layout is usually not enough, however, to serve all screen sizes. More often, two or three layouts are produced to meet requirements across devices, with small adjustments between layout shifts.

Designers typically start with a one-column layout that fits well on small handheld devices and rearrange elements into columns as more space is available. They may also have the design for the widest screens worked out early on so there is an end-point in mind. The design process may involve a certain amount of switching between views and making decisions about what happens along the way.

Layout and line length

A good trigger for deciding when to adjust the layout is to look at text line lengths. Lines of text that are too stubby or too long are difficult to read, so you should aim for optimal line lengths of 45 to 75 characters, including spaces. If your text lines are significantly longer, it's time to make changes to the layout such as increasing the margins or introducing an additional column. You might also increase the font size of the text to keep the character count in the desired range.

Clarissa Peterson introduces a neat trick for testing line lengths in her book *Learning Responsive Web Design* (O'Reilly). Put a **span** around the 45th to 75th characters in the text and give it a background color (FIGURE 17-8). That way, you can easily check whether the line breaks are happening in the safe zone

Conditional Loading

Content parity doesn't mean that all of the content that fits on a large screen should be stuffed onto the small-screen layout. All that scrolling and extra data to download isn't doing mobile users any favors.

A better approach is to use conditional loading, in which small-screen users get the most important content with links to access supplemental content (comments, product details, ads, lists of links, etc.) when they want it. The information is available to them, just not all at once. Meanwhile, on larger screens, those supplemental pieces of content get displayed in sidebars automatically.

Conditional loading requires JavaScript to implement, so I won't be giving you the specific how-tos here, but it is good to know that there are alternatives to cramming every little thing onto every device.

at a glance. Of course, this line length hint would be removed before the site is made public.

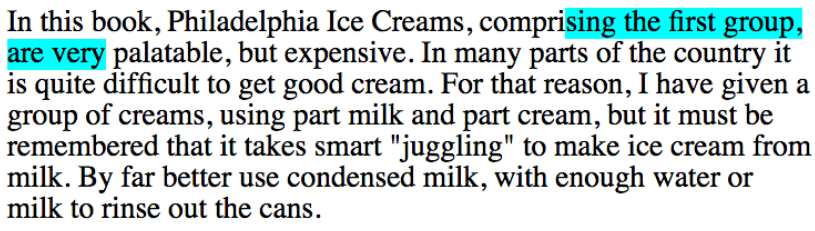


FIGURE 17-8. Highlight the 45th to 75th characters to test for optimal line lengths at a glance.

Responsive layout patterns

The manner in which a site transitions from a small-screen layout to a wide-screen layout must make sense for that particular site, but there are a few patterns (common and repeated approaches) that have emerged over the years. We can thank Luke Wroblewski (known for his "Mobile First" approach to web design, which has become the standard) for doing a survey of how responsive sites handle layout. The article detailing his findings, "Multi-Device Layout Patterns" (*www.lukew.com/ff/entry.asp?1514*), is getting on in years, but the patterns persist today. Following are the top patterns Luke named in his article (FIGURE 17-9):

Mostly fluid

This pattern uses a single-column layout for small screens, and another fluid layout that covers medium and large screens, with a maximum width set to prevent it from becoming too wide. It generally requires less work than other solutions.

Column drop

This solution shifts between one-, two-, and three-column layouts based on available space. When there isn't room for extra columns, the sidebar columns drop below the other columns until everything is stacked vertically in the one-column view.

Layout shifter

If you want to get really fancy, you can completely reinvent the layout for a variety of screen sizes. Although expressive and potentially cool, it is not necessary. In general, you can solve the problem of fitting your content to multiple environments without going overboard.

Tiny tweaks

Some sites use a single-column layout and make tweaks to type, spacing, and images to make it work across a range of device sizes.

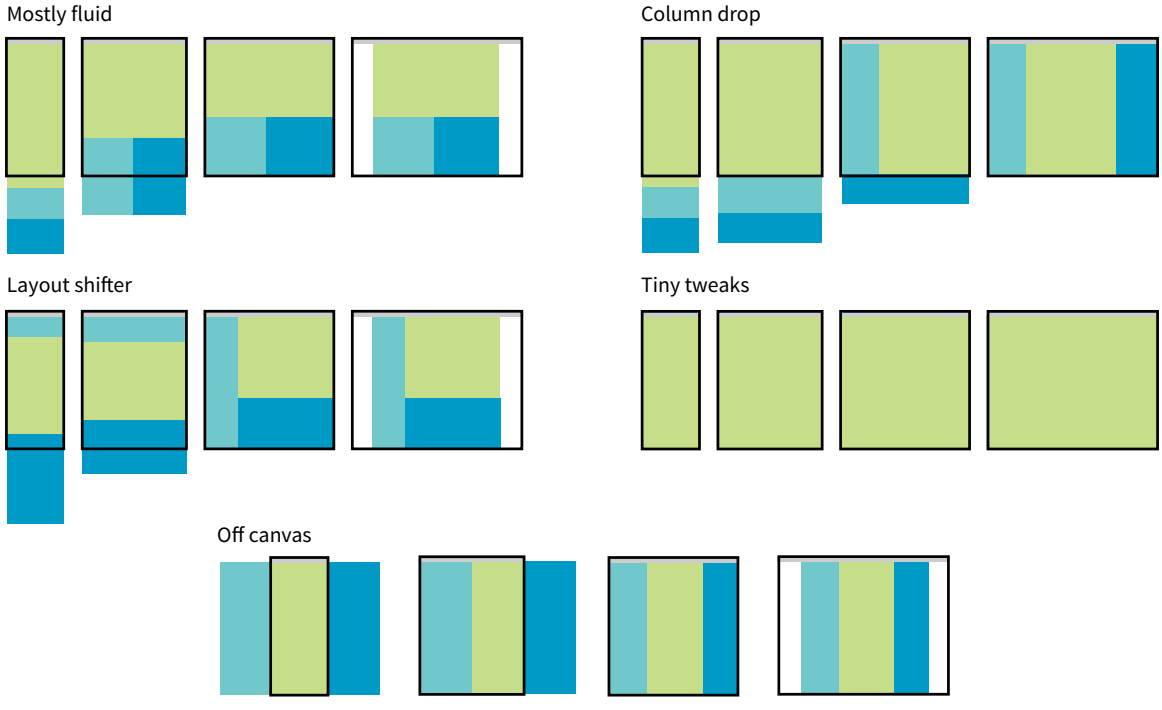


FIGURE 17-9. Examples of the responsive layout patterns identified by Luke Wroblewski.

Off canvas

As an alternative to stacking content vertically on small screens, you may choose to use an "off-canvas" solution. In this pattern, a page component is located just out of sight on the left or right of the screen and flies into view when requested. A bit of the main content screen remains visible on the edge to orient users as to the relationship of moving parts. This was made popular by Facebook, wherein Favorites and Settings were placed on a panel that slid in from the left when users clicked a menu icon.

You can see working examples of these and additional layout patterns on the "Responsive Patterns" page assembled by Brad Frost (*bradfrost.github.io/this-is-responsive/patterns.html*).

Typography

Typography requires fine-tuning along the spectrum from small-screen to wide-screen views in order to keep it legible and pleasant to read. Here are a few typography-related pointers (FIGURE 17-10):

Font face

Be careful about using fancy fonts on small screens and be sure to test for legibility. At small sizes, some fonts become difficult to read because

FIGURE 17-10. General typography guidelines for small and large screens.

line strokes become too light or extra flourishes become little blobs. Consider also that small screens may be connecting over cellular, so taking advantage of locally available fonts may be better for performance than requiring a web font to download. If a strict brand identity requires font consistency on all devices, be sure to choose a font face that works well at all sizes. If that is not a concern, consider using a web font only on larger screens. We strive to serve the same design to all devices, but as with everything else in web design, flexibility is important.

Font size

Varying viewport widths can wreak havoc on line lengths. You may find that you need to increase the font size of text elements for wider viewports to maintain a line length of between 45 and 75 characters. It also makes it easier to read from the distance users typically sit from their large screens. Conversely, you could use em-based media queries so that the layout stays proportional to the font size. With em-based queries, line lengths stay consistent.

Line height

Line height is another measurement that you may want to tweak as screens get larger. On average, line height should be about 1.5 (using a number value for the `line-height` property); however, slightly tighter line spacing (1.2 to 1.5) is easier to read with the shorter line lengths on small screens. Large screens, where the type is also likely to be larger, can handle more open line heights (1.4 to 1.6).

Margins

On small screens, make the most of the available space by keeping left and right margins on the main column to a minimum (2–4%). As screens get larger, you will likely need to increase side margins to keep the line

Variable Fonts

In late 2016, OpenType released a new font technology called OpenType Font Variations, known less formally as "variable fonts." You can change the weight, width, style (italic), slant, and optical size of a variable font by using `font-*` style properties. The marvel of this technology is that you can deliver one font file (that's just one call to the server) and stretch and manipulate it to suit many purposes, such as to make it narrower to preserve height and line length on small screens. Browser support for variable fonts is due to start kicking in in 2018. For more information, see the article "Get Started with Variable Fonts" by Richard Rutter at *medium.com/@clagnut/get-started-with-variable-fonts-c055fd73ecd7*.

The *axis-praxis.org* site allows you to play around with variable fonts using sliders to adjust the weight and other qualities. Note that you need a browser that supports variable fonts for it to work.

lengths under control and just to add some welcome whitespace to the layout. Remember to specify margins above and below text elements in em units so they stay proportional to the type.

Fluid Typography with Viewport Units

To make the size of text proportional to the size of the viewport, use the viewport-percentage lengths, **`vw`** and **`vh`**, for **`font-size`**. A **`vw`** (viewport width) unit is equal to 1% of the width of the viewport (or the "initial containing block," as it is called in the specification). One **`vh`** is 1% of the viewport height. The spec also defines a **`vmin`** unit (the smaller of **`vw`** or **`vh`**) and **`vmax`** (the larger of **`vw`** or **`vh`**), but they are not as well supported.

Browser support is pretty good with the exception of IE9 and earlier and support for **`vmin`** and **`vmax`**. There are quite a few known bugs, so be sure to check the Known Issues tab on the *CanIUse.com* page for these values.

For an exploration of using viewport units for text, read "Responsive Font Size and Fluid Typography with vh and vw" by Michael Riethmuller at *Smashing Magazine* (*www.smashingmagazine.com/2016/05/fluid-typography*).

Navigation

Navigation feels a little like the Holy Grail of Responsive Web Design. It is critical to get it right. Because navigation at desktop widths has pretty much been conquered, the real challenges come in re-creating our navigation options on small screens. A number of successful patterns have emerged for small screens, which I will briefly summarize here (FIGURE 17-11):

Top navigation

If your site has just a few navigation links, they may fit just fine in one or two rows at the top of the screen.

Priority +

In this pattern, the most important navigation links appear in a line across the top of the screen alongside a More link that exposes additional options. The pros are that the primary links are in plain view, and the number of links shown can increase as the device width increases. The cons include the difficulty of determining which links are worthy of the prime small-screen real estate.

Select menu

For a medium list of links, some sites use a **`select`** input form element. Tapping the menu opens the list of options using the select menu UI of the operating system, such as a scrolling list of links at the bottom of the screen or on an overlay. The advantage is that it is compact, but on the downside, forms aren't typically used for navigation, and the menu may be overlooked.

Link to footer menu

One straightforward approach places a Menu link at the top of the page that links to the full navigation located at the bottom of the page. The risk with this pattern is that it may be disorienting to users who suddenly find themselves at the bottom of the scroll.

Accordion sub-navigation

When there are a lot of navigation choices with sub-navigation menus, the small-screen solution becomes more challenging, particularly when you can't hover to get more options as you can with a mouse. Accordions that expand when you tap a small arrow icon are commonly used to reveal and hide sub-navigation. They may even be nested several levels deep. To avoid nesting navigation in accordion submenus, some sites simply link to separate landing pages that contain a list of the sub-navigation for that section.

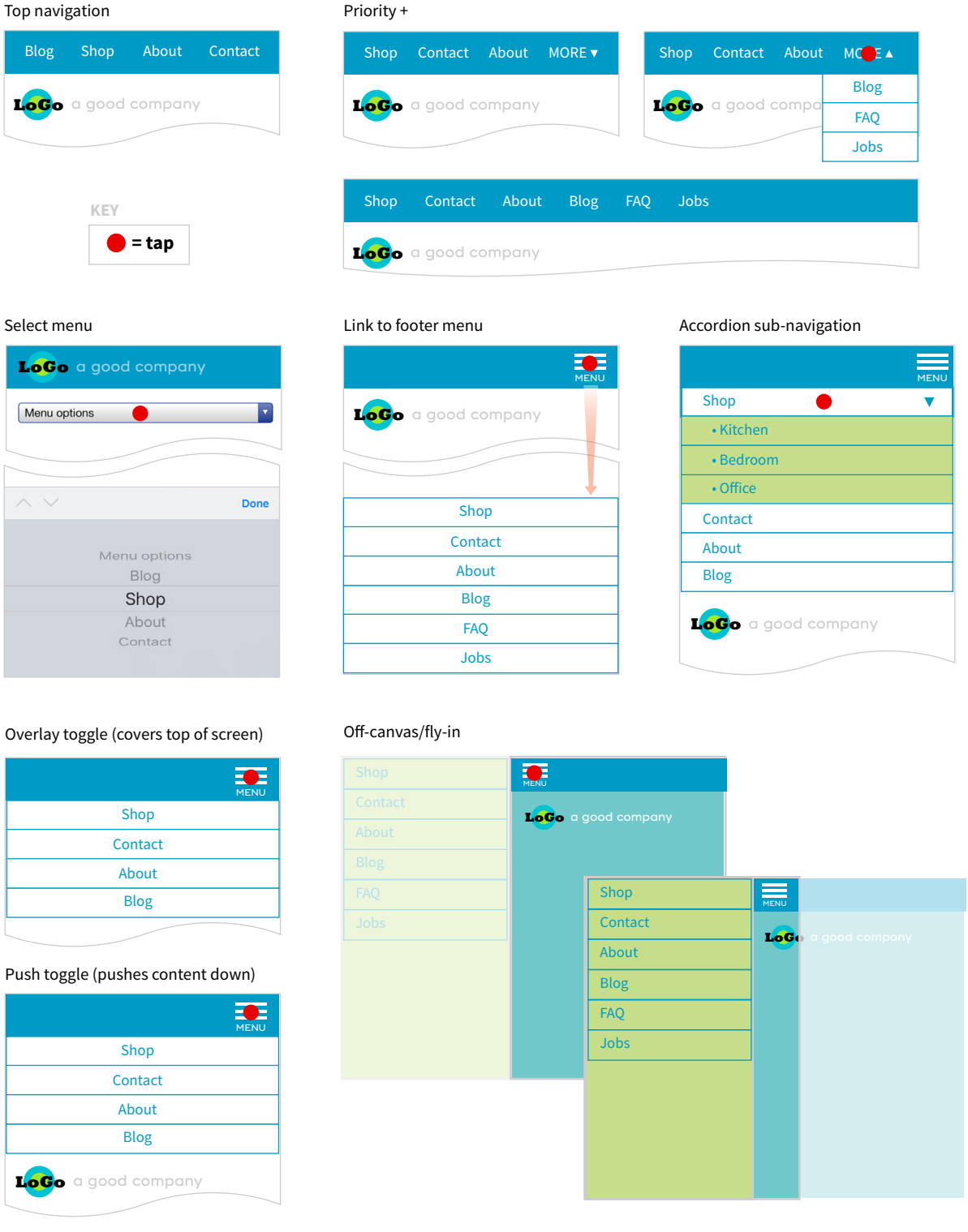


FIGURE 17-11. Responsive navigation patterns.

Designing for Fingers

Keep in mind that people use their fingers to get around on touch devices, which these days include smartphones, tablets, and even desktop-sized screens like Microsoft Surface and iPad Pro.

Links in navigation should be big enough to easily target with thumbs and fingertips. Apple requires 44 pixels for its apps, and that's a good ballpark to keep in mind for links on web pages as well.

Another consideration for touch devices is that there is no hover state. Hovering has become the convention for opening sub-navigation on web pages on the desktop, but with no mouse, that experience is very different with touch. Most devices open the submenu with a second click. If you use hover in your navigation and elsewhere on your site, you'll need to do thorough device testing. Someday, we may be able to write a media query to test for hover, but in the meantime, either avoid it or test the alternatives.

A really great book about all of this stuff is Josh Clark's *Designing for Touch* (A Book Apart).

Push and overlay toggles

In toggle navigation, the navigation is hidden but expands downward when the menu link is tapped. It may push the main content down below it (push toggle) or slide down in front of the content (overlay toggle).

Off-canvas/fly-in

This popular pattern puts the navigation in an off-screen panel to the left or right of the main content that slides into view when you tap the menu icon.

For a deeper dive into the pros and cons of navigation patterns, read Brad Frost's article "Responsive Navigation Patterns" (*bradfrost.com/blog/web/responsive-nav-patterns*). Brad also includes examples of these patterns and more on his Responsive Patterns page (*bradfrost.github.io/this-is-responsive/patterns.html*).

For working examples of these patterns with the code used to create them, see the "Adventures in Responsive Navigation" page assembled by Eric Arbé at *responsivenavigation.net*.

Images

Images require special attention in responsive designs. Here is a quick run-down of some of the key issues, most of which should sound familiar:

- Use responsive image markup techniques (covered in **Chapter 7**) to provide multiple versions of key images for various sizes and resolutions.
- Serve the smallest version as the default to prevent unnecessary data downloads.
- Be sure that important image detail is not lost at smaller sizes. Consider substituting a cropped version of the image for small screens.
- Avoid putting text in graphics, but if it is necessary, provide alternate versions with larger text for small screens.

Special Content

Without the luxury of wide-open, desktop viewports, some of our common page elements pose challenges when it comes to fitting on smaller screens:

Forms

Forms often take a little finagling to fit the available space appropriately. Flexbox is a great tool for adding flexibility and conditional wrapping to form fields and their labels. A web search will turn up some fine tutorials. Also make sure that your form is as efficient as possible, with no unnecessary fields, which is good advice for any screen size. Finally, consider that

form inputs will be used with fingertips, not mouse pointers, so increase the target size by adding ample padding or margins and by making labels tappable to select an input.

Tables

One of the greatest challenges in small-screen design is how to deal with large data tables. Not surprisingly, because there are many types of tables, there are also many solutions. See the **"The Trouble with Tables"** sidebar for more information and resources.

Interactive elements

A big embedded map may be great on a desktop view of a site, but it is less useful when it is the size of a postage stamp. Consider whether some interactive features should be substituted for other methods for performing the same task. In the case of the map, adding a link to a map can trigger the device's native mapping app to open, which is designed to provide a better small-screen experience. Other interactive components, such as carousels, can be adapted for smaller viewports.

The Trouble with Tables

Large tables, such as those shown back in FIGURE 8–1, can be difficult to use on small-screen devices. By default, they are shrunk to fit the screen width, rendering the text in the cells too small to be read. Users can zoom in to read the cells, but then only a few cells may be visible at a time, and it is difficult to parse the organization of headings and columns.

Designers and developers have created a number of approaches for making tables responsive. To be honest, using tables on small devices is still relatively new, so right now what we're seeing is a lot of experimentation and throwing solutions against the wall to see what sticks. Most solutions involve some advanced web development mojo (more than we can take on with only a few chapters under your belt), but I do want you to be familiar with responsive tables. There are three general approaches: scrolling, stacking, and hiding.

For scrolling solutions, the table stays as wide as it needs to be, and users can scroll to the right to see the columns that won't fit. This can be accomplished with JavaScript or CSS alone. You can even anchor the left column to the window so that it stays put when the rest of the table scrolls.

Another approach is to stack up the entries in a long, narrow scroll. Each entry repeats the headers, so the data is always presented with the proper context. Again, you could do this with JavaScript or CSS alone. The downside is that the list can end up very long, and it makes it difficult to compare entries, but at least all of the information is visible without horizontal scrolling.

You may also choose to hide certain columns of information when the page first loads on small devices and give the user the option to click to see the whole table or to toggle on and off specific columns. That is a little more risky from an interaction design perspective. Those columns just might not be seen at all.

CSS Tables and Flexbox are other options for making tabular material responsive. The best approach entirely depends on the type of data you're publishing and how the table is expected to be used. If you are interested in learning more, I recommend the following resources:

- "Accessible, Simple, Responsive Tables" by Davide Rizzo on CSS-Tricks (*css-tricks.com/accessible-simple-responsive-tables/*): A roundup of solutions using CSS tables.
- "CSS-only Responsive Tables" by David Bushell (*dbushell.com/2016/03/04/css-only-responsive-tables/*): A CSS-only scrolling approach using CSS shadows for improved usability.
- "Picking a Responsive Tables Solution" by Jason Grigsby at Cloud Four (*cloudfour.com/thinks/picking-responsive-tables-solution/*).
- Responsive Tables by ZURB Studios (*zurb.com/playground/responsive-tables*): A fixed-left-column scrolling solution using JavaScript and CSS.
- Tablesaw by Filament Group (*github.com/filamentgroup/tablesaw*): A group of JQuery (JavaScript) plug-ins for creating a variety of responsive table effects.

That should give you a feel for some of the aspects of a site that need special attention in a responsive design. We covered content hierarchy, various layout patterns, typography tweaks, responsive navigation patterns, and image strategies, and addressed tables, forms, and interactive features. I'd say that's enough lecturing. Now you'll get some hands-on time in EXERCISE 17-1.

EXERCISE 17-1. Making the bakery home page responsive

We've done a lot of work on the Black Goose Bakery site over the last few chapters, but the resulting site works best on large screens. In this exercise, we're going to back up a few steps and build it again using a small-screen-first strategy, making changes to layout, navigation, typography, and more at strategic breakpoints.

I've done the heavy lifting of writing the necessary styles for each breakpoint, but I will talk you through each step and share the reasoning for the changes. The starting style sheet (*bakery-rwd.css*) as well as the finished style sheet (*bakery-rwd-finished.css*) and the other files for the site are provided with the materials for this chapter. The HTML file, *bakery.html*, hasn't changed since we added the container element to it in **Chapter 16, CSS Layout with Flexbox and Grid**, and we will not need to edit it again.

Getting Started

Open the HTML file (*bakery.html*) in a browser with a Responsive View (see the previous sidebar, **"How Wide Is the Viewport?"**) so you can expand the viewport window and watch the changing pixel dimensions. FIGURE 17-12 shows the page at 320 pixels wide with the default, narrow-screen styles that will be the starting point for this design.

The content of the page is the same as in previous chapters, but if you worked on the exercises in **Chapter 16**, you'll notice that I've changed a few styles to make the initial layout suitable for small screens. Allow me to point out the characteristics of this baseline design:

- **Layout**: The page has a one-column layout for small screens. There are no borders around the main text area, and the Hours section has a scalloped edge on the top instead of the side. That maintains the look and feel, but is more appropriate when the sections are stacked.
- **Navigation:** The navigation menu, which was created with Flexbox, couldn't flex small enough to fit across a small screen. To make it fit, I turned on wrapping (`flex-wrap: wrap;`) and set the width of each `li` to 50% so there would be two on each row. I also made it so they can both grow and shrink as needed (`flex: 1 1 50%`).
- **Conditional header text:** The tagline was taking up a lot of vertical space, and I decided it wasn't critical. I hid the paragraph (`display: none;`) and I will make it visible again when there is more room.
- **Typography:** On small screens, I decided to use a legible sans-serif font for the text and not to employ my web font because it is likely to be difficult to read at small sizes.
- **Images:** I set the `img` elements for the bread and muffin images to `display: block` so they have the full width of the viewport to themselves with no text sneaking in next to them. Setting the side margins to `auto` keeps them centered horizontally.
- **Miscellaneous:**

  — The award appears at the bottom of the page because there is not enough space for it to be positioned at the top.

  — I highlighted a `span` from the 45th to 75th characters to reveal when the line lengths get too long.

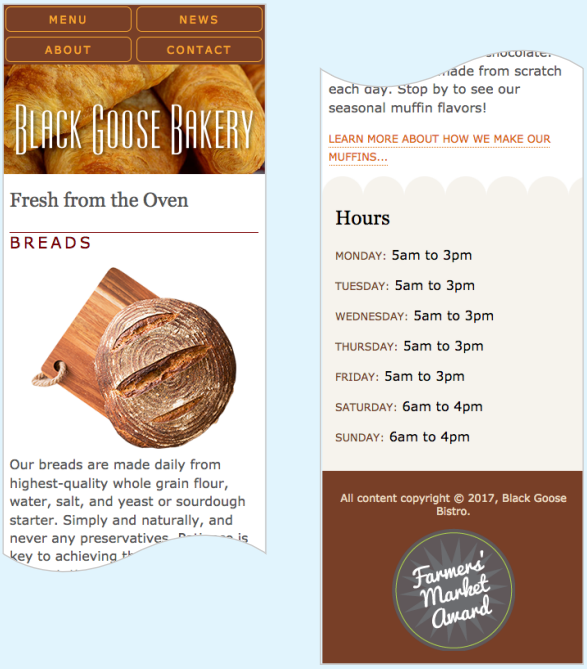


FIGURE 17-12. The small-screen design is our starting point.

Fixing the Navigation

Now we can start tailoring the design for other screen sizes. Using a Responsive View tool, I can resize the viewport and get an instant readout of the dimensions of the window. Give it a try on your browser. Keep making it wider, and you'll see that some things look OK, and some things start looking awkward pretty quickly.

One thing that looks awkward to me right away is the stacked navigation at the top. I'd like it to switch to one centered line as soon as there is room, which to my eye happens when the viewport is 400 pixels wide (FIGURE 17-13).

Are you ready to write your first media query? Open the style sheet (*bakery-rwd.css*) in a text editor. Remember that media queries need to come after other rules for the same declaration, so to keep this exercise simple, we'll add them at the end of the style sheet, before **</style>**. Add this query as you see it here. Remember to make sure you have the right number of nesting curly brackets:

```
@media screen and (min-width: 400px) {
  nav ul li {
    flex: none;
  }
  nav ul {
    justify-content: center;
  }
}
```

This tells the browser that when the page is on a screen and the viewport is 400 pixels or wider, set the "flex" of menu list items to "none." The **none** keyword is equivalent to **`flex: 0 0 auto;`**, so the items are not allowed to grow or shrink and will be sized based on their content. I've centered the flexbox container by setting **`justify-content: center`**.

Save the style sheet and reload the page in the browser. Try resizing the viewport to see how it works at wider sizes. I think this centered arrangement will work for even the widest of screens, so navigation is all set. If you had navigation with additional elements such as an inline logo and a search box, you might find it best to create a few different arrangements over a number of breakpoints.

Floating Images

As I continue to make the viewport gradually wider, I notice that the main images start looking very lonely on a line alone, and that there is room to start wrapping text around them again at about 480 pixels wide. Let's take care of that awkward whitespace by floating the images to the left once the screen reaches 480 pixels (FIGURE 17-14):

```
@media screen and (min-width: 480px) {
  main img {
    float: left;
    margin: 0 1em 1em 0;
  }
}
```

NOTE: If you like, you can include the CSS shapes from **Chapter 15, Floating and Positioning,** for a more interesting text wrap. I've omitted them here for brevity and because of limited browser support.

Text and Typography

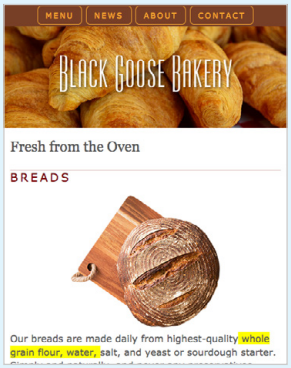
Once the screen gets to be about 600 pixels wide, I feel like there is enough room to introduce some embellishments. There is room for the tagline in the header, so I'll set that to display again.

Now some attention to typography. I like the Stint Ultra Expanded web font, but it isn't key to the company's brand, so I omitted it on the narrow layout because of line length issues. At this breakpoint, I can begin using it because I know it will be more legible and result in comfortable line lengths. I've also loosened up the line height a little. I'll take advantage of the extra space to add a

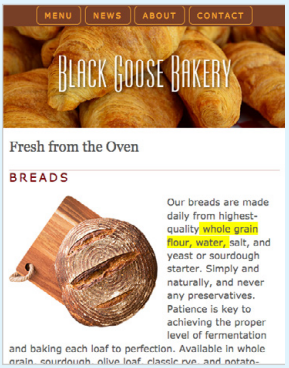
Before breakpoint change

After

FIGURE 17-13. The navigation started to look awkward, so I add a breakpoint at 400 pixels to switch it to one line.



A breakpoint is needed to fill in the awkward space around the image.



At 480 pixels wide, the image is floated to the left.

FIGURE 17-14. The images float left once there is enough width to accommodate wrapping text.

EXERCISE 17-1. Continued

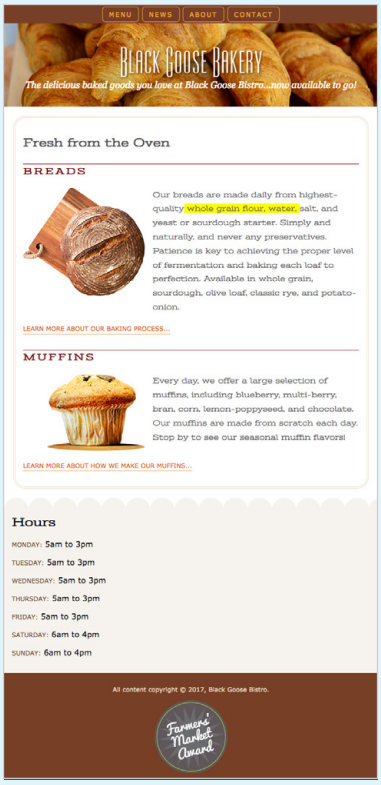


FIGURE 17-15. This medium size layout is well suited for tablet-sized devices.

rounded border around the main text area to bring it closer to the original brand identity for the site. The result is an enhanced one-column layout that is well suited for tablet-sized devices (FIGURE 17-15).

Here is the media query for the 600-pixel breakpoint. Add this to the bottom of the style sheet after the other two queries:

```
@media screen and (min-width: 600px) {
  header p {
    display: block;
    margin-top: -1.5em;
    font-family: Georgia, serif;
    font-style: italic;
    font-size: 1.2em;
  }
  main, h2, h3 {
    font-family: 'Stint Ultra Expanded', Georgia, serif;
  }
  h2, h3 {
    font-weight: bold;
  }
  main {
    line-height: 1.8em;
    padding: 1em;
    border: double 4px #EADDC4;
    border-radius: 25px;
    margin: 2.5%;
  }
}
```

Multicolumn Layout

As I continue to make the viewport wider and pay attention to the yellow highlighted span of characters, I see that the text line is growing longer than 75 characters. I could increase the font size or the margins, but I think this is a good point to introduce a second column to the layout. If you aren't targeting a specific device, the exact breakpoint is subjective. I've chosen 940 pixels as the point above which the page gets a columned layout.

I've simply taken the grid layout styles from the previous chapter and reapplied them here. On the **aside** element, I moved the scalloped background graphic to the left edge. In addition, I set a maximum width of 1200px on the container and set its side margins to **auto**, so if the browser window is wider than 1,200 pixels, the layout will stay a fixed width and get centered in the viewport. Finally, I absolutely positioned the award graphic at the top of the page now that there's enough room (FIGURE 17-16).

Add this final media query at the end of the style sheet. You can copy and paste them from the final exercise in **Chapter 16** (that's what I did) and make a few tweaks to the **#container** and **#aside** rules as shown:

```
@media screen and (min-width: 940px) {
  #container {
    display: grid;
    grid-template-rows: auto min-height 5em;
    grid-template-columns: minmax(25em, 1fr) 16em;
    grid-template-areas:
      "banner banner"
      "main hours"
      "footer footer";
    max-width: 1200px;
```

```
    margin: 0 auto;
    position: relative;
  }
  header {
    grid-area: banner;
  }
  main {
    grid-area: main;
  }
  aside {
    grid-area: hours;
    background: url(images/scallop.png) repeat-y left top;
    background-color: #F6F3ED;
    padding: 1em;
    padding-left: 45px;
  }
  footer {
    grid-area: footer;
  }
  #award {
    position: absolute;
    top: 30px;
    left: 50px;
  }
}
```

And we're done! Is this the most sophisticated responsive site ever? Nope. Is there even more we could do to improve the design at various screen sizes? Certainly! But now you should have a feel for what it's like to start with a small-screen design and make changes that optimize for increasingly larger sizes. Consider it a modest first step to future adventures in RWD.

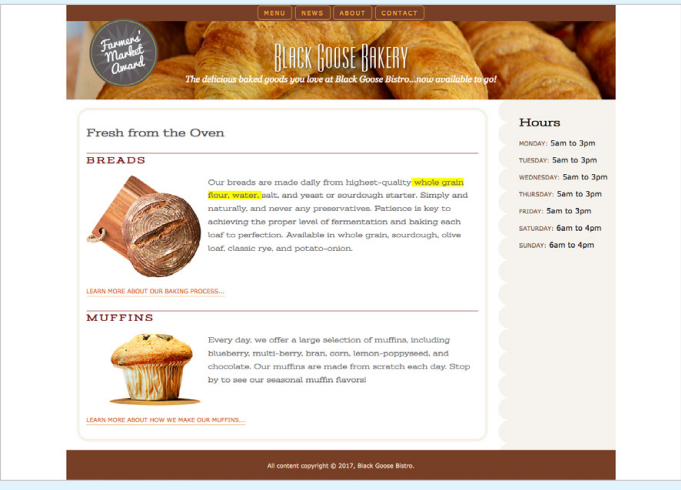


FIGURE 17-16. The two-column grid layout is appropriate for viewports over 940 pixels. On very wide screens, as shown here, the container stops expanding at 1,200 pixels wide and is centered horizontally.

NOTE

The highlighted background on the length span should be turned off before you publish, but I've left it visible in the figures so you can see how our line length is faring across layouts.

A FEW WORDS ABOUT TESTING

In the previous exercise, we relied on the Responsive View in a modern browser to make decisions about style changes at various sizes, but although it's a handy tool for creating an initial design, much more testing is required before the design can be considered ready for final launch. That is even more critical for sites that include features that rely on JavaScript or server-side functionality.

There are three general options for testing sites: real devices, emulators, and third-party services. We'll look at each in this section.

Real Devices

There is really no substitute for testing a site on a variety of real devices and operating systems. Beyond just seeing how the site looks, testing on real devices shows you how your site *performs*. How fast does it load? Are the links easy to tap? Do all the interactive features work smoothly? Do they work at all?

Web development companies may have a device lab comprising iPhones and iPads of various sizes, Android smartphones and tablets of various sizes, and Macs and PCs with recent operating systems (Windows and Linux) that can be used by designers and developers for testing sites (FIGURE 17-17). The size of the device lab depends on the size of the budget, of course (electronic devices aren't cheap!).

Building a Device Lab

If you want to set up your own device lab, I recommend reading the primer ***Building a Device Lab*** by Destiny Montague and Lara Hogan (Five Simple Steps Publishing). The book is a summary of everything the authors learned while creating a killer device lab for Etsy. It is available for free at *buildingadevicelab.com*.

FIGURE 17-17. The device lab at Filament Group in Boston, Massachusetts.

If you don't have the luxury of working at a big company with a big lab, there are alternatives:

- If you live in a big city, you may be near a device lab that is open for public use. Check the *opendevicelab.com* site to see if there is one near you.

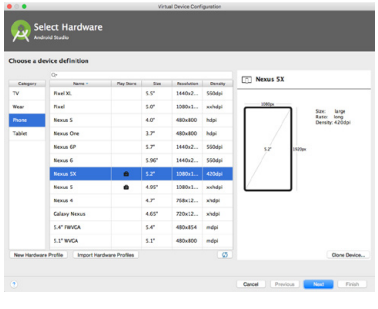
- You can build your own lab with a collection of used devices. At minimum, you should have access to an iPhone, Android phone, iPad, 7" tablet (like iPad Mini), and computers running macOS and Microsoft Windows. The good news is that you generally don't need a data plan for every device because you can test over WiFi.
- If buying devices is not feasible, you can ask friends and coworkers to borrow their phones and tablets briefly. Asking permission at a mobile retail store to load web pages on their devices is not unheard of.

If you do have multiple real devices for testing, using a synchronization tool makes the process a whole lot smoother. Software like BrowserSync (*browsersync.io*) and Ghostlab (*www.vanamco.com/ghostlab/*) runs on your computer and beams whatever is on your screen to all your devices simultaneously so you don't need to load the page on each one individually. It's like magic!

Emulators

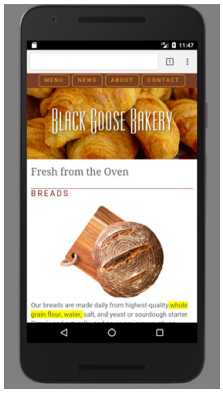
If a particular device is out of your reach, you could use an emulator, a desktop application that emulates mobile device hardware and operating systems. The emulator presents a window that shows exactly how your site would behave on that particular device (FIGURE 17-18). Emulators require a lot of space on your computer and they can be buggy, but it is certainly better than not testing on that device at all.

A good starting point for exploring emulators is Maximiliano Firtman's "Mobile Emulators & Simulators: The Ultimate Guide" (*www.mobilexweb.com/emulators*).



The Android Emulator lets you set up a wide variety of phones, televisions, wearables, and tablets for testing. I chose a Nexus 5X.

The Nexus 5X emulator displays an image of the device at actual size. All of the buttons work as they would on the phone.

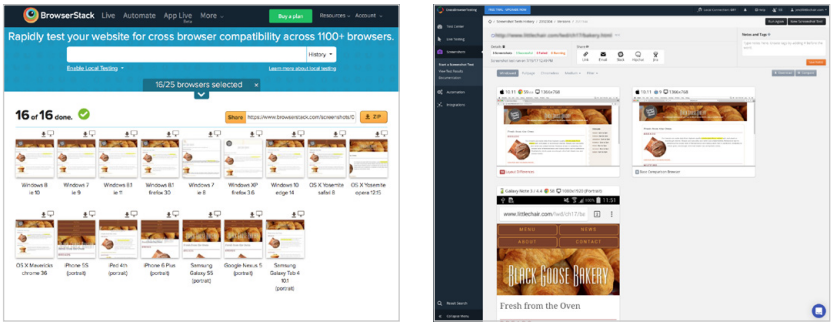


The Bakery page viewed on the Nexus 5X emulator.

FIGURE 17-18. Examples of the Android Emulator (download at *developer.android.com/studio/index.html*).

Third-Party Services

Another option for testing your site on over 1,000 devices is to subscribe to a service like BrowserStack (*browserstack.com*) or CrossBrowserTesting (*crossbrowsertesting.com*). For a monthly fee, you get access to a huge variety of device simulators (FIGURE 17-19). There are many such services available, some of which are free or offer free trials. They don't give you the same insights as testing on actual devices, but it is another better-than-nothing alternative.



BrowserStack.com CrossBrowserTesting.com

FIGURE 17-19. Screenshots generated by BrowserStack and CrossBrowserTesting (using free trial tools). Notice the variation in how the bakery page displays. This is why we test!

MORE RWD RESOURCES

We've covered the mechanics of using fluid layouts, flexible images, and media queries to make a page that is usable across a wide range of screen sizes. We've looked at the design concerns and some common responsive patterns for layout, navigation, typography, and images. You even got a chance to try out creating a responsive page on your own. But this is really only the tip of the iceberg, and I encourage you to continue learning about RWD, particularly if you are considering web design or development as a career. Following is a list of RWD resources that I've found helpful and should point you in the right direction.

Books

***Responsive Web Design, 2e*, by Ethan Marcotte (A Book Apart)**

This book is required reading. Ethan goes into much greater detail than I was able to here on how to calculate flexible grids and how to use media queries. Plus, it's just plain fun to read.

***Learning Responsive Web Design: A Beginner's Guide* by Clarissa Peterson (O'Reilly)**

Clarissa provides a comprehensive overview of all aspects of responsive design, from detailed code examples to broad strategies on workflow and mobile-first design.

***Smashing Book #5: Real-Life Responsive Web Design*, various authors (Smashing Magazine)**

A collection of practical techniques and strategies from prominent web designers.

***Atomic Design* by Brad Frost (self-published)**

Brad describes his modular approach to RWD, which has become quite popular for large site development.

***Responsive Design Workflow* by Stephen Hay (New Riders)**

Stephen Hay introduces his "design in the browser" method to creating responsive sites. This book is jam-packed with suggestions on how to approach web design and development.

***Implementing Responsive Design* by Tim Kadlec (New Riders)**

Tim Kadlec is a leader in the mobile web design community, and his book is a comprehensive guide to designing and building a responsive site.

NOTE

Most of these titles were written before CSS Grid Layout became a viable option. Keep in mind that you have advanced tools for flexible layouts not mentioned in these books.

Online Resources

***Responsive Web Design Is…* (responsivedesign.is)**

A collection of articles and podcasts about web design. You can also sign up for the "RWD Weekly" newsletter and keep your finger on the pulse of RWD. The site is a side project of Justin Avery and Simple Things.

***Responsive Resources* (bradfrost.github.io/this-is-responsive/resources.html)**

For one-stop shopping for everything you could possibly want to know about RWD, look no further than Brad Frost's Responsive Resources. He has gathered hundreds of links to resources related to strategy, design tools, layout, media queries, typography, images, components, development, testing, content management systems, email, tutorials, and more. Seriously, there is enough here to keep you busy for months.

***Media Queries* (mediaqueri.es)**

A gallery of exceptional examples of responsive websites curated by Eivind Uggedal.

TEST YOURSELF

Here we are at the end of another chapter, so you know what that means... *quiz time*! Get the answers in **Appendix A** if you're stumped.

1. What makes a responsive site different from a mobile (m-dot) site?

2. What does this do?

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

3. How do you make sure an image gets smaller when its container gets smaller in the layout?

4. What does this do?

```
@media screen and (min-width: 60em) {
  body {
    margin: 0 10%;
  }
}
```

5. What are some strategies for creating a layout that adjusts to the available width of the viewport?

6. What is the advantage of using ems as a measurement in media queries?

7. List three ways in which a media query may be used.

8. Name three tweaks you may make to typography to make it work well on small screens.

9. How might you handle navigation with a lot of submenus on a small screen?

10. List three options for testing websites on multiple devices.

CHAPTER
18

TRANSITIONS, TRANSFORMS, AND ANIMATION

IN THIS CHAPTER

We've seen CSS used for visual effects like rounded corners, color gradients, and drop shadows that previously had to be created with graphics. In this chapter, we'll look at some CSS3 properties for producing animated interactive effects that were previously possible only with JavaScript or Flash.

We'll start with CSS Transitions, a nifty way to make style changes fade smoothly from one to another. Then we'll discuss CSS Transforms for repositioning, scaling, rotating, and skewing elements and look at how you can animate them with transitions. I'm going to close out the chapter with brief introductions to 3-D Transforms and CSS Animation, which are important to know about but are too vast a topic to cover here, so I'll give you just a taste.

The problem with this chapter is that animation and time-based effects don't work on paper, so I can't show them off right here. I did the next best thing, though, and made the source code for the figures available in the materials for this chapter (*learningwebdesign.com/5e/materials*) in a folder called *figures*. Just open the file in your browser.

EASE-Y DOES IT (CSS TRANSITIONS)

Picture, if you will, a link in a navigation menu that changes from blue to red when the mouse hovers over it. The background is blue...mouse passes over it...BAM! Red! It goes from state to state instantly, with no states in between. Now imagine putting your mouse over the link and having the background gradually change from blue to red, passing through several shades of purple on the way. It's smoooooth. And when you remove the mouse, it fades back down to blue again.

That's what CSS Transitions do. They smooth out otherwise abrupt changes to property values between two states over time by filling in the frames in between. Animators call that tweening. When used with reserve, CSS Transitions can add sophistication and polish to your interfaces and even improve usability.

CSS Transitions were originally developed by the WebKit team for the Safari browser, and they are a Working Draft at the W3C (see **Note**). Browser support for Transitions is excellent (see the **"CSS Transition Support"** sidebar), so there is no reason not to use them in your designs, particularly if you treat them as an enhancement. For example, on the rare non-supporting browser (I'm looking at you, old IE), our link snapping directly from blue to red is not a big deal.

NOTE

You can read CSS Transitions Module for yourself at www.w3.org/TR/css-transitions-1/.

Transition Basics

Transitions are a lot of fun, so let's give them a whirl. When applying a transition, you have a few decisions to make, each of which is set with a CSS property:

- Which CSS property to change (`transition-property`) *(Required)*
- How long it should take (`transition-duration`) *(Required)*
- The manner in which the transition accelerates (`transition-timing-function`)
- Whether there should be a pause before it starts (`transition-delay`)

Transitions require a beginning state and an end state. The element as it appears when it first loads is the beginning state. The end state needs to be triggered by a state change such as `:hover`, `:focus`, or `:active`, which is what we'll be using for the examples in this chapter. You could use JavaScript to change the element (such as adding a `class` attribute) and use that as a transition trigger as well.

Let's put that all together with a simple example. Here is that blue-to-red link you imagined earlier (FIGURE 18-1). There's nothing special about the markup. I added a `class` so I could be specific about which links receive transitions.

The transition properties are applied to the object that will be transitioned—in this case, the `a` element in its normal state. You'll see them in the set of other declarations for `.smooth`, like `padding` and `background-color`. I've changed the background color of the link to red by declaring the `background-color` for the `:hover` state (and `:focus` too, in case someone is tabbing through links with a keyboard).

■ SUPPORT TIP

CSS Transition Support

The good news is that all modern browsers released since 2013 support CSS transition properties without the need for prefixes. There are a few holes in support you should know about:

- Most notably, Internet Explorer versions 9 and earlier do not support transitions and ignore transition properties entirely.
- Chrome and Safari versions released between 2010 and 2013 support transitions with the `-webkit-` prefix. Later versions do not require a prefix.
- On mobile, iOS versions 3.1–6.0 (2010–2013) and Android versions 2.1–4.3 (2009–2013) require the `-webkit-` prefix. Later versions do not require prefixes.
- Firefox versions released between 2011 and 2012 require the `-moz-` prefix, but they are nearly extinct as I write this.

As always, check your own server's statistics (be sure to pay attention to mobile use) to see which browsers you need to support, and check *CanIUse.com* for support and bug details.

In the examples throughout this chapter, I use only the standard (non-prefixed) properties. If you need to support browsers that require prefixes, I suggest using Autoprefixer, which is discussed in **Chapter 19, More CSS Techniques**. And remember, when using prefixed properties, always include the non-prefixed version last for forward compatibility with supporting browsers.

THE MARKUP

```
<a href="…" class="smooth">awesomesauce</a>
```

THE STYLES

```
.smooth {
  display: block;
  text-decoration:none;
  text-align: center;
  padding: 1em 2em;
  width: 10em;
  border-radius: 1.5em;
  color: #fff;
  background-color: mediumblue;
  transition-property: background-color;
  transition-duration: 0.3s;
}
.smooth:hover, .smooth:focus {
  background-color: red;
}
```

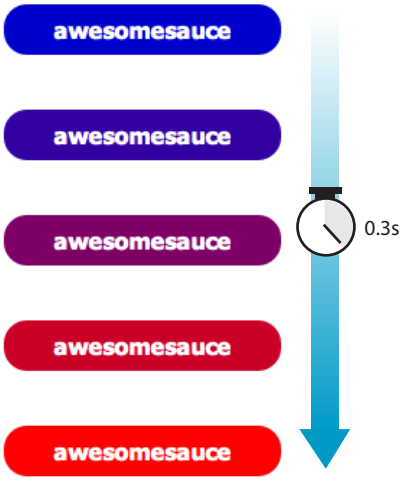


FIGURE 18-1. The background color of this link gradually fades from blue to red over .3 seconds when ~~awesome sauce~~ a transition is applied.

Specifying the property

transition-property

Values: *property-name* | `all` | none

Default: `all`

Applies to: all elements, **`:before`** and **`:after`** pseudo-elements

Inherits: no

`transition-property` identifies the CSS property that is changing and that you want to transition smoothly. In our example, it's the **`background-color`**. You can also change the foreground color, borders, dimensions, font- and text-related attributes, and many more. TABLE 18-1 lists the animatable CSS properties as of this writing. The general rule is that if its value is a color, length, or number, that property can be a transition property.

Add the transition properties to the object that will be transitioned.

How long should it take?

transition-duration

Values: *time*

Default: 0s

Applies to: all elements, **`:before`** and **`:after`** pseudo-elements

Inherits: no

`transition-duration` sets the amount of time it takes for the animation to complete in seconds (**`s`**) or milliseconds (**`ms`**). I've chosen .3 seconds, which is just enough to notice something happened but not so long that the transition feels sluggish or slows the user down. There is no correct duration, of course, but I've found that .2s seems to be a popular transition time for UI elements. Experiment to find the duration that makes sense for your application.

TABLE 18-1.
Animatable CSS properties

Backgrounds

- background-color
- background-position

Borders and outlines

- border-bottom-color
- border-bottom-width
- border-left-color
- border-left-width
- border-right-color
- border-right-width
- border-top-color
- border-top-width
- border-spacing
- outline-color
- outline-width

Color and opacity

- color
- opacity
- visibility

Font and text

- font-size
- font-weight
- letter-spacing
- line-height
- text-indent
- text-shadow
- word-spacing
- vertical-align

Element box measurements

- height
- width
- max-height
- max-width
- min-height
- min-width
- margin-bottom

Continues...

Timing Functions

transition-timing-function

Values: ease | linear | ease-in | ease-out | ease-in-out | step-start | step-end | steps | cubic-bezier(#,#,#,#)

Default: ease

Applies to: all elements, **:before** and **:after** pseudo-elements

Inherits: no

The property and the duration are required and form the foundation of a transition, but you can refine it further. There are a number of ways a transition can roll out over time. For example, it could start out fast and then slow down, start out slow and speed up, or stay the same speed all the way through, just to name a few possibilities. I think of it as the transition "style," but in the spec, it is known as the timing function or easing function.

The timing function you choose can have a big impact on the feel and believability of the animation, so if you plan on using transitions and CSS animations, it is a good idea to get familiar with the options.

If I set the **transition-timing-function** to **ease-in-out**, the transition will start out slow, then speed up, then slow down again as it comes to the end state.

```
.smooth {
  …
  transition-property: background-color;
  transition-duration: 0.3s;
  transition-timing-function: ease-in-out;
}
```

The **transition-timing-function** property takes one of the following keyword values:

ease

Starts slowly, accelerates quickly, and then slows down at the end. This is the default value and works just fine for most short transitions.

linear

Stays consistent from the transition's beginning to end. Because it is so consistent, some say it has a mechanical feeling.

ease-in

Starts slowly, then speeds up.

ease-out

Starts out fast, then slows down.

ease-in-out

Starts slowly, speeds up, and then slows down again at the very end. It is similar to **ease**, but with less pronounced acceleration in the middle.

`cubic-bezier(x1,y1,x2,y2)`

The acceleration of a transition can be plotted with a curve called a Bezier curve. The steep parts of the curve indicate a fast rate of change, and the flat parts indicate a slow rate of change. FIGURE 18-2 shows the Bezier curves that represent the function keywords as well as a custom curve I created. You can see that the **ease** curve is a tiny bit flat in the beginning, gets very steep (fast), then ends flat (slow). The **linear** keyword, on the other hand, moves at a consistent rate for the whole transition.

You can get the feel of your animation *just right* by creating a custom curve. The site *Cubic-Bezier.com* is a great tool for playing around with transition timing and generating the resulting code. The four numbers in the value represent the x and y positions of the start and end Bezier curve handles (the pink and blue dots in FIGURE 18-2).

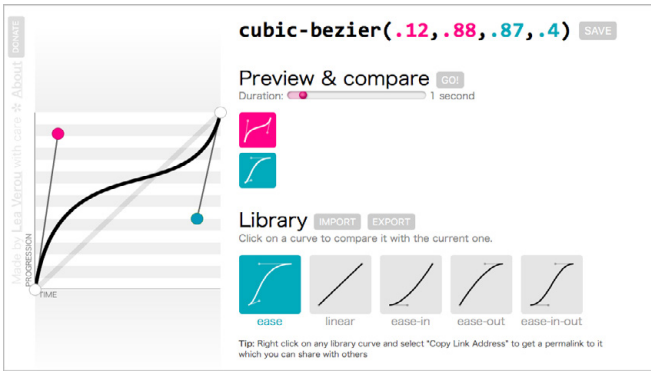


FIGURE 18-2. Examples of Bezier curves from Cubic-Bezier.com. On the left is my custom curve that starts fast, slows down, and ends fast.

`steps(#, start|end)`

Divides the transitions into a number of steps as defined by a stepping function. The first value is the number of steps, and the **start** and **end** keywords define whether the change in state happens at the beginning (**start**) or end of each step. Step animation is especially useful for keyframe animation with sprite images. For a better explanation and examples, I recommend the article "Using Multi-Step Animations and Transitions," by Geoff Graham on CSS-Tricks (*css-tricks.com/using-multi-step-animations-transitions/*).

`step-start`

Changes states in one step, at the beginning of the duration time (the same as **steps(1,start)**). The result is a sudden state change, the same as if no transition had been applied at all.

`step-end`

Changes states in one step, at the end of the duration time (the same as **steps(1,end)**).

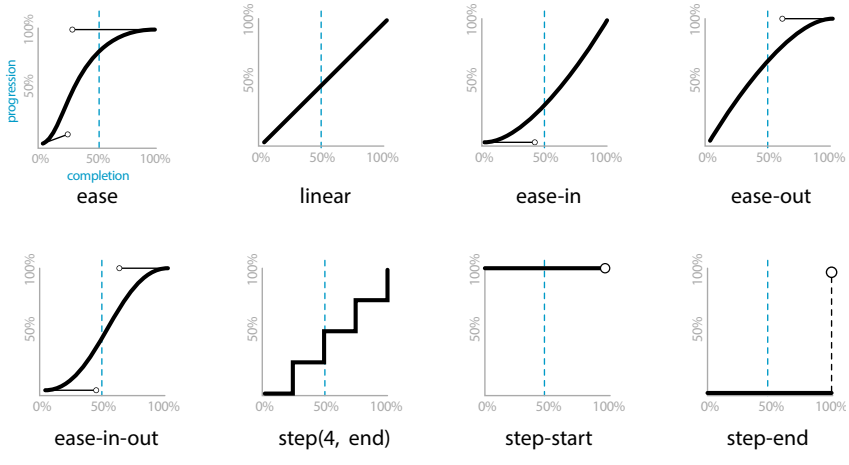
TABLE 18-1. *Continued.*

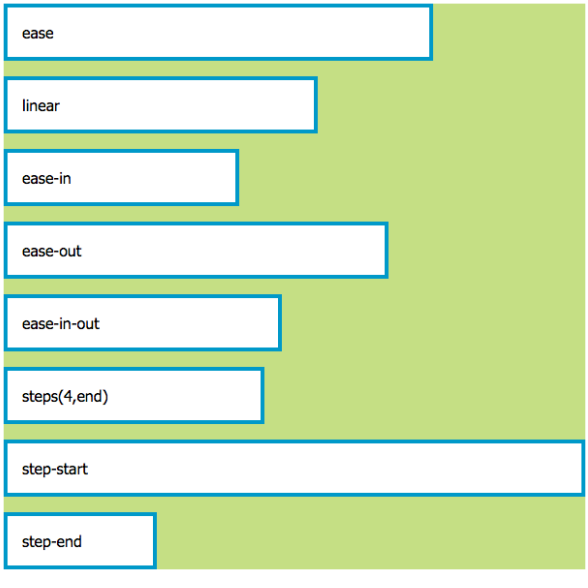
`margin-left`
`margin-top`
`padding-bottom`
`padding-left`
`padding-right`
`padding-top`
Position
`top`
`right`
`bottom`
`left`
`z-index`
`clip-path`
Transforms (not in the spec as of this writing, but supported)
`transform`
`transform-origin`

NOTE

The W3C has broken out the timing functions into their own spec so they are easier to share among modules. It is available at www.w3.org/TR/css-timing-1/.

It's difficult to show the various options on a still page, but I have put together a little demo, which is illustrated in FIGURE 18-3 and available in the *figures* folder with the materials for this chapter. The width of each labeled element (white with a blue border) transitions over the course of 4 seconds when you hover over the green box. They all arrive at their full width at exactly the same time, but they get there in different manners. The image shown in FIGURE 18-3 was taken at the 2-second mark, halfway through the duration of the transition.





The width of the white boxes is set to transition from 0 to 100% width over 4 seconds. This screenshot shows the progress after 2 seconds (50%) for each timing function.

FIGURE 18-3. In this **`transition-timing-function`** demo, the elements reach full width at the same time but vary in the manner in which they get there. If you'd like to see it in action, the *ch18_figures.html* file is available with the materials for this chapter.

Setting a Delay

`transition-delay`

Values: *time*

Default: 0s

Applies to: all elements, **`:before`** and **`:after`** pseudo-elements

Inherits: no

The **`transition-delay`** property, as you might guess, delays the start of the animation by a specified amount of time. In the following example, the background color transition starts .2 seconds after the pointer moves over the link.

```
.smooth {
…
  transition-property: background-color;
  transition-duration: 0.3s;
  transition-timing-function: ease-in-out;
  transition-delay: 0.2s;
}
```

The Shorthand transition Property

Thankfully, the authors of the CSS3 spec had the good sense to give us the shorthand **`transition`** property to combine all of these properties into one declaration. You've seen this sort of thing with the shorthand **`border`** property. Here is the syntax:

```
transition: property duration timing-function delay;
```

The values for each of the **`transition-*`** properties are listed out, separated by character spaces. The order isn't important as long as the duration (which is required) appears before delay (which is optional). If you provide only one time value, it will be assumed to be the duration.

Using the blue-to-red link example, we could combine the four transition properties we've applied so far into this one line:

```
.smooth {
  …
  transition: background-color 0.3s ease-in-out 0.2s;
}
```

Definitely an improvement.

Applying Multiple Transitions

So far, we've changed only one property at a time, but it is possible to transition several properties at once. Let's go back to the "awesomesauce" link example. This time, in addition to changing from blue to red, I'd like the **`letter-spacing`** to increase a bit. I also want the text color to change to black,

but more slowly than the other animations. FIGURE 18-4 attempts to show these transitions on this static page.

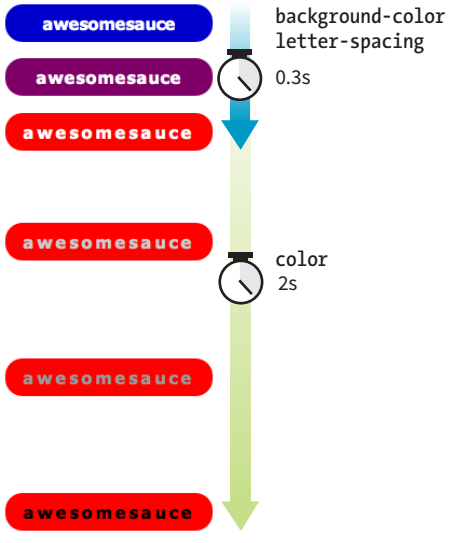


FIGURE 18-4. The **color**, **background-color**, and **letter-spacing** change at different paces.

One way to do this is to list all of the values for each property separated by commas, as shown in this example:

```
.smooth {
  …
  transition-property: background-color, color, letter-spacing;
  transition-duration: 0.3s, 2s, 0.3s;
  transition-timing-function: ease-out, ease-in, ease-out;

}
.smooth:hover, .smooth:focus {
  background-color: red;
  letter-spacing: 3px;
  color: black;
}
```

The values are matched up according to their positions in the list. For example, the transition on the **color** property (second in the list) has a duration of 2s and uses the **ease-in** timing function. If one list has fewer values than the others, the browser repeats the values in the list, starting over at the beginning. In the previous example, if I had omitted the third value (.3s) for **transition-duration**, the browser would loop back to the beginning of the list and use the first value (.3s) for **letter-spacing**. In this case, the effect would be the same.

You can line up values for the shorthand **transition** property as well. The same set of styles we just saw could also be written as follows:

```
.smooth {
  …
  transition: background-color 0.3s ease-out,
              color 2s ease-in,
              letter-spacing 0.3s ease-out;
}
```

A Transition for All Occasions

But what if you just want to add a little bit of smoothness to all your state changes, regardless of which property might change? For cases when you want the same duration, timing function, and delay to apply to all transitions that might occur on an element, use the **`all`** value for **`transition-property`**. In the following example, I've specified that any property that might change for the **`.smooth`** element should last .2 seconds and animate via the **`ease-in-out`** function.

```
.smooth {
  …
  transition: all 0.2s ease-in-out;
}
```

For user interface changes, a short, subtle transition is often all you need for all your transitions, so the **`all`** value will come in handy. Well, that wraps up our lesson on CSS3 Transitions. Now you give it a try in EXERCISE 18-1.

EXERCISE 18-1. Trying out transitions

In this exercise, we're going to create the rollover and active states for a menu link (FIGURE 18-5) with animated transitions. I've put together a starter document (*exercise_18-1.html*) for you in the *materials* folder for this chapter at *learningwebdesign.com/5e/materials*. Be sure you are using an up-to-date desktop browser to view your work (see **Note**).

Normal state.

`:hover`, **`:focus`**
The background and border colors change.

`:active`
Link appears to be pressed down.

FIGURE 18-5. In this exercise, we'll create transitions between these link states.

NOTE

If you're using a touch device for this exercise, you'll miss out on this effect because there is no hover state on touch screens. You may see the hover state with a single tap. Transitions triggered by a click/tap or when the page loads will work on all devices, but they are not covered here.

EXERCISE 18-1. Continued

First, take a look at the styles that are already applied. The list has been converted to a horizontal menu with Flexbox. The **a** element has been set to display as a block element, underlines are turned off, dimensions and padding are applied, and the color, background color, and border are established. I used the **box-shadow** property to make it look as though the links are floating off the page.

1. Now we'll define the styles for the hover and focus states. When the user puts the pointer over or tabs to the link, make the background color change to green (**#c6de89**) and the border color change to a darker shade of green (**#a3c058**).

```
a:hover, a:focus {
  background-color: #c6de89;
  border-color: #a3c058;
}
```

2. While the user clicks the link (**:active**), make it move down by 3 pixels as though it is being pressed. Do this by setting the **a** element's **position** to relative and its **top** position to 0px , and then change the value of the **top** property for the active state. This moves the link 3 pixels away from the top edge (in other words, down).

   NOTE: Setting the top to 0px in the initial state is for working around a bug that arises when transitioning the **top**, **bottom**, **left**, and **right** properties.

```
a {
  …
  position: relative;
  top: 0px;
}
a:active {
  top: 3px;
}
```

3. Logically, if the button were pressed down, there would be less room for the shadow, so we'll reduce the **box-shadow** distance as well.

```
a:active {
  top: 3px;
  box-shadow: 0 1px 2px rgba(0,0,0,.5);
}
```

4. Save the file and give it a try in the browser. The links should turn green and move down when you click or tap them. I'd say it's pretty good just like that. Now we can enhance the experience by adding some smooth transitions.
5. Make the background and border color transition ease in over 0.2 seconds, and see how that changes the experience of using the menu. I'm using the shorthand **transition** property to keep the code simple. I'm also using the default **ease** timing function at first so we can omit that value.

   I'm not using any vendor prefixes here because modern browsers don't need them. If you wanted to support mobile browsers released in 2013 and earlier, you could include the **-webkit-** prefixed version as well, but since this isn't production code, we're fine without it.

```
a {
  transition: background-color 0.2s,
              border-color 0.2s;
}
```

6. Save your document, open it in the browser, and try moving your mouse over the links. Do you agree it feels nicer? Now I'd like you to try some other duration values. See if you can still see the difference with a 0.1s duration. Now try a full second (1s). I think you'll find that 1 second is surprisingly slow. Try setting it to several seconds and trying out various **timing-function** values (just add them after the duration times). Can you tell the difference? Do you have a preference? When you are done experimenting, set the duration back to 0.2 seconds.
7. Now let's see what happens when we add a transition to the downward motion of the link when it is clicked or tapped. Transition both the **top** and **box-shadow** properties because they should move in tandem. Let's start with a 0.2s duration like the others.

```
a {
  transition:
    background-color 0.2s,
    border-color 0.2s,
    top 0.2s,
    box-shadow 0.2s;
}
```

   Save the file, open it in the browser, and try clicking the links. That transition really changes the experience of using the menu, doesn't it? The buttons feel more difficult to "press." Try increasing the duration. Do they feel even more difficult? I find it interesting to see the effect that timing has on the experience of a user interface. It is important to get it right and not make things feel sluggish. I'd say that a very short transition such as 0.1 second—or even no transition at all—would keep these buttons feeling snappy.
8. If you thought increasing the duration made the menu uncomfortable to use, try adding a short 0.5-second delay to the **top** and **box-shadow** properties.

```
a {
  transition:
    background-color 0.2s,
    border-color 0.2s,
    top 0.1s 0.5s,
    box-shadow 0.1s 0.5s;
}
```

   I think you'll find that little bit of extra time makes the whole thing feel broken. Timing is everything!

CSS TRANSFORMS

transform

Values: rotate() | rotateX() | rotateY() | rotateZ() | rotate3d() | translate() | translateX() | translateY() | scale() | scaleX() | scaleY() | skew() | skewX() | skewY() | none

Default: none

Applies to: transformable elements (see sidebar)

Inherits: no

The CSS3 Transforms Module (*www.w3.org/TR/css-transforms-1*) gives authors a way to rotate, relocate, resize, and skew HTML elements in both two- and three-dimensional space. It is worth noting up front that transforms change how an element displays, but it is not motion- or time-based. However, you can animate from one transform state to another using transitions or keyframe animations, so they are useful to learn about in the context of animation.

This chapter focuses on the more straightforward two-dimensional transforms because they have more practical uses. Transforms are supported on virtually all current browser versions without vendor prefixes (see the sidebar **"CSS Transforms Support"** for exceptions).

You can apply a transform to the normal state of an element, and it appears in its transformed state when the page loads. Just be sure that the page is still usable on browsers that don't support transforms. It is common to introduce a transform only when users interact with the element via **:hover** or a JavaScript event. Either way, transforms are a good candidate for progressive enhancement—if an IE8 user sees an element straight instead of at a jaunty angle, it's probably no biggie.

FIGURE 18-6 shows a representation of four two-dimensional transform functions: **rotate()**, **translate()**, **scale()**, and **skew()** (see **Note**). The dashed outline shows the element's original position.

Transformable Elements

You can apply the **transform** property to most element types:

- HTML elements with replaced content, such as **img**, **canvas**, form inputs, and embedded media
- Elements with their display set to **block**, **inline-block**, **inline-table** (or any of the **table-*** display types), **grid**, and **flex**

It may be easier to note the element types you *cannot* transform, which include:

- Non-replaced inline elements, like **em** or **span**
- Table columns and column groups (but who'd want to?)

NOTE

There are actually five 2-D transform functions in the CSS spec. The fifth, **matrix()***, allows you to craft your own combined transformation using six values and some badass trigonometry. There are tools that can take a number of transforms and combine them into a matrix function, but the result isn't very user-friendly. Fascinating in theory, but more than I want to take on personally.*

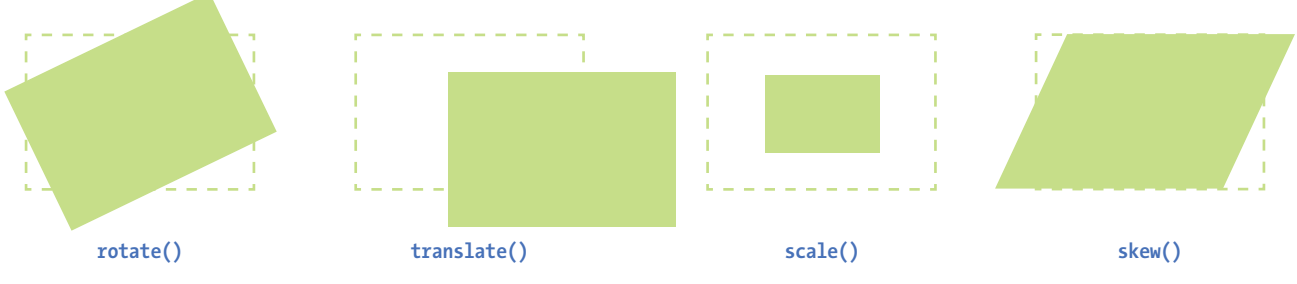


FIGURE 18-6. Four types of transforms: **rotate()**, **translate()**, **scale()**, and **skew()**.

■ SUPPORT TIP

CSS Transforms Support

As of this writing, CSS Transforms are supported by every major browser without vendor prefixes; however, that support has happened more recently than Transitions, and there are a few more holes. Here are a few browser-related pointers:

- Internet Explorer 8 and earlier have no support for transforms. Version 9 supports Transforms with the **-ms-** prefix.
- IE 10 and 11 and all versions of Edge support transforms without prefixes, but they do not support transforms on elements in SVGs.
- Use the **-webkit-** prefix if you need to support the following browsers:
  - — Android v2.1 to 4.4.4 (prefixes dropped in 2017)
  - — OS Safari v3.2 to 8.4 (prefixes dropped in 2015)
  - — Safari 8 and earlier (prefixes dropped in 2015)
  - — Opera versions up to v.22 (prefixes dropped in 2014)

As of this writing, it is still recommended that you include **-ms-** and **-webkit-** prefixes for **transform**, but that may no longer be the case by the time you are reading this. Check *CanIUse.com* for updated browser information, and *ShouldIPrefix.com* for recommendations.

When an element transforms, its element box keeps its original position and influences the layout around it, in the same way that space is left behind by a relatively positioned element. It is as though the transformation magically picks up the pixels of the rendered element, messes around with them, and lays them back down on top of the page. So, if you move an element with **transform**, you're moving only a picture of it. That picture has no effect on the surrounding layout. Let's go through the transform functions one by one, starting with **rotate()**.

Transforming the Angle (rotate)

If you'd like an element to appear on a bit of an angle, use the **rotate()** transform function. The value of the **rotate()** function is an angle specified in positive (clockwise) or negative (counterclockwise) degrees. The image in FIGURE 18-7 has been rotated –10 degrees (350 degrees) with the following style rule. The tinted image shows the element's original position for reference.

```
img {
  width: 400px;
  height: 300px;
  transform: rotate(-10deg);
}
```

Notice that the image rotates around its center point, which is the default point around which all transformations happen. But you can change that easily with the **transform-origin** property.

transform: rotate(-10deg);

FIGURE 18-7. Rotating an **img** element by using **transform: rotate()**.

transform-origin

Values: *percentage* | *length* | `left` | `center` | `right` | `top` | `bottom`

Default: `50% 50%`

Applies to: transformable elements

Inherits: no

The value for **`transform-origin`** is either two keywords, length measurements, or percentage values. The first value is the horizontal offset, and the second is the vertical offset. If only one value is provided, it will be used for both. The syntax is the same as you learned for **`background-position`** back in **Chapter 13, Colors and Backgrounds**. If we wanted to rotate our image around a point at the center of its top edge, we could write it in any of the following ways:

```
transform-origin: center top;
transform-origin: 50%, 0%;
transform-origin: 200px, 0;
```

The images in FIGURE 18-8 have all been rotated 25 degrees, but from different origin points. It is easy to demonstrate the origin point with the **`rotate()`** function, but keep in mind that you can set an origin point for any of the transform functions.

`transform-origin: center top;`

`transform-origin: 100% 100%;`

`transform-origin: 400px 0;`

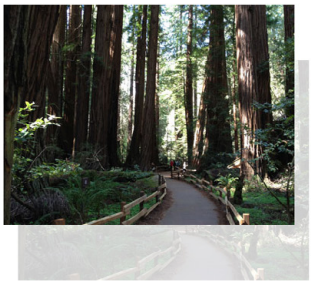
FIGURE 18-8. Changing the point around which the image rotates by using **`transform-origin`**.

Transforming the Position (translate)

Another thing you can do with the **`transform`** property is give the element's rendering a new location on the page by using one of three **`translate()`** functions, as shown in the examples in FIGURE 18-9. The **`translateX()`** function allows you to move an element on a horizontal axis; **`translateY()`** is for moving along the vertical axis; and **`translate()`** combines both x and y values.

```
transform: translateX(50px);
transform: translateY(25px);
transform: translate(50px, 25px); /* (translateX, translateY) */
```

`transform: translate(90px, 60px);`



`transform: translate(-5%, -25%);`

FIGURE 18-9. Moving an element around with the **`translate()`** function.

Provide length values in any of the CSS units or as a percentage value. Percentages are calculated on the width of the bounding box—that is, from border edge to border edge (which, incidentally, is how percentages are calculated in SVG, from which transforms were adapted). You can provide positive or negative values, as shown in FIGURE 18-9.

If you provide only one value for the shorthand **translate()** function, it will be presumed to be the **translateX** value, and **translateY** will be set to zero. So **translate(20px)** would be equivalent to applying both **translateX(20px)** and **translateY(0)**.

How do you like the **transform** property so far? We have two more functions to go.

Transforming the Size (scale)

Make an element appear larger or smaller by using one of three scale functions: **scaleX()** (horizontal), **scaleY()** (vertical), and the shorthand **scale()**. The value is a unitless number that specifies a size ratio. This example makes an image 150% its original width:

```
a img {
  transform: scaleX(1.5);
}
```

The **scale()** shorthand lists a value for **scaleX** and a value for **scaleY**. This example makes an element twice as wide but half as tall as the original:

```
a img {
  transform: scale(2, .5);
}
```

Unlike **translate()**, however, if you provide only one value for **scale()**, it will be used as the scaling factor in both directions. So specifying **scale(2)** is the same as applying **scaleX(2)** and **scaleY(2)**, which is intuitively the way you'd want it to be.

FIGURE 18-10 shows the results of all our scaling endeavors.

transform: scale(1.25);

transform: scale(.75);

transform: scale(1.5, .5);

FIGURE 18-10. Changing the size of an element with the **scale()** function.

Making It Slanty (skew)

The quirky collection of skew properties—**skewX()**, **skewY()**, and the shorthand **skew()**—changes the angle of either the horizontal or vertical axis (or both axes) by a specified number of degrees. As for **translate()**, if you provide only one value, it is used for **skewX()**, and **skewY()** will be set to zero.

The best way to get an idea of how skewing works is to take a look at some examples (FIGURE 18-11):

```
a img {
  transform: skewX(15deg);
}

a img {
  transform: skewY(30deg);
}

a img {
  transform: skew(15deg, 30deg);
}
```

transform: skewX(15deg);

transform: skewY(30deg);

transform: skew(15deg, 30deg);

FIGURE 18-11. Slanting an element by using the **skew()** function.

Applying Multiple Transforms

It is possible to apply more than one transform to a single element by listing out the functions and their values, separated by spaces, like this:

```
transform: function(value) function(value);
```

In the example in FIGURE 18-12, I've made the forest image get larger, tilt a little, and move down and to the right when the mouse is over it or when it is in focus:

```
img:hover, img:focus {
  transform: scale(1.5) rotate(-5deg) translate(50px,30px);
}
```

Normal state

:hover, **:focus**
rotate(), translate(), and scale() applied

FIGURE 18-12. Applying **scale()**, **rotate()**, and **translate()** to a single element.

It is important to note that transforms are applied in the order in which they are listed. For example, if you apply a **translate()** and then **rotate()**, you get a different result than with a **rotate()** and then a **translate()**. Order matters.

Another thing to watch out for is that if you want to apply an additional transform on a different state (such as **:hover**, **:focus**, or **:active**), you need to repeat all of the transforms already applied to the element. For example, this **a** element is rotated 45 degrees in its normal state. If I apply a **scale()** transform on the **hover** state, I would lose the rotation unless I explicitly declare it again:

```
a {
  transform: rotate(45deg);
}
a:hover {
  transform: scale(1.25);  /* rotate on a element would be lost */
}
```

To achieve both the rotation and the scale, provide both transform values:

```
a:hover {
  transform: rotate(45deg) scale(1.25);  /* rotates and scales */
}
```

Smoooooth Transforms

The multiple transforms applied to the redwood forest image look interesting, but it might *feel* better if we got there with a smooth animation instead of just BAM! Now that you know about transitions and transforms, let's put them together and make some magic happen. And by "magic," of course I mean some basic animation effects between two states. We'll do that together, step-by-step, in EXERCISE 18-2.

EXERCISE 18-2. Transitioning transforms

In this exercise, we'll make the travel photos in the gallery shown in FIGURE 18-13 grow and spin out to an angle when the user mouses over them—and we'll make it smoooooth with a transition. A starter document (*exercise_18-2.html*) and all of the images are available in the *materials* folder for this chapter.

1. Open *exercise_18-2.html* in a text editor, and you will see that there are already styles that arrange the list items horizontally and apply a slight drop shadow. The first thing we'll do is add the **transform** property for each image.
2. We want the transforms to take effect only when the mouse is over the image or when the image has focus, so the **transform** property should be applied to the **:hover** and **:focus** states. Because I want each image to tilt a little differently, we'll need to write a rule for each one, using its unique ID as the selector. You can save and check your work when you're done.

FIGURE 18-13. Photos get larger and tilt on **:hover** and **:focus** . A transition is used to help smooth out the change between states. You can see how it works when you are finished with this exercise (or check it out in the *ch18_figures.html* page).

```
a:hover #img1, a:focus #img1 {
  transform: rotate(-3deg);
}
a:hover #img2, a:focus #img2 {
  transform: rotate(5deg);
}
a:hover #img3, a:focus #img3 {
  transform: rotate(-7deg);
}
a:hover #img4, a:focus #img4 {
  transform: rotate(2deg);
}
```

NOTE

As of this writing, prefixes are still recommended for the **transform** *property, so for production-quality code, the complete rule would look like this:*

```
a:hover #img1, a:focus #img1 {
  -webkit-transform: rotate(-3deg);
  -ms-transform: rotate(-3deg); /* for IE9 */
  transform: rotate(-3deg);
}
```

Because we are checking our work on a modern browser, we can omit the prefixes for this exercise.

3. Now let's make the images a little larger as well, to give visitors a better view. Add **scale(1.5)** to each of the **transform** values. Here is the first one; you do the rest:

   ```
   a:hover #img1 {
     transform: rotate(-3deg) scale(1.5);
   }
   ```

   Note that my image files are created at the larger size and then scaled down for the thumbnail view. If we started with small images and scaled them larger, they would look crummy.

4. As long as we are giving the appearance of lifting the photos off the screen, let's make the drop shadow appear to be a little farther away by increasing the offset and blur, and lightening the shade of gray. All images should have the same effect, so add one rule using **a:hover img** as the selector.

   ```
   a:hover img {
     box-shadow: 6px 6px 6px rgba(0,0,0,.3);
   }
   ```

   Save your file and check it out in a browser. The images should tilt and look larger when you mouse over them. But the action is kind of jarring. Let's fix that with a transition.

5. Add the **transition** shorthand property to the normal **img** state (i.e., not on **:hover** or **:focus**). The property we want to transition in this case is **transform**. Set the duration to 0.3 seconds and use the **linear** timing function.

   ```
   img {
     …
     transition: transform 0.3s linear;
   }
   ```

NOTE

The prefixed **transform** *property should be included in the context of a transition as well, as shown in this fully prefixed declaration:*

```
-webkit-transition: -webkit-transform .3s linear;
```

The **-ms-** *prefix is not needed because transitions are not supported by IE9. Those users will see an immediate change to the transformed image without the smooth transition, which is fine.*

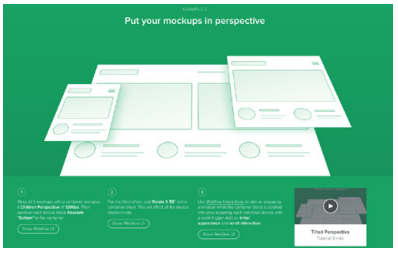
And that's all there is to it! You can try playing around with different durations and timing functions, or try altering the transforms or their origin points to see what other effects you can come up with.

3-D Transforms

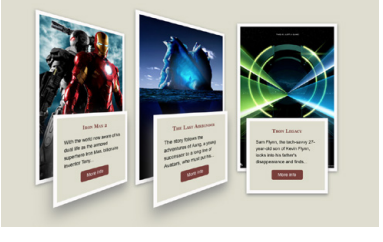
In addition to the two-dimensional transform functions we've just seen, the CSS Transforms spec also describes a system for creating a sense of three-dimensional space and perspective. Combined with transitions, you can use 3-D transforms to create rich interactive interfaces, such as image carousels, flippable cards, or spinning cubes! FIGURE 18-14 shows a few examples of interfaces created with 3-D transforms.

It's worth noting that this method does not create 3-D objects with a sense of volume; it merely tilts the otherwise flat element box around on three axes (animation expert Val Head calls them "postcards in space"). The rotating cube example in the figure merely stitches together six element boxes at different angles. That said, 3-D transforms still add some interesting depth to an otherwise flat web page.

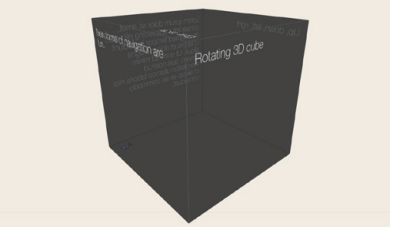
Animated book covers by Marco Barria
tympanus.net/Development/AnimatedBooks/



Webflow transform tools example
3d-transforms.webflow.com



Movie poster animation by Marco Kuiper
demo.marcofolio.net/3d_animation_css3/



3D CSS Rotating Cube by Paul Hayes
paulrhayes.com/experiments/cube-3d/

FIGURE 18-14. Some examples of 3-D transforms. The book covers, movie posters, and 3-D cube also have cool animation effects, so it's worth going to the links and checking them out. Webflow is a visual web design tool that includes the ability to create 3-D transformed elements.

3-D transforms are not a need-to-know skill for folks just starting out in web design, so I'm not going to go into full detail here, but I will give you a taste of what it takes to add a third dimension to a design. If you'd like to learn more, the following tutorials are good places to start (although the browser support information they contain may be out-of-date):

- "Adventures in the Third Dimension: CSS 3D Transforms" by Peter Gasston (*coding.smashingmagazine.com/2012/01/06/adventures-in-the-third-dimension-css-3-d-transforms/*)
- "Intro to CSS 3D Transforms" by David DeSandro (*desandro.github.com/3dtransforms/*)

To give you a very basic example, I'm going to use the images from EXERCISE 18-2 and arrange them as though they are in a 3-D carousel-style gallery (FIGURE 18-15).

FIGURE 18-15. Our aquarium images arranged in space…*space*…*space*…

The markup is the same unordered list used in the previous exercise:

```
<ul>
  <li><a href=""><img src="anchovies.jpg" id="img1" alt=""></a></li>
  <li><a href=""><img src="jellyfish1.jpg" id="img2" alt=""></a></li>
  <li><a href=""><img src="bluejellyfish.jpg" id="img3" alt=""></a>
  </li>
  <li><a href=""><img src="seadragon.jpg" id="img4" alt=""></a></li>
</ul>
```

The first step is to add some amount of "perspective" to the containing element by using the **perspective** property. This tells the browser that the child elements should behave as though they are in 3-D space. The value of the **perspective** property is some integer larger than zero that specifies a distance from the element's origin on the z-axis. The lower the value, the more extreme the perspective. I have found that values between 300 and 1,500 are reasonable, but this is something you need to fuss around with until you get the desired effect.

```
ul {
  width: 1000px;
  height: 100px;
  list-style-type: none;
  padding: 0;
  margin: 0;
  perspective: 600;
}
```

NOTE

When using the **-webkit-** *prefix for* **transform**, *include the prefixed version of* **perspective** *as well (*-webkit-perspective*).*

The **perspective-origin** property (not shown) describes the position of your eyes relative to the transformed items. The values are a horizontal position (**left**, **center**, **right**, or a length or percentage) and a vertical position (**top**, **bottom**, **center**, or a length or percentage value). The default (FIGURE 18-15) is centered vertically and horizontally (**perspective-origin: 50% 50%**). The final transform-related property is **backface-visibility**, which controls whether the reverse side of the element is visible when it spins around.

With the 3-D space established, apply one of the 3-D transform functions to each child element—in this case, the **li** within the **ul**. The 3-D functions include **translate3d**, **translateZ**, **scale3d**, **scaleZ**, **rotate3d**, **rotateX**, **rotateY**, **rotateZ**, and **matrix3d**. You should recognize some terms in there. The ***Z** functions define the object's orientation relative to the z-axis (picture it running from your nose to this page, whereas the x- and y-axes lie flat on the page).

In our example in FIGURE 18-15, each **li** is rotated 45 degrees around its y-axis (vertical axis) by using the **rotateY** function, which works as though the element boxes are rotating around a pole.

Compare the result to FIGURE 18-16, in which each **li** is rotated on its x-axis (horizontal axis) by using **rotateX**. It's as though the element boxes are rotating around a horizontal bar.

```
li {
  float: left;
  margin-right: 10px;
  transform: rotateX(45deg);
}
```

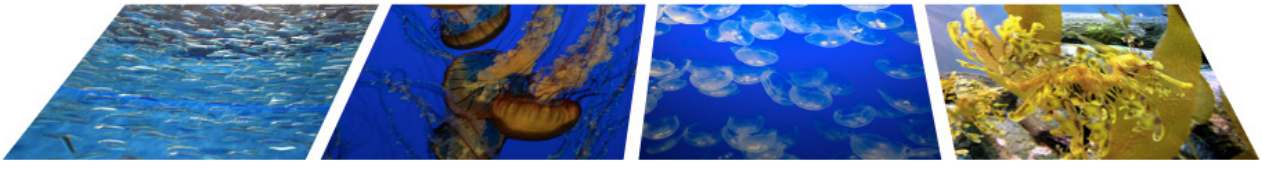


FIGURE 18-16. The same list of images rotated on their horizontal axes with **rotateX()**.

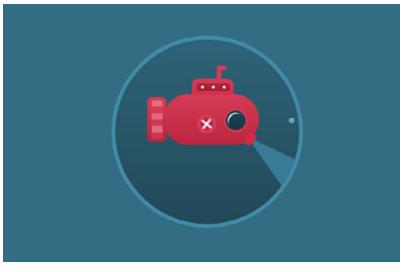
Obviously, I'm barely scratching the surface of what can be done with 3-D transforms, but this should give you a mental model for how it works. Next up, I'll introduce you to a more sophisticated way to set your web pages in motion.

KEYFRAME ANIMATION

The CSS Animations Module allows authors to create real, honest-to-goodness keyframe animation. FIGURE 18-17 shows just a few examples that you can see in action online. Unlike transitions that go from a beginning state to

MADMANIMATION
by Anthony Calzadilla and Andy Clarke
stuffandnonsense.co.uk/content/demo/madmanimation/—hint: click WATCH

How I Learned to Walk
by Andrew Wang-Hoyer
andrew.wang-hoyer.com/experiments/walking/



Adorable animated submarine
by Alberto Jerez
codepen.io/ajerez/pen/EaEEOW

Animated Web Banner
by Caleb Jacob
tympanus.net/codrops/2012/01/10/animated-web-banners-with-css3/

CSS transitions are animations with two keyframes: a start state and an end state. More complex animations require many keyframes to control property changes in the sequence.

FIGURE 18-17. Examples of animations using only CSS.

an end state, keyframe animation allows you to explicitly specify other states at points along the way, allowing for more granular control of the action. Those "points along the way" are established by keyframes that define the beginning or end of a segment of animation.

Creating keyframe animations is complex, and more than I can cover here. But I would like for you to have some idea of how it works, so I'll sketch out the minimal details. The following resources are good starting points for learning more:

- *CSS Animations Level 1 (a Working Draft at the time of this writing) at www.w3.org/TR/css-animations-1/.*
- *Transitions and Animations in CSS* by Estelle Weyl (O'Reilly).
- "Animation & UX Resources" by Val Head (*valhead.com/ui-animation/*). Val has compiled a mega-list of resources regarding web animation, including links to tutorials, articles, tools, galleries, and more. It is not limited to CSS keyframe animation, but as long as you're delving into animation, you can trust Val to point you to good stuff.

NOTE

Keyframe animation is known as explicit animation *because you program its behavior. By contrast, transitions are an example of* implicit animation *because they are triggered only when a property changes.*

Animation Tools

If you want to add a simple animation effect to an element—a quick flip here or a little shimmy there—you may be able to find a premade effect you can apply to your design. Here are a few sites that provide ready-made CSS for common animation effects (some also use JQuery plug-ins, but they explain how to use them):

- Animate.css by Daniel Eden (*daneden.github.io/animate.css/*)
- CSS Animation Cheat Sheet by Justin Aguilar (*www.justinaguilar.com/animations/index.html*)
- AngryTools CSS Animation Kit (*angrytools.com/css/animation/*)

FIGURE 18-18. Animating through the colors of the rainbow by using keyframes.

- "CSS: Animation" course by Val Head on Lynda.com (*www.lynda.com/CSS-tutorials/CSS-Animation/439683-2.html*). You'll need a subscription to Lynda.com, but if you are in web-design-learning mode, it may be a good investment.
- "CSS Animation for Beginners" by Rachel Cope (*robots.thoughtbot.com/css-animation-for-beginners*). This is a clearly written tutorial with lots of examples.
- "The Guide to CSS Animation: Principles and Examples" by Tom Waterhouse (*www.smashingmagazine.com/2011/09/the-guide-to-css-animation-principles-and-examples/*). This tutorial goes beyond CSS code to include tips for creating natural animation effects.

Establishing the Keyframes

The animation process has two parts:

1. Establish the keyframes with a **`@keyframes`** rule.
2. Add the animation properties to the elements that will be animated.

Here is a very simple set of keyframes that changes the background color of an element over time. It's not a very action-packed animation, but it should give you a basic understanding of what a **`@keyframes`** rule does.

```
@keyframes colors {
  0% { background-color: red; }
  20% { background-color: orange; }
  40% { background-color: yellow; }
  60% { background-color: green; }
  80% { background-color: blue; }
  100% { background-color: purple; }
}
```

The keyframes at-rule identifies the name of the animation, the stages of the animation represented by percentage (%) values, and the CSS properties that are affected for each stage. Here's what a **`@keyframes`** rule looks like abstracted down to its syntax:

```
@keyframes animation-name {
  keyframe { property: value; }
  /* additional keyframes */
}
```

The sample **`@keyframes`** rule says: create an animation sequence called "colors." At the beginning of the animation, the **`background-color`** of the element should be red; at 20% through the animation runtime, the background color should be orange; and so on, until it reaches the end of the animation. The browser fills in all the shades of color in between each keyframe (or *tweens* it, to use the lingo). This is represented the best I could in FIGURE 18-18.

Each percentage value and the property/value declaration defines a keyframe in the animation sequence.

As an alternative to percentages, you can use the keyword **from** for the start of an animation sequence (equivalent to 0%) and the keyword **to** for denoting the end (100%). The following example makes an element slide in from right to left as the left margin reduces to 0:

```
@keyframe slide {
  from { margin-left: 100% }
  to { margin-left: 0%; }
}
```

Adding Animation Properties

Now we can apply this animation sequence to an element or multiple elements in the document by using a collection of animation properties that are very similar to the set of transition properties that you already know.

I am going to apply the rainbow animation to the **#magic div** in my document:

```
<div id="magic">Magic!</div>
```

In the CSS rule for **#magic**, I make decisions about the animation I want to apply:

- Which animation to use (**animation-name**) (*Required*).
- How long it should take (**animation-duration**) (*Required*).
- The manner in which it should accelerate (**animation-timing-function**). This property uses the same timing function keywords that we covered for CSS Transitions.
- Whether to pause before it starts (**animation-delay**).

Looks familiar, right? There are a few other animation-specific properties to know about as well:

animation-iteration-count

How many times the animation should repeat. This can be set to a whole number or **infinite**.

animation-direction

Whether the animation plays forward (**normal**), in reverse (**reverse**), or alternates back and forth starting at the beginning (**alternate**), or alternates starting from the end (**alternate-reverse**).

animation-fill-mode

The animation fill mode determines what happens with the animation before it begins and after it ends. By default (**none**), the animation shows whatever property values were not specified via **@keyframes**. If you want the last keyframe to stay visible after the animation plays, use the **forwards** keyword. If there is a delay set on the animation and you want the first keyframe to show during that delay, use **backwards**. To retain the beginning and end states, use **both**.

■ SUPPORT TIP

CSS Keyframe Browser Support

All current versions of major desktop and mobile browsers support CSS keyframe animation without vendor prefixes. Here are the exceptions:

- Internet Explorer 9 and earlier do not support keyframe animation at all. The animation will appear in its start state, so be sure that first frame is an acceptable fallback.
- You need to use the **-webkit-** prefix to support the following browsers: Safari and iOS Safari 8 and earlier (2014), Chrome 41 and earlier (2015), Opera 29 and earlier (2015), and Android 4.4.4 and earlier (2014). As I am writing this, these browsers represent enough traffic that it is still recommended that you include the **-webkit-** prefix, but that may change based on when you are doing development and who your target audience is.

  Note that you need the prefixed keyframe at-rule as well as prefixed **animation-*** properties. As always, the standard, unprefixed rules go after prefixed versions.

  `@-webkit-keyframes`

  `-webkit-animation-*`

animation-play-state

Whether the animation should be **running** or **paused** when it loads. The play-state can be toggled on and off based on user input with JavaScript or on hover.

The **animation-name** property tells the browser which keyframe sequence to apply to the **#magic div**. I've also set the duration and timing function, and used **animation-iteration-count** to make it repeat infinitely. I could have provided a specific number value, like 2 to play it twice, but how fun are only two rainbows? And for fun, I've set the **animation-direction** to **alternate**, which makes the animation play in reverse after it has played forward. Here is the resulting rule for the animated **div**:

```
#magic {
  …
  animation-name: colors;
  animation-duration: 5s;
  animation-timing-function: linear;
  animation-iteration-count: infinite;
  animation-direction: alternate;
}
```

That gets a bit verbose, especially when you consider that each property may also follow a prefixed version. You can also use the **animation** shorthand property to combine the values, just as we did for **transition**:

```
#magic {
  animation: colors 5s linear infinite alternate;
}
```

Those are the bare bones of creating keyframes and applying animations to an element on the page. To make elements move around (what we typically think of as "animation"), use keyframes to change the position of an element on the screen with **translate** (the best option for performance) or with the **top**, **right**, **bottom**, and **left** properties. When the keyframes are tweened, the object will move smoothly from position to position. You can also animate the other transform functions such as **scale** and **skew**.

When to Use Keyframe Animation

To keep my example simple, I chose to change only the background color of a button element, but of course, keyframe animations can be used to create real animations, especially when combined with the CSS transform functions for spinning and moving elements around on the page. If you only need to change an element from one state to another, a transition is the way to go. But if you have a linear animation such as moving a character, an object, or its parts around, keyframe animation is the most appropriate choice.

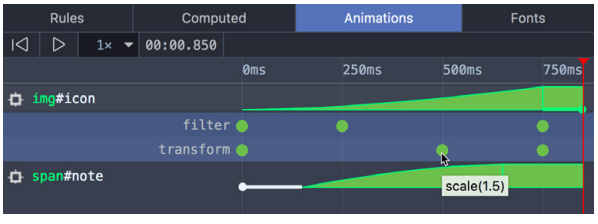
For more complex keyframe animations, particularly those that change with user interaction or require complex physics, using JavaScript for animation may be a better choice than CSS animation. JavaScript animation also has

Animation Inspectors

Both Chrome and Firefox offer tools to inspect and modify web animations (FIGURE 18-19). When you inspect an animated element in the Developer Tools, click the Animations tab to see a timeline of all the animations applied to that object. You can slow down the animation to reveal what is happening on a detailed level. You can also modify the animation by making changes to the timing, delay, duration, and keyframes. For more information, see the following:

- Firefox Animation Inspector (*developer.mozilla.org/en-US/docs/Tools/Page_Inspector/How_to/Work_with_animations*)
- Chrome Animation Inspector (*developers.google.com/web/tools/chrome-devtools/inspect-styles/animations*)

Firefox Animation Inspector



Chrome Animation Inspector

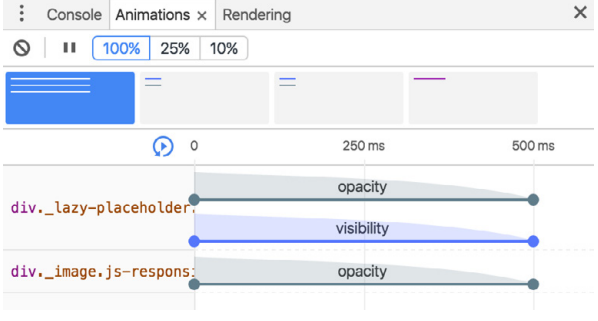


FIGURE 18-19. Animation inspectors are part of the developer tools offered by Firefox and Chrome browsers.

better support in older browsers, making it preferable if animation is critical to the mission of the page. CSS keyframe animation is a good solution for simple animations used as an enhancement to a baseline experience.

I should note that as I write this, there is a lot of excitement in the web community for animating SVG graphics. When you place the source code for an SVG directly in the HTML document, the elements in it are available to be animated. As of this writing, there are still limitations and browser support issues around using CSS to animate SVGs, but as browser support grows, this approach looks very promising. In the meantime, JavaScript has better access to SVG properties, has better browser support, and is the more common solution for SVG animation.

WRAPPING UP

I hope I've helped you to wrap your head around how CSS can be used to add a little motion and smoothness to your pages. For adding motion to a web page, we have CSS Transitions to smooth out changes from one state to another and CSS Keyframe Animation for animating a series of states. We also looked at CSS Transforms for repositioning, spinning, resizing, or skewing an element when it is rendered on the screen.

Used thoughtfully, animation can make your interfaces more intuitive and enhance your brand personality. It's powerful stuff, but with great power comes great responsibility. To learn how to use web animation to enhance the user experience in a meaningful way, I recommend the book *Designing Interface Animation: Meaningful Motion for User Experience* by Val Head (Rosenfeld Media).

Now let's see if you were paying attention with a 12-question quiz!

Need a Little Inspiration?

The Codrops blog (*tympanus.net/codrops/*), curated by Manoela Ilic and Pedro Botelho, is a treasure trove of examples of CSS transitions, transforms, and animations. Check out the Playground for cool experiments (like the collection of hover effects in FIGURE 18-20) and the Tutorials section for step-by-step how-to information with code examples.

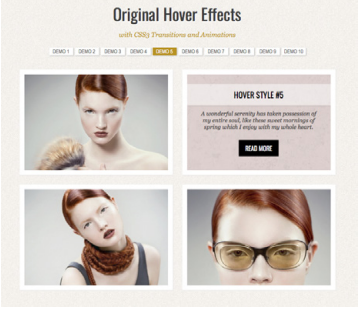


FIGURE 18-20. One of many examples of CSS transitions, transforms, and animations at the Codrops blog.

TEST YOURSELF

Think you know your way around transitions, transforms, and keyframe animations? Here are a few questions to find out (answers in **Appendix A**):

1. What is tweening?

2. If a transition had keyframes, how many would it have?

3. Write out the transition declaration (property and value) you would use to accomplish the following:
   a. Wait .5 seconds before the transition starts.
   b. Make the transition happen at a constant speed.
   c. Make the transition last .5 seconds.
   d. Make the lines of text slowly grow farther apart.

4. Which of the following can you *not* animate?
   a. `width`
   b. `padding`
   c. `text-transform`
   d. `word-spacing`

5. Which timing function will be used if you omit the **`transition-timing-function`** property? Describe its action.

6. In the following transition, what does .2s describe?
   ```
   transition: color .2s linear;
   ```

7. Which transition will finish first?
   a. `transition: width 300ms ease-in;`
   b. `transition: width 300ms ease-out;`

8. Write the **`transform`** declaration to accomplish the following:
   a. Tilt the element 7 degrees clockwise.
   b. Reposition the element 25 pixels up and 50 pixels to the left.
   c. Rotate the element from its bottom-right corner.
   d. Make a 400-pixel-wide image display at 500 pixels wide.

9. In the following transform declaration, what does the 3 value describe?
   ```
   transform: scale(2, 3)
   ```

10. Which 3-D transform would look more angled and dramatic?

    a. `perspective: 250;`

    b. `perspective: 1250;`

11. What happens halfway through this animation?

```
@keyframes border-bulge {
  from { border-width: 1px; }
  25% { border-width: 10px; }
  50% { border-width: 3px; }
  to { border-width: 5px; }
}
```

12. Write the animation declaration you would use to accomplish the following:

    a. Make the animation play in reverse.

    b. Make the entire animation last 5 seconds.

    c. Wait 2 seconds before running the animation.

    d. Repeat the animation three times and then stop.

    e. The end state of the animation stays visible after the animation is done playing.

CSS REVIEW: TRANSITIONS, TRANSFORMS, AND ANIMATION

Here is a summary of the properties covered in this chapter.

Property	Description
`animation`	A shorthand property that combines animation properties
`animation-name`	Specifies the named animation sequence to apply
`animation-duration`	Specifies the amount of time the animation lasts
`animation-timing-function`	Describes the acceleration of the animation
`animation-iteration-count`	Indicates the number of times the animation repeats
`animation-direction`	Specifies whether the animation plays forward, in reverse, or alternates back and forth
`animation-play-state`	Specifies whether the animation is running or paused
`animation-delay`	Indicates the amount of time before the animation starts running
`animation-fill-mode`	Overrides limits to when animation properties can be applied
`backface-visibility`	Determines whether the reverse side of an element may be visible in 3-D transforms
`perspective`	Establishes an element as a 3-D space and specifies the perceived depth
`perspective-origin`	Specifies the position of your viewpoint in a 3-D space
`transform`	Specifies that the rendering of an element should be altered via one of the 2-D or 3-D transform functions
`transform-origin`	Denotes the point around which an element is transformed
`transform-style`	Preserves a 3-D context when transformed elements are nested
`transition`	A shorthand property that combines transition properties
`transition-property`	Defines which CSS property will be transitioned
`transition-duration`	Specifies the amount of time the transition animation lasts
`transition-timing-function`	Describes the manner in which the transition happens (changes in acceleration rates)
`transition-delay`	Specifies the amount of time before the transition starts

CHAPTER
19

MORE CSS TECHNIQUES

IN THIS CHAPTER

Styling forms

Style properties for tables

Using a CSS reset or normalizer

Image replacement techniques

CSS sprites

CSS feature detection

By now you have a solid foundation in writing style sheets. You can style text and element boxes, float and position objects, create responsive page layouts using Flexbox and Grid, and even add subtle animation effects to your designs. But there are a few more common techniques that you should know about.

If you look over at the **"In This Chapter"** list, you'll see that this chapter is a grab bag of sorts. It starts with general approaches to styling forms and the special properties for table formatting. We'll cover other tricks of the trade including clearing out browser styles with a CSS reset, using images in place of text (only when necessary!), reducing the number of server requests with CSS sprites, and checking whether a browser supports a particular CSS feature. Let's dig in!

STYLING FORMS

Web forms can look a bit hodgepodge right out of the box with no styles applied (FIGURE 19-1), so you'll certainly want to give them a more professional appearance using CSS. Not only do they look better, but studies show that forms are much easier and faster to use when the labels and inputs are lined up nicely. In this section, we'll look at how various form elements can be styled.

Now, I'm not going to lie: styling forms is somewhat of a dark art because of the variety of ways in which browsers handle form elements. And for really slick, custom forms, you will generally need to turn to JavaScript. But the efforts are well worth it to improve usability.

There aren't any special CSS properties for styling forms; just use the standard color, background, font, border, margin, and padding properties that

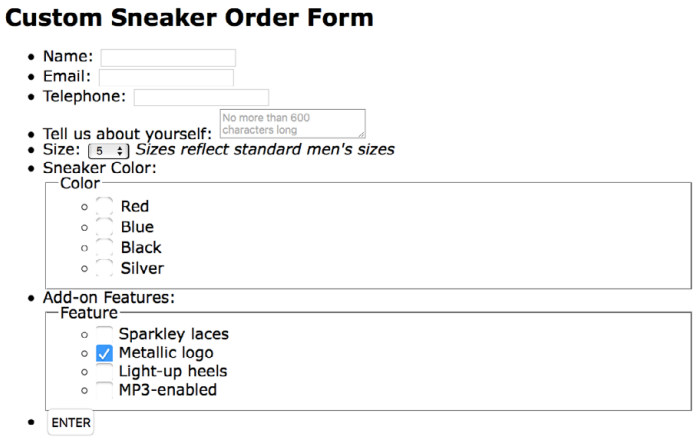


FIGURE 19-1. Forms tend to be ugly and difficult to use with HTML alone. Don't worry—this one gets spiffed up in FIGURE 19-2.

you've learned in the previous chapters. The following is a quick rundown of the types of things you can do for each form control type.

***Text inputs* (`text`, `password`, `email`, `search`, `tel`, `url`)**

Change the appearance of the box itself with **`width`**, **`height`**, **`background-color`**, **`background-image`**, **`border`**, **`border-radius`**, **`margin`**, **`padding`**, and **`box-shadow`**. You can also style the text inside the entry field with the **`color`** property and the various font properties.

The* `textarea` *element

This can be styled in the same way as text-entry fields. **`textarea`** elements use a monospace font by default, so you may want to change that to match your other text-entry fields. Because there are multiple lines, you may also specify the line height. Note that some browsers display a handle on the lower-right corner of the **`textarea`** box that makes it resizable, but you can turn it off by adding the style **`resize: none`**. Text areas display as **`inline-block`** by default, but you can change them to **`block`** with the **`display`** property.

***Button inputs* (`submit`, `reset`, `button`)**

Apply any of the box properties to submit and reset buttons (**`width`**, **`height`**, **`border`**, **`background`**, **`margin`**, **`padding`**, and **`box-shadow`**). It is worth noting that buttons are set to the border-box sizing model by default. Most browsers also add a bit of padding by default, which can be overridden by your own padding value. You can also style the text that appears on the buttons.

Radio and checkbox buttons

The best practice for radio and checkbox buttons is to leave them alone. If you are tenacious, you can use JavaScript to change the buttons altogether.

Coordinated Fonts

By default, a browser may use different fonts at different sizes for various input types. For example, it may use the system font on buttons and a constant-width font like Courier for **`textarea`** inputs. If you'd like all inputs to use the same font as the surrounding text on the page, you can force the form elements to inherit font settings:

```
button, input, select,
textarea {
  font-family: inherit;
  font-size: 100%
}
```

Drop-down and select menus (`select`)

You can specify the width and height for a **`select`** element, but note that it uses the border-box box-sizing model by default. Some browsers allow you to apply **`color`**, **`background-color`**, and font properties to **`option`** elements, but it's probably best to leave them alone to be rendered by the browser and operating system.

Fieldsets and legends

You can treat a **`fieldset`** as any other element box, adjusting the border, background, margin, and padding. Turning the border off entirely is one way to keep your form looking tidy while preserving semantics and accessibility. By default, **`legend`** elements are above the top border of the **`fieldset`**, and, unfortunately, browsers make them very difficult to change. Some developers use a **`span`** or **`b`** element within the **`legend`** and apply styles to the contained element for more predictable results. Some choose to hide it in a way that it will still be read by screen readers (**`legend {width: 1px; height: 1px; overflow: hidden;}`**).

Now we know what we can do to style individual controls, but the grander goal is to make the form more organized and easier to use. FIGURE 19-2 shows the "after" shots of the unstyled form from FIGURE 19-1. There color, font, border, and spacing changes, and the labels and input elements are nicely aligned as well. And not only that, the form is responsive! I've used Flexbox to

HEADS UP

The following form elements cannot be changed with CSS alone: inputs for range, color, date pickers, file picker, **`option`**, **`optgroup`**, **`datalist`**, **`progress`**, *and* **`meter`**. *It is possible to customize them by using JavaScript, which is beyond the scope of this book.*

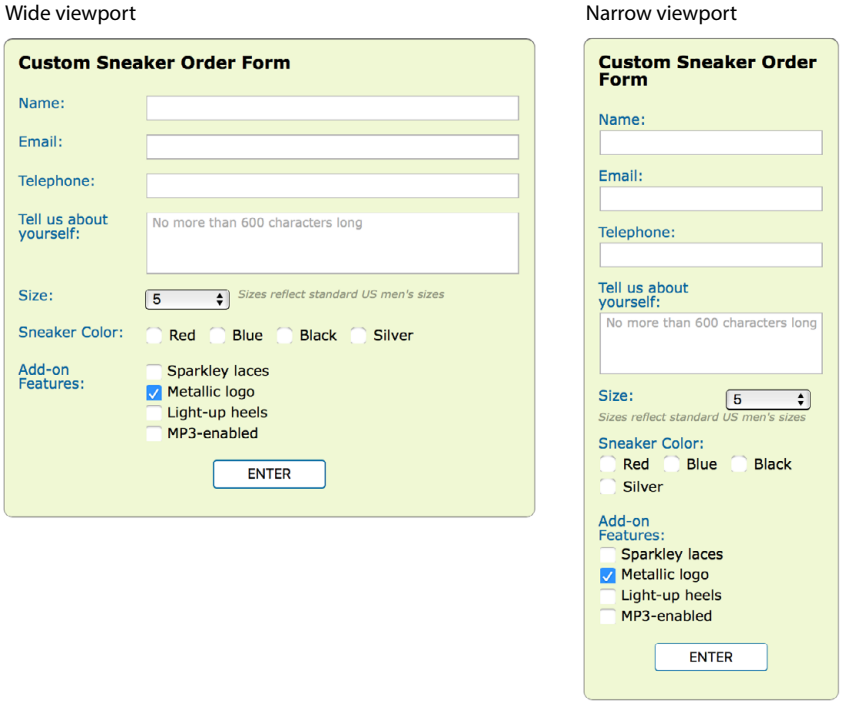


FIGURE 19-2. This responsive form uses Flexbox to allow text inputs to resize and to shift the position of the labels on small screens.

make the labels stack on top of their respective inputs and fieldsets on narrow screens so there is no wasted space.

If you'd like to take a look at the actual markup and styles, the document *sneakerform.html* is available with the materials for this chapter (*learningwebdesign.com/5e/materials*). I've left careful and thorough comments throughout that explain exactly what each style is for. My approach to styling the Custom Sneaker Order Form can be summarized as follows:

- Set the `box-sizing` to `border-box` for the whole document. This makes sizing form elements more predictable.
- Give the `form` a `max-width` (so it can shrink to fit smaller viewports) and optional decorative styling like the green background and rounded border in the example.
- Get rid of the bullets and spacing around the unordered lists that were used to mark up the form semantically.
- Turn list items (each containing a label and some sort of input or fieldset) into flex containers by setting their `display` to `flex` (see **Note**). Turn on wrapping, which is what allows the input to shift below the labels on small screens.
- Give the labels fixed widths (`flex: 0 0 8em;`) so they are sized the same regardless of screen size. Because labels on checkboxes and radio buttons work differently, set them to override the 8em width (`flex: 1 1 auto;`).
- Allow the `input`, `textarea`, and `fieldset`s to grow to fill the remaining space (`flex: 1 1 20em;`). When the screen is too narrow for them to fit next to the labels, they wrap below.
- Set the text input fields' `font-family` to `inherit` so they use the same font as the rest of the document instead of whatever font the browser uses for forms. Text inputs also get heights, borders, and a little padding.
- Fieldsets and legends are tricky to style. Turn off the border and padding on the `fieldset`, and then hide the `legend` in a way that it will still be read aloud before each checkbox or radio button option. Because there is both a label and a legend for each fieldset, I made sure they are not exactly the same so they won't be redundant when read aloud by a screen reader. The legend should be shorter because it is repeated for each option.
- The submit button has a rounded border, background color, and font styling. Set the side margins to `auto` so it will always be centered in the width of the form.

This is a very simple example, but it should give you a general idea of how forms can be styled. You may also want to add highlight styles for interactivity, such as `:hover` styles on the buttons and `:focus` styles for text inputs when they are selected.

NOTE

If you don't want to use Flexbox, you can line up labels by using floats. Set labels to `display: block`*, give them a width and height, and float them to the left. You need to clear the* `li` *elements (*`clear: both`*) so they start below the previous floated pair.*

STYLING TABLES

Like any other text content on a web page, content within table cells can be formatted with various font, text, and background properties.

You will probably want to adjust the spacing in and around tables. To adjust the amount of space within a cell (cell padding), apply the **padding** property to the **td** or **th** element. Spacing between cells (cell spacing) is a little more complicated and is related to how CSS handles cell borders. CSS provides two methods for displaying borders between table cells: separated or collapsed. These options are specified with the table-specific **border-collapse** property with **separate** and **collapse** values, respectively.

border-collapse

Values: separate | collapse

Default: separate

Applies to: table and inline-table elements

Inherits: yes

Separated Borders

By default, borders are separated, and a border is drawn on all four sides of each cell. The **border-spacing** property lets you specify the space between cell borders.

border-spacing

Values: *horizontal-length vertical-length*

Default: 0

Applies to: table and inline-table elements

Inherits: yes

The values for **border-spacing** are two length measurements. The horizontal value comes first and applies between columns. The second measurement is applied between rows. If you provide one value, it will be used both horizontally and vertically. The default setting is 0, causing the borders to double up on the inside grid of the table (see **Note**).

The table in FIGURE 19-3 is set to **separate** with 15 pixels of space between columns and 5 pixels of space between rows. A purple border has been applied to the cells to make their boundaries clear.

NOTE

Although the **border-spacing** *default is 0, browsers generally add 2 pixels of space for the obsolete* **cellspacing** *attribute by default. If you want to see the borders double up, you need to set the* **cellspacing** *attribute to 0 in the* **table** *element.*

NOTE

In the past, cell padding and spacing were handled by the **cellpadding** *and* **cellspacing** *attributes in the* **table** *element, respectively, but they have been made obsolete in HTML5 because of their presentational nature.*

```
td {
  border: 3px solid purple;
}
table {
  border-collapse: separate;
  border-spacing: 15px 5px;
  border: none;
}
```

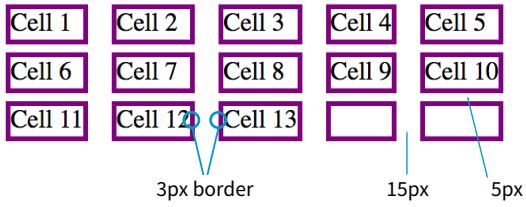


FIGURE 19-3. The separated table border model.

For tables with separated borders, you can indicate whether you want empty cells to display their backgrounds and borders by using the **empty-cells** property. For a cell to be "empty," it may not contain any text, images, or non-breaking spaces. It may contain carriage returns and space characters.

empty-cells

Values: show | hide

Default: show

Applies to: table cell elements

Inherits: yes

FIGURE 19-4 shows the previous separated table-border example with its empty cells (what would be Cell 14 and Cell 15) set to **hide**.

```
empty-cells: hide;
```

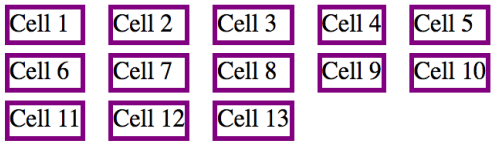


FIGURE 19-4. Hiding empty cells with the **empty-cells** property.

Collapsed Borders

In the collapsed border model, the borders of adjacent borders "collapse" so that only one of the borders is visible and the space is removed (FIGURE 19-5). In the example, although each table cell has a 3-pixel border, the borders between cells measure a total of 3 pixels, not 6. In instances where neighboring cells have different border styles, a complicated pecking order is called in to determine which border will display, which you can read in the spec.

The advantage to using the collapsed table-border model is that you can style the borders for **tr**, **col**, **rowgroup**, and **colgroup** elements. With the separated model, you can't. Strategic use of horizontal and vertical borders improves the readability of complicated tables, making the collapsed model an attractive choice.

```
td {
  border: 3px solid purple;
}
table {
  border-collapse: collapse;
  border: none;
}
```

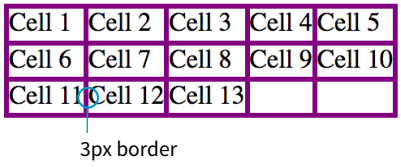


FIGURE 19-5. The collapsed border model.

Table Layout

table-layout

Values: auto | fixed

Default: auto

Applies to: table or inline-table elements

Inherits: yes

The **table-layout** property allows authors to specify one of two methods of calculating the width of a table. The **fixed** value bases the table width on **width** values provided for the table, columns, or cells. The **auto** value bases the width of the table on the minimum width of the contents of the table. Auto layout may display nominally more slowly because the browser must calculate the default width of every cell before arriving at the width of the table.

That covers basic form and table formatting. I know this is a beginner's book, but in the next section, I'm going to introduce you to a few intermediate CSS techniques that may make your work easier and your pages faster.

Table Display Properties

CSS 2.1 includes a number of values for the **display** property that allow authors to attach table display behaviors to elements. The table-related **display** values are **table**, **inline-table**, **table-row-group**, **table-header-group**, **table-footer-group**, **table-row**, **table-column-group**, **table-column**, **table-cell**, and **table-caption**.

The original intent for these values was to provide a mechanism for applying table display behaviors to XML languages that may not have elements like **table**, **tr**, or **td** in their vocabularies.

In recent years, table display values have become another method for achieving page layout effects such as vertical centering and flexible column widths. CSS table layout may be useful as a fallback design for older browsers that do not support CSS Grid or Flexbox. Note that this is not the same as using table-based layout with HTML markup. With CSS table layout, the semantics of the source document stay intact. If you'd like to learn more, I recommend the article "Layout Secret #1: The CSS Table Property" by Massimo Cassandro (*www.sitepoint.com/solving-layout-problems-css-table-property/*).

Now that Flexbox and Grid are gaining momentum, I suspect the table layout methods will eventually go by the wayside.

Pick a Side

When you use the **caption** element in a table, it will appear above the table by default. If you'd prefer it to be below the table, you can use the **caption-side** property to position it there.

caption-side

Values: top | bottom

Default: top

Applies to: table caption element

Inherits: yes

A CLEAN SLATE (RESET AND NORMALIZE.CSS)

As you know, browsers have their own built-in style sheets (called user agent style sheets) for rendering HTML elements. If you don't supply styles for an `h1`, you can be certain that it will display as large, bold text with space above and below. But just how much larger and how much space may vary from browser to browser, giving inconsistent results. Furthermore, even if you do provide your own style sheet, elements in your document may be secretly inheriting certain styles from the user agent style sheets, causing unexpected results.

There are two methods for getting a consistent starting point for applying your own styles: a CSS reset or normalize.css. They take different approaches, so one or the other may be the best solution for what you need to achieve.

CSS Reset

The older approach is a CSS reset, a collection of style rules that overrides *all* user agent styles and creates a starting point that is as neutral as possible. With this method, you need to specify all the font and spacing properties for every element you use. It's a truly from-scratch starting point.

The most popular reset was written by Eric Meyer (the author of too many CSS books to list). It is presented here, and I've also included a copy of it in the *materials* folder for this chapter for your copy-and-paste pleasure (see **Note**). If you look through the code, you'll see that the margins, border, and padding have been set to 0 for a long list of block elements. There are also styles that get typography to a neutral starting point, clear out styles on lists, and prevent browsers from adding quotation marks to quotes and blockquotes.

NOTE

You can get the CSS reset on the web at meyerweb.com/eric/tools/css/reset/.

```
/* http://meyerweb.com/eric/tools/css/reset/
 v2.0 | 20110126 License: none (public domain)*/
html, body, div, span, applet, object, iframe,
h1, h2, h3, h4, h5, h6, p, blockquote, pre,
a, abbr, acronym, address, big, cite, code,
del, dfn, em, img, ins, kbd, q, s, samp,
small, strike, strong, sub, sup, tt, var,
b, u, i, center, dl, dt, dd, ol, ul, li,
fieldset, form, label, legend,
table, caption, tbody, tfoot, thead, tr, th, td,
article, aside, canvas, details, embed,figure, figcaption, footer,
header, hgroup,menu, nav, output, ruby, section, summary,
time, mark, audio, video {
    margin: 0;
    padding: 0;
    border: 0;
    font-size: 100%;
    font: inherit;
    vertical-align: baseline;
}
```

```
/* HTML5 display-role reset for older browsers */
article, aside, details, figcaption, figure,footer, header, hgroup,
menu, nav, section {
    display: block;
}
body {
    line-height: 1;
}
ol, ul {
    list-style: none;
}
blockquote, q {
    quotes: none;
}
blockquote:before, blockquote:after,
 q:before, q:after {
    content: '';
    content: none;
}
table {
    border-collapse: collapse;
    border-spacing: 0;
}
```

To use the reset, place these styles at the top of your own style sheet so your own styles override them. You can use them exactly as you see them here or customize them as your project requires. I also recommend reading Eric's posts about the thinking that went into his settings at *meyerweb.com/eric/tools/css/reset/* and *meyerweb.com/eric/thoughts/2007/04/18/reset-reasoning/*. A web search will reveal other, potentially smaller, CSS reset options.

Normalize.css

A more nuanced approach is to use Normalize.css, created by Nicolas Gallagher and Jonathan Neal. They painstakingly combed through the user agent styles of every modern browser (desktop and mobile) and created a style sheet that tweaks their styles for consistency, rather than just turning them all off. Normalize.css gives you a reasonable starting point: paragraphs still have some space above and below, headings are bold in descending sizes, lists have markers and indents as you would expect. It also includes styles that make form widgets consistent, which is a nice service. FIGURE 19-6 shows the difference between CSS reset and Normalize.css starting points.

You can download Normalize.css at *necolas.github.io/normalize.css/* and include it before your own styles. It is too long to print here, but you will find that it is well organized and includes comments with clear explanations for each section. For Nicolas's thoughts on the project, see *nicolasgallagher.com/about-normalize-css/*.

Normalize.css is considered a superior successor to the cruder CSS reset, but I think it is important to be aware of both options. Or, if slight differences from browser to browser are just fine with you (as they are for a lot of professional developers), you don't need to use either.

FIGURE 19-6. The difference between CSS reset (left) and Normalize.css (right). (Credit: screenshot of a Codepen created by Zach Wolf.)

IMAGE REPLACEMENT TECHNIQUES

Before web fonts were a viable option, we needed to use an image anytime we wanted text in a font fancier than Times or Helvetica. Fortunately, that is no longer the case, but every now and then, it may still be necessary to replace a text element with an image in a way that is still accessible to screen readers. One common scenario is using a stylized logo in place of a company name in a heading (see **Note**).

Removing the text altogether and replacing it with an `img` element is a bad idea because the text content is gone forever. The solution is to use a CSS-based image replacement technique that uses the image as a background in the element, then shifts the text out of the way so that it is not rendered on the page. Visual browsers see the background image, while the text content stays in the file for the benefit of search engines, screen readers, and other assistive devices. Everybody wins!

Many image replacement techniques have been developed over the years (see **Note**), but the most popular is the Phark technique created by Mike Rundle. It uses a large negative indent to move the text off to the left of the visible page.

In the example in FIGURE 19-7, I use the Phark technique to display the Jenware logo in place of the `h1` "Jenware" text in the HTML source. The markup is simple:

```
<h1 id="logo">Jenware</h1>
```

NOTE

Before going through the effort of an image replacement technique, consider whether the `alt` *text in an* `img` *element is all you need. In the case of a logo, the* `alt` *text could represent the company name should the image of the logo not be seen. Frankly, the logo example in this section could probably be handled that way. That said, there may be instances in which you need to replace an actual string of text with an image, in which case an image replacement technique might be a good thing to have in your CSS toolbox.*

The style rule is as follows:

```
#logo {
  width: 450px;
  height: 80px;
  background: url(jenware.png) no-repeat;
  text-indent: -9999px;
}
```

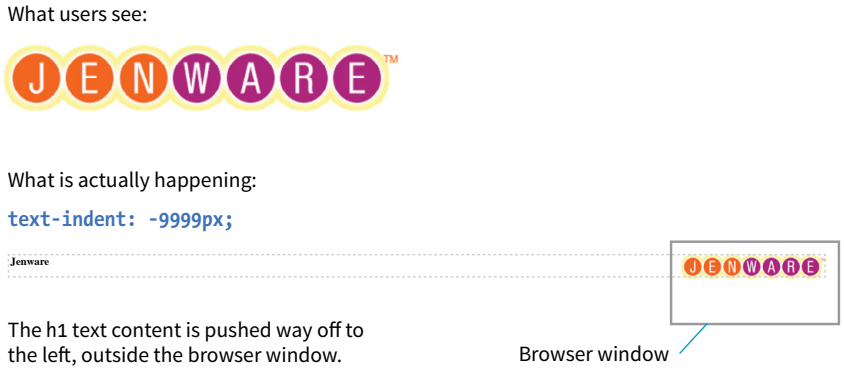


FIGURE 19-7. The Phark image replacement technique hides the HTML text by pushing it out of the visible element box with a large negative text indent so only the background image displays.

There are a few things of note here. First, the **h1** element displays as a block by default, so we can just specify its **width** and **height** to match the dimensions of the image used as a background. The **text-indent** property pushes the word *Jenware* over to the left by 9,999 pixels. This requires the browser to render a very wide element box, but the performance hit is minimal.

The downside to any image replacement approach is that it means an extra request to the server for every image used. It can also be more work creating graphics every time a heading changes. Again, before you reach for an image replacement, consider whether a web font or inline image with **alt** text may do the trick. In the next section, we'll look at a way to curb unnecessary server requests.

CSS SPRITES

When I talked about performance back in **Chapter 3, Some Big Concepts You Need to Know**, I noted that you can improve site performance by reducing the number of requests your page makes to the server (a.k.a. HTTP requests). One strategy for reducing the number of image requests is to combine all your little images into one big image file so that only one image gets requested. The large image that contains multiple images is known as a sprite, a term coined by the early computer graphic and video game industry. That image

NOTE

You can view a gallery of old techniques at "The Image Replacement Museum," assembled by Marie Mosley (css-tricks.com/the-image-replacement-museum/).

gets positioned in the element via the **background-position** property in such a way that only the relevant portion of it is visible. An example should make this clear.

If I want to show a collection of six social media icons on my page, I can turn those six graphics into one sprite and reduce the number of HTTP requests accordingly (FIGURE 19-8). You can see in the figure that the icons have been stacked into one tall graphic (*social.png*). This example also uses an image replacement technique so the text for each link is still available to screen readers.

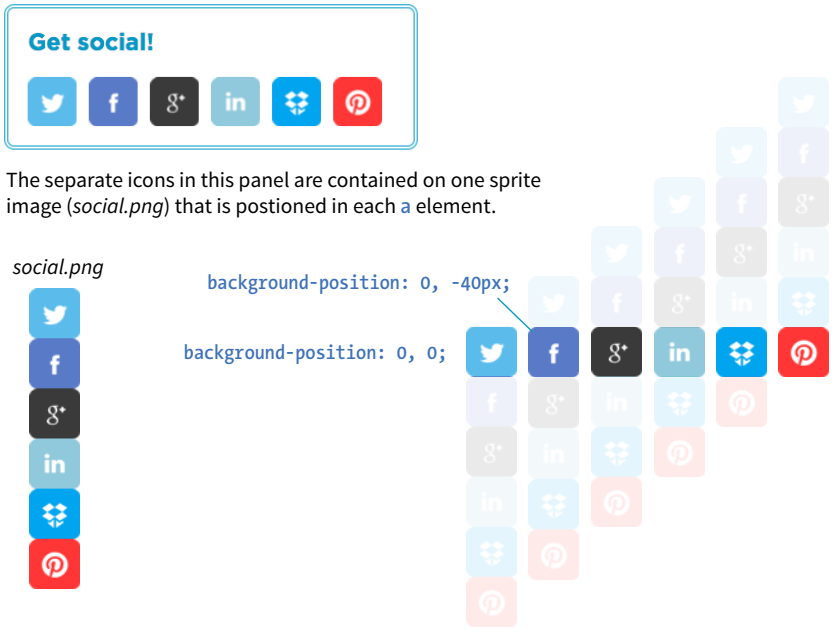


FIGURE 19-8. Replacing separate graphic files with one sprite image cuts down on the number of HTTP requests to the server and improves site performance.

THE MARKUP

```
<ul>
  <li><a href="" class="hide twitter">Twitter</a></li>
  <li><a href="" class="hide fb">Facebook</a></li>
  <li><a href="" class="hide gplus">Google+</a></li>
  <li><a href="" class="hide linkedin">LinkedIn</a></li>
  <li><a href="" class="hide dropbox">Dropbox</a></li>
  <li><a href="" class="hide pinterest">Pinterest</a></li>
</ul>
```

THE STYLES

```
.hide {
  text-indent: 100%;
  white-space: nowrap;
  overflow: hidden;
}
```

```
li a {
  display: block;
  width: 40px;
  height: 40px;
  background-image: url(social.png);
}
li a.twitter { background-position: 0 0; }
li a.fb { background-position: 0 -40px; }
li a.gplus { background-position: 0 -80px; }
li a.linkedin { background-position: 0 -120px; }
li a.dropbox { background-position: 0 -160px; }
li a.pinterest { background-position: 0 -200px; }
```

In the markup, each item has two **class** values. The **hide** class is used as a selector to apply an image replacement technique. This one was developed by Scott Kellum and uses a left indent of 100% to move the text out of sight. The other **class** name is particular to each social network link. The unique **class** values allow us to position the sprite appropriately for each link.

At the top of the style sheet you'll see the image replacement styles. Notice in the next rule that all link (**a**) elements use *social.png* as their background image.

Finally, we get to the styles that do the heavy lifting. The **background-position** is set differently for each link in the list, and the visible element box works like a little window revealing a portion of the background image. The first item has the value **0,0**; this positions the top-left corner of the image in the top-left corner of the element box. To make the Facebook icon visible, we need to move the image *up* by 40 pixels, so its vertical position is set to –40px (its horizontal position of 0 is fine). The image is moved up by 40-pixel increments for each link, revealing image areas farther and farther down the sprite stack.

In this example, all of the icons have the same dimensions and stack up nicely, but that is not a requirement. You can combine images with a variety of dimensions on one sprite. The process of setting a size for the element and then lining the sprite up perfectly with the **background-position** property is the same.

Sprite Generators

There are many online tools that create sprite image files and their respective styles automatically. Just upload or drag-and-drop your individual graphics to the page, and the tool does the rest. One that I find easy to use is CSSsprites (*css.spritegen.com*). If you need your sprites to be responsive, use their responsive version at *responsive-css.spritegen.com*.

CSS FEATURE DETECTION

One of the dominant challenges facing web designers and developers is dealing with uneven browser support. Useful new CSS properties emerge regularly, but it takes a while for them find their way into browsers, and it takes much longer for the old non-supporting browsers to fade into extinction.

Fortunately, we have a few methods for checking to see if a browser supports a particular feature so we can take advantage of cutting-edge CSS while also providing thoughtful fallbacks for non-supporting browsers. Using feature detection with fallbacks sure beats the alternatives of a) not using a property

until it is universally supported, or b) using it and letting users with non-supporting browsers have a broken experience.

We'll look at two ways to detect whether a feature is supported: feature queries with a new CSS at-rule (**@supports**) and a JavaScript-based tool called Modernizr.

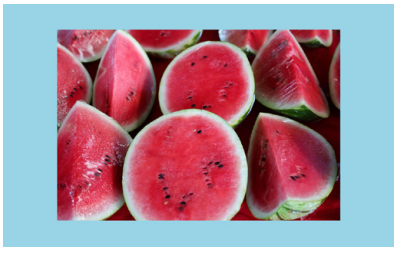
CSS Feature Queries (@supports)

The CSS3 Conditional Rules Module Level 3 (*www.w3.org/TR/css3-conditional/*) introduces the **@supports** rule for checking browser support of a particular property and value declaration. Commonly referred to as a feature query, it works like a media query in that it runs a test, and if the browser passes that test, it applies the styles contained in the brackets of the at-rule. The syntax for **@supports** is as follows:

```
@supports (property: value) {
  /* Style rules for supporting browsers here */
}
```

Note that the query is for an entire declaration, both the property and a value. It was designed this way because sometimes you may test for a new property (such as **initial-letter**), and sometimes you may need to test for a new value for an existing property. For example, the **display** property is universally supported, but the newer **grid** keyword value is not. Note also that there is no semicolon at the end.

Let's look at a more specific example. I think it would be cool to use the new **mix-blend-mode** property to make a photo of watermelons blend in with the background (similar to a Layer Blending Mode in Photoshop). As of this writing, it is supported only in Firefox, Chrome, and Safari. As a fallback for non-supporting browsers, I create a somewhat less interesting blended effect using the **opacity** property (FIGURE 19-9).



Original image (no effect)

As seen on browsers that support `mix-blend-mode: multiply;`

Fallback for non-supporting browsers (`opacity: .5`)

FIGURE 19-9. The original image (left), the result using the **mix-blend-mode** property with **multiply** keyword (center), and the fallback style using **opacity** (right).

As of this writing, the best practice is to specify the fallback styles first, and then override them with a set of styles targeted at browsers that support the feature. Note that I also need to set the **opacity** back to 1 so it overrides my fallback style.

THE MARKUP

```
<div id="container">
  <figure class="blend">
    <img src="watermelon.jpg" alt="">
  </figure>
</div>
```

THE STYLES

```
#container {
  background-color: #96D4E7;
  padding: 5em;
}
.blend img {
  opacity: .5;
}
@supports (mix-blend-mode: multiply) {
  .blend img {
    mix-blend-mode: multiply;
    opacity: 1;
  }
}
```

WARNING

The browser has to report for itself whether it has implemented the feature. If the feature is implemented in a buggy way, you may still encounter problems even when using feature queries.

Operators

The **@supports** rule can be used with three operators to refine the feature test: **not**, **and**, and **or**:

not

The **not** operator lets us test for when a specific property/value pair is *not* supported.

```
@supports not (mix-blend-mode: hue) {
  /* styles for non-supporting browsers */
}
```

Someday, this will be useful for supplying fallback styles, but with the current browser support, you risk non-supporting browsers skipping everything in the **@supports** rules, including the fallbacks. That's why I used the override method in the previous example.

and

Applies styles only when all of the conditions in a series of two or more are met.

```
@supports (border-radius: 10em) and (shape-outside: circle()) {
  /* styles that apply only when the browser supports
     shape-outside AND border-radius */
}
```

or

Use the **or** operator to apply styles when any of a series of conditions are met. This one is particularly useful for vendor-prefixed properties.

```
@supports (-webkit-transform: rotate(10deg)) or
          (-ms-transform: rotate(10deg)) or
          (transform: rotate(10deg))
  /* transform styles */
}
```

Browser support

> **AUTHORING TIP**
>
> Not every new feature needs a feature query. Some features, such as **border-radius**, simply don't render on non-supporting browsers, and that is just fine.

Feature queries began working in Chrome, Firefox, and Opera back in 2013, and they are supported by every version of Microsoft Edge. Safari added support in version 9 in 2015. Unfortunately, no version of Internet Explorer supports feature queries, which leaves a big hole in the support picture until those old browsers go away.

Non-supporting browsers use your fallback design, so make sure that it is usable at the very least. Beware, however, of browsers that do not support **@supports** but may support newer CSS features that you might be inclined to test. Flexbox is a great example. Safari 8 recognizes the Flexbox properties, but does not recognize **@supports**, so if all of your Flexbox layout rules are tucked away inside a feature query, Safari 8 won't see them. That's why feature queries aren't the best tool for detecting Flexbox or any property that has better support than **@supports** itself. Grid Layout, on the other hand, is a great place to put feature queries to work because every browser that supports **display: grid** also supports **@supports**. Again, *CanIUse.com* is a good resource for comparing support.

Pros and cons

Feature queries are an exciting new tool for web development. They allow us to take advantage of new CSS properties sooner in a way that doesn't rely on JavaScript (we'll look at Modernizr, a JavaScript solution, next). Downloading and running a script (even a small one) is slower than using CSS alone.

On the downside, limited browser support (for now) means **@supports** is not as far-reaching as Modernizr. However, if it accomplishes your goals, it should be your first choice. Fortunately, the browser environment will only continue to improve, giving CSS feature queries the advantage over a script-based solution in the long run.

So what is this "Modernizr" you're hearing so much about?

Modernizr

Modernizr is a lightweight JavaScript library that runs behind the scenes and tests for a long list of HTML5 and CSS3 features when the page is loaded in

the browser. For each feature it tests, it stores the result (supports/doesn't support) in a JavaScript object that can be accessed with scripts and optionally as a class name in the **html** root element that can be used in CSS selectors. I'm going to focus on the latter CSS method.

How it works

When Modernizr runs, it appends the **html** element with a class name for each feature it detects. For example, if it is configured to test for Flexbox, when it runs on a browser that *does* support Flexbox, it adds the **.flexbox** class name to the **html** element:

```
<html class="js flexbox">
```

If the feature is not supported, it adds the feature name with a **.no-** prefix. On a non-supporting browser, the Flexbox test would be reported like this:

```
<html class="js no-flexbox">
```

With the class name in place on the root element, everything on the page becomes part of that class. We can use the class name as part of a selector to provide different sets of styles depending on feature support:

```
.flexbox nav {
  /* flexbox styles for the nav element here */
}

.no-flexbox nav {
  /* fallback styles for the nav element here */
}
```

This example is short and sweet for demonstration purposes. Typically, you'll use Modernizr to test for many features, and the **html** tag gets filled with a long list of class names.

Modernizr is a lightweight JavaScript library that tests for a variety of HTML and CSS features.

How to use it

First, you need to download the *Modernizr.js* script. Go to *Modernizr.com* and find the Download link. From there you can customize the script to contain just the HTML and CSS features you want to test, a nice way to keep the file size of the script down. Click the Build button, and you will be given several options for how it can be saved. A simple click on Download saves the script in a *.js* file on your computer.

Once you have your script, put it in the directory with the rest of the files for your project. Add it to the **head** of your HTML document, before any linked style sheets or other scripts that need to use it:

```
<head>
  <script src="modernizr-custom.js"></script>
  <!--other scripts and style sheets -->
</head>
```

Finally, open your HTML document and assign the **no-js** class name to the **html** element.

```
<html class="no-js">
```

Modernizr will change it to **js** once it detects that the browser supports JavaScript. If JavaScript (and therefore Modernizr) fails to run, you will not know whether or not features are supported.

Pros and cons

Modernizr is one of the most popular tools in web developers' arsenals because it allows us to design for particular features rather than whole browsers. It is easy to use, and the Modernizr site has thorough and clear documentation to help you along. Because it's JavaScript, it works on the vast majority of browsers. The flip side to that, however, is that because it relies on JavaScript, you can't be 100% certain that it will run, which is its main disadvantage. It will also be slightly slower than using CSS alone for feature detection.

WRAPPING UP STYLE SHEETS

That concludes our whirlwind tour of Cascading Style Sheets. You've come a long way since styling an **h1** and a **p** back in **Chapter 11, Introducing Cascading Style Sheets**. By now, you should be comfortable formatting text and even doing basic page layout. While CSS is easy to learn, it takes a lot of time and practice to master. If you get stuck, you will find that there are many resources online to help you find the answers you need. The nice thing about CSS is that you can start with just the basics and then build on that knowledge as you gain proficiency in your web development skills.

In the next chapter, I'll introduce you to tools that web developers use to improve their workflow, including tools for writing CSS more efficiently and optimizing the results. But if you're feeling overwhelmed with CSS properties, you can breathe a sigh of relief. We're *done*!

TEST YOURSELF

See how well you picked up the CSS techniques in this chapter with these questions. As you may have guessed, the answers are available in **Appendix A**.

1. What is the purpose of a CSS reset?
   a. To override browser defaults
   b. To make presentation more predictable across browsers
   c. To prevent elements from inheriting unexpected styles
   d. All of the above

2. What is the purpose of a CSS sprite?
   a. To improve site performance
   b. To use small images in place of large ones, reducing file size
   c. To reduce the number of HTTP requests
   d. a and c
   e. All of the above

3. What is the purpose of an image replacement technique?
   a. To achieve really big text indents
   b. To use a decorative graphic in place of text
   c. To remove the text from the document and replace it with a decorative image
   d. To maintain the semantic content of the document
   e. b and d
   f. All of the above

4. Name two approaches to aligning form controls and their respective labels without tables. A general description will do here.

5. Match the style rules with their respective tables in FIGURE 19-10.
   a.
```
table { border-collapse: collapse;}
td { border: 2px black solid; }
```
   b.
```
table { border-collapse: separate; }
td { border: 2px black solid; }
```
   c.
```
table {
  border-collapse: separate;
  border-spacing: 2px 12px; }
td { border: 2px black solid; }
```
   d.
```
table {
  border-collapse: separate;
  border-spacing: 5px;
  border: 2px black solid; }
td { background-color: #99f; }
```
   e.
```
table {
  border-collapse: separate;
  border-spacing: 5px; }
td { background-color: #99f;
  border: 2px black solid; }
```

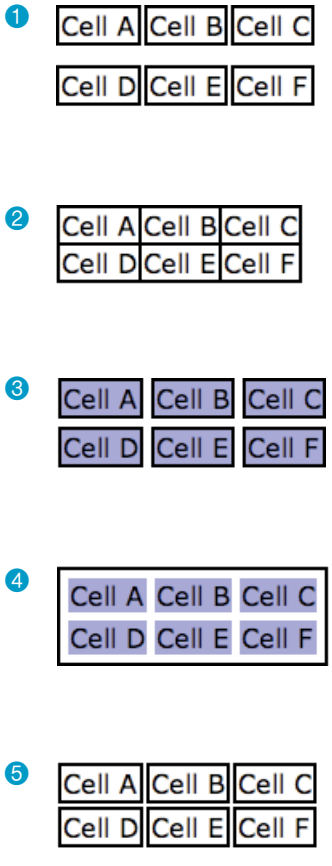


FIGURE 19-10. Match these tables with the code examples in Question 5.

6. Using Modernizr to test for **`border-radius`**, say whether the **`div`** will display with rounded corners based on the following generated class results:

```
.border-radius div {
  border: 1px solid green;
  border-radius: .5em;
}
```

   a. `<html class="js .no-border-radius">`

   b. `<html class="js .border-radius">`

   c. `<html class="no-js">`

7. As of this writing, what advantage does Modernizr have over CSS feature detection? What long-term advantage will CSS feature detection have over Modernizr?

CSS REVIEW: TABLE PROPERTIES

The following is a summary of the properties covered in this chapter.

Property	Description
`border-collapse`	Specifies whether borders between cells are separate or collapsed
`border-spacing`	Denotes the space between cells set to render as separate
`caption-side`	Specifies the position of a table caption relative to the table (top or bottom)
`empty-cells`	Specifies whether borders and backgrounds should render for empty cells
`table-layout`	Specifies how table widths are calculated

CHAPTER

20

MODERN WEB DEVELOPMENT TOOLS

In the exercises in this book, you've been writing static HTML pages with embedded style sheets, saving them, and opening them in your browser. Although that is a perfectly valid approach, it is likely not the way you would work if you were doing web development for a living. I figure if you are learning web design and development, you should be familiar with how things are done in a professional environment.

This chapter introduces you to some of the tools used by web developers to make their work easier and their code more robust:

- CSS processors for writing CSS more efficiently and optimizing the resulting code so it works across all browsers
- Build tools that automate the sorts of repetitive tasks you encounter when producing code
- Git, a version control program that keeps track of your previous versions and makes it easy for teams to work together on the same code

What these advanced tools have in common is that they are generally used with a command-line interface (CLI). So, before we look at specific tools, let's first get up to speed with the command line.

IN THIS CHAPTER

GETTING COZY WITH THE COMMAND LINE

You probably use a computer with a graphical user Interface (GUI), with icons that stand for files and folders, pull-down menus full of options, and intuitive actions like dragging files from folder to folder or into the trash.

Computer users in the '60s and '70s didn't have that luxury. The only way to get a computer to perform a task was to type in a command. Despite our

fancy GUIs, typing commands into a command-line terminal is far from obsolete. In fact, the more experienced you become at web development, the more likely it is you'll dip into the command line for certain tasks. If you are already a programmer, the command line will be nothing new.

The command line is still popular for a number of reasons. First, it is useful for accessing remote computers, and developers often need to access and manage files on remote web servers. In addition, it is easier to write a program for the command line than a standalone application with a GUI, so many of the best tools for optimizing our workflow exist as command-line programs only. A lot of those tools can be used together in a pipeline for accomplishing complex tasks.

The time- and sanity-saving benefits are powerful incentives to take on the command line. Trust me: if you can learn all those elements and style properties, you can get used to typing a few commands.

The Command-Line Terminal

The program that interprets the commands you type is called a shell (visual interfaces are also technically a shell; they're just fancier). Every Mac and Linux machine comes installed with Terminal, which uses a shell program called bash. On macOS, you will find the Terminal program in Applications → Utilities (FIGURE 20-1).

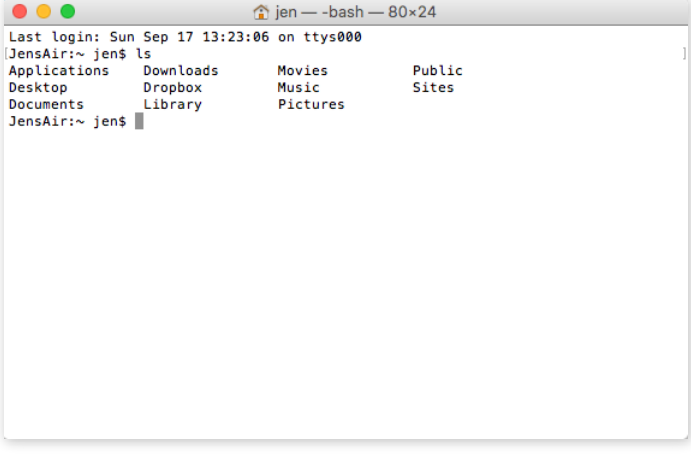


FIGURE 20-1. The Terminal window in macOS.

Windows users have a few more hoops to jump through to get set up. The default command-line tool on Windows is Command Prompt (most easily accessed with Search), which can perform many of the functions you may want to do as a developer; however, it does not use bash. Because so many tools use bash, it is better to install a bash-based shell emulator like

Cygwin (*cygwin.com*) or cmder (*cmder.net*). If you use Windows 10, it is recommended that you install a Linux environment on your machine by using Windows Subsystem (*msdn.microsoft.com/en-us/commandline/wsl/about*) or Ubuntu, available in the Windows store (*www.microsoft.com/en-us/store/p/ubuntu/9nblggh4msv6*).

Getting Started with Commands

When you launch a Terminal window, the first thing you see is a command-line prompt, which is a string of characters that indicates the computer is ready to receive your command:

```
$: _
```

The dollar sign is common, but you may see another symbol in your terminal program (see **Terminal Tip**). The underscore in this example stands for the cursor position, which may appear as a little rectangle or a flashing line.

The complete prompt that I see in Terminal begins with my computer's name ("JensAir") and an indication of the working directory—that is, the directory the shell is currently looking at. In GUI terms, the working directory is the folder you are "in." In this example, the tilde (**~**) indicates that I am looking at my root User directory. The "jen" before the prompt character is my username. In future examples, I will abbreviate the prompt to simply **$:**.

```
JensAir:~ jen$: _
```

When you see the prompt, type in a command, and hit Enter. The computer executes the command and gives you a new prompt when it is finished. It is very no-nonsense about it. For some commands, there may be feedback or information displayed before the next prompt appears. Sometimes everything happens behind the scenes, and all you see is a fresh prompt.

When you're learning about the command line, it is common to start with the built-in commands for navigating the file system, tasks typically handled by the Finder on the Mac and My Computer on Windows. Because they are fairly intuitive, that's where I'm going to start my simple command-line lesson as well.

A nice little utility to try as a beginner is **pwd** (for "print working directory"), which displays the complete path of the working (current) directory. To use it, simply type **pwd** after the prompt. It's a good one to try first because you can't break anything with it, but for seasoned users, it is useful for figuring out exactly where you've landed if you're disoriented. The forward slash indicates that this path starts at the root directory for the entire computer.

```
$: pwd
/Users/jen
```

Here's another easy (and low-risk!) example. Typing the **ls** command at the prompt returns a list of the files and directories in the working directory

TERMINAL TIP

You can customize the appearance of Terminal by selecting **Preferences → Profile** and changing the settings. If you want to keep yourself amused, you can change the prompt character from $ to the character of your choice, including an emoji (*osxdaily.com/2013/04/08/add-emoji-command-line-bash-prompt/*).

NOTE

Your user directory is the default root directory in Terminal and is represented by a tilde (~) in the prompt, as we saw in the previous example.

(*/Users/jen*). You can compare it to the Finder view of the same folder in FIGURE 20-2. They are two ways of looking at the same thing, just as *directory* and *folder* are two terms for the same thing depending on your view.

```
JensAir:~ jen$ ls
Applications    Downloads       Movies          Public
Desktop         Dropbox         Music           Sites
Documents       Library         Pictures
JensAir:~ jen$
```

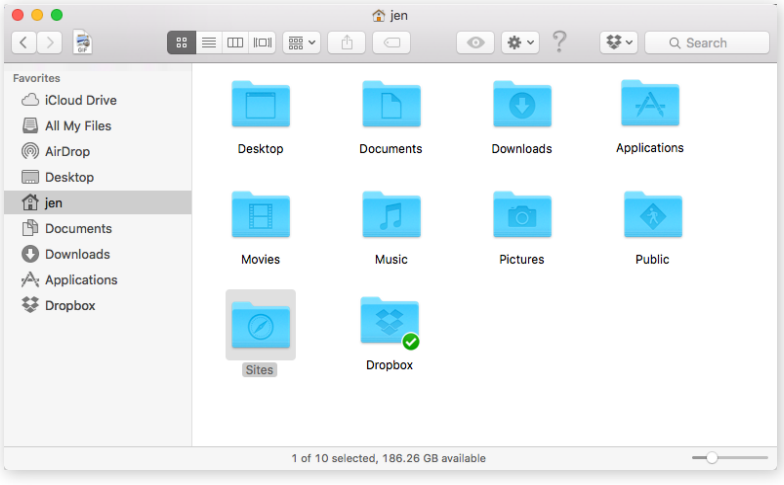


FIGURE 20-2. Finder view of the *jen* home folder.

Dotfiles

There are some files on your computer that are kept hidden in the Finder view. These files, known as dotfiles, start with (you guessed it) a dot, and they tend to handle information that is intended to function behind the scenes. If you type **`ls -a`** (**`-a`** is shorthand for "all"), you can reveal the dotfiles lurking in a directory. In macOS, it is possible to configure Finder to show dotfiles as well, but for most users' purposes, hidden is a good thing.

Some utilities, like **`pwd`**, require only their name at the prompt to run, but it is more common that you'll need to provide additional information in the form of flags and arguments. A flag changes how the utility operates, like an option or a preference. It follows the command name and is indicated by a single or double dash (**`-`**). In many cases, flags can be abbreviated to just their first letter because they are used in context with a particular utility. For example, I can modify the **`ls`** utility with the **`-l`** flag, which instructs the computer to display my directory contents in "long" format, including permission settings and creation dates:

```
JensAir:~ jen$ ls -l
total 0
drwxr-xr-x     5 jen  staff   170 Jul  8  2016 Applications
drwx------    57 jen  staff  1938 Sep 11 09:47 Desktop
drwx------    26 jen  staff   884 May 18 11:34 Documents
drwx------+ 151 jen  staff  5134 Sep  3 15:47 Downloads
drwx------@  48 jen  staff  1632 Aug 16 16:34 Dropbox
drwx------@  72 jen  staff  2448 Jul 15 11:21 Library
drwx------    22 jen  staff   748 Oct  6  2016 Movies
drwx------    12 jen  staff   408 Sep 29  2016 Music
drwx------    14 jen  staff   476 Oct 13  2016 Pictures
drwxr-xr-x     6 jen  staff   204 May  6  2015 Public
drwxr-xr-x    11 jen  staff   374 Jul 10  2016 Sites
JensAir:~ jen$
```

An argument provides the specific information required for a particular function. For example, if I want to change to another directory, I type **cd** (for "change directory") as well as the name of the directory I want to go to (see **Mac Terminal Tip**). To make my Dropbox directory the new working directory, I type this:

```
JensAir:~ jen$: cd Dropbox
```

After I hit Enter, my prompt changes to **JensAir:Dropbox jen$**, indicating that I am now in the Dropbox directory. If I entered **ls** now, I'd get a list of the files and folders contained in the Dropbox folder (definitely way too long to show here).

To go up a level, and get back to my home user directory (**~**), I can use the Unix shorthand for "go up a level": **..** (remember that from your URL path lesson?). The returned prompt shows I'm back at my root directory (**~**).

```
JensAir:Dropbox jen$ cd ..
JensAir:~ jen$
```

Some other useful file-manipulation commands include **mv** (moves files and folders), **cp** (copies files), and **mkdir** (creates a new empty directory). The **rm** command removes a file or folder in the working directory. Be careful with this command, however, because it doesn't just move files to the Trash; it removes them from your computer entirely (see the **"A Word of Caution"** note).

Another handy command is **man** (short for *manual*), which displays documentation for any command you pass to it. For example, **man ls** shows a description of the **ls** (list) command and all of its available flags. Some man pages are long. To move down in the scroll, hitting the Return key moves you down one line at a time. To move down a page at a time, hit fn+down arrow on a Mac or Shift+Page Down on Linux. To go back up a page, it's fn+up arrow or Shift+Page Up, respectively. Finally, to quit out of the man page, type **q** to return to the prompt.

Learning More

Not surprisingly, these commands are just the tip of the tip of the iceberg when it comes to command-line utilities. For a complete list of commands that can be used with bash, see "An A–Z Index of the Bash Command Line for Linux" at *ss64.com/bash/*. You'll pick these up on an as-needed basis, so don't get overwhelmed. In addition, as you start installing and using new tools like the ones listed in this chapter, you'll gradually learn the commands, flags, and arguments for those too. All part of a day's work!

Clearly, I don't have the space (and if I'm being honest, the experience) to write a comprehensive tutorial on the command line in this chapter, but you will find books and plenty of tutorials online that can teach you. I found Michael Hartl's tutorial "Learn Enough Command Line to Be Dangerous"

MAC TERMINAL TIP

On the Mac, Terminal is well connected to Finder. If you need to enter a pathname to a directory or a file, you can drag the icon for that file or folder from Finder to Terminal, and it will fill in the pathname for you.

COMMAND-LINE TIP

Typing **cd** followed by a space always takes you back to your home directory.

A WORD OF CAUTION

The command line allows you to muck around in critical parts of your computer that your GUI graciously protects from you. It's best not to type in a command if you don't know exactly what it does and how it works. Make a complete backup of your computer before you start playing around with command line so you have the peace of mind that your files are still available if something goes horribly wrong.

to be thorough and accessible if you are starting from square one (*www.learnenough.com/command-line-tutorial#sec-basics*). I also recommend the series of tutorials from Envato Tuts+, "The Command Line for Web Design" (*webdesign.tutsplus.com/series/the-command-line-for-web-design--cms-777*). If you enjoy video tutorials, try the "Command Line for Non-Techies" course by Remy Sharp (*terminal.training*).

Now that you have a basic familiarity with the command line, let's look at tools you might use it for, beginning with tools for writing and optimizing CSS.

Here's the Thing About Development Tools

Be aware that the development tool landscape is ever-shifting. Tools come and go in rapid-fire fashion, with the whole development community jumping on one framework bandwagon, then moving on to the next new thing. It's difficult to write about specific tools in a book that has to last a couple of years. I have made an effort to present the most established and stable tools as of early 2018, but you should know that there are many more niche tools out there, and by the time you read this, some new tool may be all the rage. As you read this chapter, focus on the functions the tools perform, start with the ones mentioned here when you're ready, and keep your ear to the ground for newer options.

CSS POWER TOOLS (PROCESSORS)

I know that you are just getting used to writing CSS, but I would be negligent if I didn't introduce you to some advanced CSS power tools that have become central to the professional web developer workflow. They fall into two general categories:

- Languages built on top of CSS that employ time-saving syntax characteristics of traditional programming languages. These are traditionally known as preprocessors. The most popular preprocessors as of this writing are Sass, LESS, and Stylus. When you write your styles in one of these languages, you have to use a program or script to convert the resulting file into a standard CSS document that browsers can understand.
- CSS optimization tools take your clean, standard CSS and make it even better by improving cross-browser consistency, reducing file size for better performance, and enhancing many other tasks. Tools that optimize browser-ready CSS are commonly known as postprocessors.

Before you get too comfortable with the terms *preprocessor* and *postprocessor*, you should know that the distinction is not exactly clear-cut. Preprocessors have always been able to do some of the optimization tasks that postprocessors are good for, and postprocessors are starting to allow some functions typically found in preprocessors. The lines are blurring, so some folks refer to all of these tools simply as CSS processors, including souped-up special syntaxes for authoring as well as CSS optimizers. Many CSS processor functions are also built in to third-party tools such as CodeKit (*codekitapp.com*, *Mac only*) for one-stop shopping. I think it is beneficial for you to be familiar with the traditional terms as they are still in widespread use, and I'm going to use them here for the sake of simplicity.

NOTE

Hat tips to Stefan Baumgartner, whose article "Deconfusing Pre- and Post-Processing" (medium.com/@ddprrt/deconfusing-pre-and-post-processing-d68e3bd078a3) helped me sort out all this CSS processing stuff, and David Clark for his clarifying article "It's Time for Everyone to Learn About PostCSS" (davidtheclark.com/its-time-for-everyone-to-learn-about-postcss/).

Introduction to Preprocessors (Especially Sass)

Preprocessors consist of an authoring syntax and a program that translates (or compiles, to use the proper term) files written in that syntax to plain old CSS files that browsers can use (FIGURE 20-3). For example, in Sass, you write

in the Sass syntax language and save your files with the *.scss* suffix, indicating it is in that language and not a CSS file. The Sass program, originally written in the Ruby language (see **Technical Note**), converts the SCSS syntax to standard CSS syntax and saves the resulting file with the *.css* suffix. LESS and Stylus work the same way, but they use JavaScript for conversion. All of these tools are installed and run via the command line.

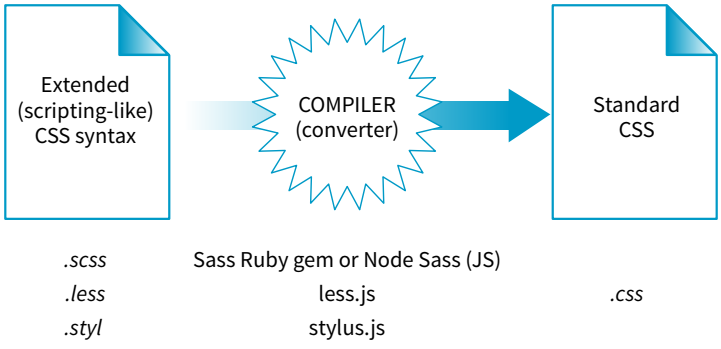


FIGURE 20-3. A simplified view of the role of a preprocessor.

TECHNICAL NOTE

The Sass project wrote a newer version in C++ that can be used with other languages. Most developers now compile *.scss* files with Node Sass because it integrates more smoothly into a workflow with other Node.js tools.

By far, the most popular preprocessor is Sass ("Syntactically awesome style sheets"), created by Hampton Catlin and Nathan Weizenbaum, who were tired of the repetitiveness of normal CSS. Their new syntax allowed CSS authors to use the type of shortcuts typical in scripting. Originally, it used an indented, bracket-free syntax (which is still an option), but a later release known as SCSS (for "Sassy CSS") is based on the bracketed (`{ }`) CSS format we know and love. In fact, a valid CSS document would also be a valid SCSS document. This makes it much easier to get started with Sass, because it is familiar, and you can use just a little bit of Sass in style sheets written the way you've learned in this book.

I'm going to show you a few examples of Sass syntax to give you the general idea. When you are ready to take on learning Sass, a great first step is Dan Cederholm's book *Sass for Web Designers* (A Book Apart). I've listed additional resources at the end of the section. In the meantime, let's look at three of the most popular Sass features: nesting, variables, and mixins.

Nesting

Let's say you have an HTML document with a `nav` element that contains an unordered list for several menu options. Sass lets you nest the style rules for the `nav`, `ul`, and `li` elements to reflect the structure of the HTML markup. That alleviates the need to type out the selectors multiple times—the Sass compiler does that for you. The following example shows nested styles as they can be written in Sass syntax:

Sass lets you nest styles to match the structure of the markup.

```
nav {
  margin: 1em 2em;

  ul {
    list-style: none;
    padding: 0;
    margin: 0;

    li {
      display: block;
      width: 6em;
      height: 2em;
    }
  }
}
```

When Sass converts the SCSS file to standard CSS, it compiles to this:

```
nav {
  margin: 1em 2em;
}

nav ul {
  list-style: none;
  padding: 0;
  margin: 0;
}

nav ul li {
  display: block;
  width: 6em;
  height: 2em;
}
```

Variables

A variable is a value you can define once, and then use multiple times throughout the style sheet. For example, O'Reilly uses the same shade of red repeatedly on its site, so their developers could create a variable named "oreilly-red" and use the variable name for color values. That way, if they needed to tweak the shade later, they need to change the variable value (the actual RGB color) only in one place. Here's what setting up and using a variable looks like in Sass:

A variable is a value that you define once and reuse throughout the style sheet.

```
$oreilly-red: #900;

a {
  border-color: $oreilly-red;
}
```

When it compiles to standard CSS, the variable value is plugged into the place where it is called:

```
a {
  border-color: #900;
}
```

The advantage of using a variable is that you can change the value in one place instead of searching and replacing through the whole document. When teams use variable names, it also helps keep styles consistent across the site.

Mixins

Sass allows you to reuse whole sets of styles by using a convention called mixins. The following example saves a combination of background, color, and border styles as a mixin named "special." To apply that combination of styles, **`@include`** it in the declaration and call it by name:

Mixins are sets of rules that can be reused.

```
@mixin special {
  color: #fff;
  background-color: #befc6d;
  border: 1px dotted #59950c;
}
a.nav {
  @include special;
}
a.nav: hover {
  @include special;
  border: 1px yellow solid;
}
```

When compiled, the final CSS looks like this:

```
a.nav {
  color: #fff;
  background-color: #befc6d;
  border: 1px dotted #59950c;
}
a.nav: hover {
  color: #fff;
  background-color: #befc6d;
  border: 1px dotted #59950c;
  border: 1px yellow solid;
}
```

Notice that the hover state has a second border declaration that overrides the values in the mixin, and that's just fine. Mixins are a popular solution for dealing with vendor prefixes. Here is a mixin for **`border-radius`** that includes an argument (a placeholder for a value you provide indicated with a **`$`**):

```
@mixin rounded($radius) {
  -webkit-border-radius: $radius;
     -moz-border-radius: $radius;
          border-radius: $radius;
}
```

When including the mixin in a style rule, provide the value for **`$radius`**, and it gets plugged into each instance in the declarations:

```
aside {
  @include rounded(.5em);
  background: #f2f5d5;
}
```

This compiles to the following:

```
aside {
  -webkit-border-radius: .5em;
     -moz-border-radius: .5em;
          border-radius: .5em;
  background: #f2f5d5;
}
```

Building a mixin around fill-in-the-blank arguments makes them reusable and even shareable. Many developers create their own mixin libraries to use on multiple projects. You can also take advantage of existing mixin libraries in tools like Compass (an open source CSS authoring framework at *compass-style.org*) or Bourbon (*bourbon.io*). By the time you read this, there may be others, so search around to see what's available.

Sass resources

Nesting, variables, and mixins are only a tiny fraction of what Sass can do. It can handle math operations, "darken" and "lighten" colors mathematically on the fly, and process `if`/`else` statements, just to name a few features.

Once you get some practice under your belt and feel that you are ready to take your style sheets to the next level, explore some of these Sass and LESS articles and resources:

- The Sass site (*sass-lang.com*)
- "Getting Started with Sass," by David Demaree (*alistapart.com/articles/getting-started-with-sass*)
- "An Introduction to LESS, and LESS Vs. Sass," by Jeremy Hixon (*www.smashingmagazine.com/2011/09/an-introduction-to-less-and-comparison-to-sass/*)

Introduction to Postprocessors (Mostly PostCSS)

As I mentioned earlier, postprocessors are scripts that optimize standard CSS code to make it better (FIGURE 20-4). "Better" usually means consistent and bug-free browser support, but there are hundreds of postprocessing scripts that do a wide variety of cool things. We'll look at some examples in a moment.

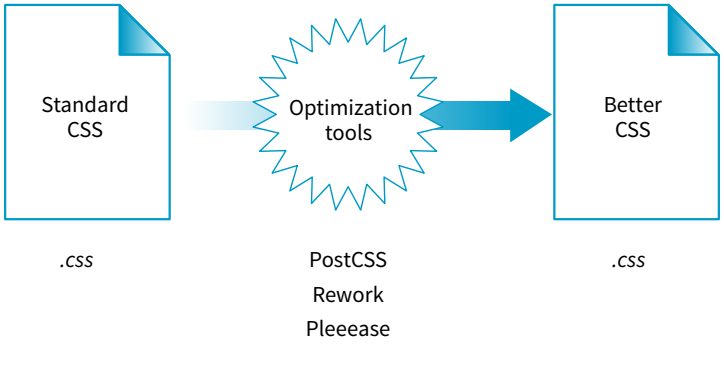


FIGURE 20-4. Postprocessors optimize existing, standard CSS files.

LESS and Stylus

Sass is the most widely used preprocessor, but it's not the only game in town for nesting, variables, mixins, and more.

LESS (*lesscss.org*) is another CSS extension with scripting-like abilities. It is very similar to Sass, but it lacks advanced programming logic features (such as `if`/`else` statements) and has minor differences in syntax. For example, variables in LESS are indicated by the `@` symbol instead of `$`. The other major difference is that a LESS file is processed into regular CSS with JavaScript (*less.js*) instead of Ruby. Note that compiling a LESS file into CSS is processor-intensive and would bog down a browser. For that reason, it is best to do the conversion to CSS before sending it to the server. LESS offers a very active developer community and the "LESShat" mixin library.

Stylus (*stylus-lang.com*) is the relative new kid on the preprocessor block. It combines the logic features of Sass with the convenience of a JavaScript-based compiler (*stylus.js*). It also offers the most flexible syntax: you can include as much CSS "punctuation" (brackets, colons, and semicolons) as you like, prepend variables with a `$` or not, and treat mixin names like regular properties. Developers who use Stylus like how easy it is to write and compile. Nib and Axis are two mixin libraries available for Stylus.

When you are ready to take your CSS authoring to the next level, you can give each of these a try. The one you choose is a matter of personal preference; however, if you are working on a professional development team, one may be chosen for you.

The poster child for postprocessing is Autoprefixer, which takes the CSS styles you write, scans them for properties that require vendor prefixes, and then inserts the prefixed properties automatically. What a time- and headache-saver!

Back in **Chapter 16, CSS Layout with Flexbox and Grid**, we used Autoprefixer via a web page interface (*autoprefixer.github.io*) to generate the required prefixes. Although the web page is handy (especially while you are learning), it is more common for postprocessors to be implemented with a task runner such as Grunt or Gulp. We'll take a quick look at them later in this chapter.

As of this writing, the postprocessing scene is dominated by PostCSS (*postcss.org*). PostCSS is "a tool for transforming CSS with JavaScript" created by Andrey Sitnik, who also created Autoprefixer. PostCSS is both a JavaScript-based program (a Node.js module, to be precise) and an ecosystem of community-created plug-ins that solve all sorts of CSS problems.

PostCSS parses the CSS (or a CSS-like syntax such as Sass or LESS), analyzes its structure, and makes the resulting "tree" available for plug-ins to manipulate the code (see **Note**).

NOTE

The "tree" is formally known as the Abstract Syntax Tree (AST) and is the API for PostCSS plug-ins.

This open API makes it easy for anyone to create a PostCSS plug-in, and as a result, there are literally hundreds of plug-ins created and shared by developers (see them at *www.postcss.parts*). They range from the life-saving to the esoteric, but because it is a modular system, you can pick and choose just the tools that you find useful or even create your own. Here are just a few:

- Stylelint (*stylelint.io*) checks your CSS file for syntax errors (a process called linting) and redundancies.
- CSSNext (*cssnext.io*) allows you to use future CSS Level 4 features today by generating fallbacks that work across browsers that haven't implemented those features yet.
- PreCSS (*github.com/jonathantneal/precss*) is a bundle of plug-ins that lets you write Sass-like syntax (loops, conditionals, variables, mixins, and so on) and converts it to standard CSS. This is an example of a postprocessor being used to aid authoring, which is where the line between pre- and postprocessing gets blurred.
- Fixie (*github.com/tivac/fixie*) inserts hacks that are required to make effects work in old versions of Internet Explorer ("Fix-IE," get it?).
- Color format converters translate alternative color formats (such as HWB, HCL, and hex + alpha channel) to standard RGB or hexadecimal.
- The Pixrem plug-in converts rem units to pixels for non-supporting browsers.
- The List-selectors plug-in lists and categorizes the selectors you've used in your style sheet for code review. It is an example of a plug-in that does not alter your file but gives you useful information about it.

NOTE

PostCSS is not the only postprocessor out there. Other frameworks include Rework (github.com/reworkcss/rework) and Pleeease (pleeease.io), but they are not as full featured. By the time you read this, there may be many more. So goes the world of web development tools.

From that short list, you can probably see why postprocessors have become so popular. They free you up to write CSS with the syntax you want, taking advantage of cutting-edge properties and values, but with the peace of mind that everything will work well across browsers. They also eliminate the need to know about every browser idiosyncrasy, past and present, in order to do your job. It's definitely worth knowing about even if you aren't quite ready to take it on right away. Check out these resources for more information:

- Drew Minns' article "PostCSS: A Comprehensive Introduction" for *Smashing Magazine* (*www.smashingmagazine.com/2015/12/introduction-to-postcss/*)
- The Envato Tuts+ tutorial "PostCSS Deep Dive" (*webdesign.tutsplus.com/series/postcss-deep-dive--cms-889*)

BUILD TOOLS (GRUNT AND GULP)

In the world of software, a build process is required to test source code and compile it into a piece of executable software. As websites evolved from a collection of static HTML files to complex JavaScript-reliant applications, often generated from templates, build tools have become integral to the web development workflow as well. Some web build tools like Grunt and Gulp are commonly referred to as task runners. You use them to define and run various tasks (anything you might do manually from the command line) on your working HTML, JavaScript, CSS, and image files to get them ready to publish.

Automation

You can automate your tasks, too, so they happen in the background without your needing to type commands. To do this, you tell the build tool to "watch" your files and folders for changes. When a change is detected, it triggers the relevant tasks to run automatically as you've configured them.

Once you have the task runner configured and set to watch your files, you can go about your business writing CSS, and all that command-line stuff happens for you without ever touching a terminal appplication. Here's how that might look. Imagine making a change to your Sass file and saving it. Grunt instantly sees that the *.scss* file has changed, automatically converts it to *.css* (see **Note**), and then reloads the browser to reflect your change.

NOTE

There is a Grunt plug-in for converting SCSS files, but it is not as full featured as Ruby.

Some Common Tasks

The previous section on CSS processors should have given you an idea of some things that would be nice to automate. Allow me to list several more to give you a solid view of the ways task runners make your job easier.

- Concatenation. It is common for web teams to divide style sheets and scripts into small, specialized chunks of *.css* and *.js*. When it's time to publish, however, you want as few calls to the server as possible for performance purposes, so those little chunks get concatenated (put together) into master files.
- Compression and "minification." Another way to improve performance is to make your files as small as possible by removing unnecessary spaces and line returns. Build tools can compress your CSS and minify JavaScript.
- Checking your HTML, CSS, and JavaScript for errors (linting).
- Optimizing images with tools that squeeze down the file size of all the images in a directory.
- Help committing or pushing changes to a version control repository (Git).
- Refreshing your browser to reflect whatever changes you just made to a file (LiveReload plug-in).
- Building final HTML files from templates and content data (see the sidebar **"Building Sites with Data and Templates"**).
- Running CSS pre- and postprocessors.

Grunt and Gulp

The first and most established web build tool is Grunt (*gruntjs.com*), presumably named for handling all of the "grunt work" for you. It is a JavaScript tool built on the open source Node.js framework, and you operate it using the command line. The compelling thing about Grunt is that the development community has created literally thousands of plug-ins that perform just about any task you can think of. Just download one, configure it, and start using it. You do not have to be a JavaScript master to get started.

Another popular option is Gulp (*gulpjs.com*), which has the advantage of running a little faster but also requires more technical knowledge than Grunt because you configure it with actual JS code. Other contenders as of this writing are Webpack (quite popular!), Brunch, Browserify, and Broccoli. New tools with amusing names pop up on a regular basis. Some developers simply use Node.js-based scripts without using a task-runner program as a go-between. The point is, there are plenty of options.

You will find many online tutorials for learning how to download and configure the build tool of your choice when you are ready to automate your workflow. I hope that I have made you aware of the possibilities, and when a job interviewer mentions Grunt and Gulp, you'll know they aren't just suffering from indigestion.

Building Sites with Data and Templates

Throughout this book, we've been writing the HTML for our pages manually, wrapping tags around content elements in a logical source order. All of the content for the page is contained right there in the *.html* document. Of course, it is completely acceptable to build whole sites out of static web pages such as these, but in the real world—where sites might have thousands of pages with content tailored to individual users—a more robust solution is required.

It is more common these days to use a template system or framework to generate web pages from content stored as data. The templates use regular HTML markup, so everything you've learned so far will serve you well, but instead of specific content between the tags, special data markers are placed to pull in content from a database or data file.

There are a vast number of tool options for site generation, all of which are well beyond the scope of this book. However, as usual, I'd like to give you a taste of what the templating process might look like.

I once worked on a site that used a template tool called Handlebars (*handlebarsjs.com*) to pull content in from data files written in the YAML (*www.yaml.org/start.html*) language. These are just two options for doing this sort of thing. Let's look at a small example of how a template and data were used to assemble the web content shown in FIGURE 20-5.

FIGURE 20-5. A small portion of a speaker web page that was created with Handlebars and YAML.

Here is a small snippet of the data as it appears in the YAML (*.yml*) file:

```
speaker--name: "Jennifer Robbins"
speaker--description: "Designer, Author, ARTIFACT
Co-founder"
speaker--photo: "/img/speakers/jennifer-robbins.
jpg"
#HTML
speaker--biography: |
  <p>Jennifer has been designing for the web since
1993 when she worked on the first commercial web
site, GNN, from O'Reilly Media. Since then she has
gone on to write several books on web design for
O'Reilly, including <i>Web Design in a Nutshell
</i>, <i>Learning Web Design</i>, and the <i>HTML5
Pocket Reference</i>. More recently, Jennifer's
days are filled with organizing the ARTIFACT
Conference. …</p>
speaker--links:
  - link--label: "Website"
    link--target: "http://www.jenville.com"
    link--title: "jenville.com"
  - link--label: "Twitter"
    link--target: "http://www.twitter.com/jenville"
    link--title: "@jenville"
```

And here is the markup from the Handlebars template document, *speakers.hbs*. (I've edited it slightly for brevity.) If you look at the highlighted code, you see that instead of actual content, there are the same data labels used in the YAML file between curly brackets. (If you turn a curly bracket on its side, it looks like a handlebar mustache, thus the name!). Notice also that the template has markup for one label/link pair, but it loops through and displays all the **speaker--links** in the data file:

```
<div class="layout--container">
<div class="speaker--photo-container">
  <img src="{{page-data.speaker--photo}}">
</div>
<article class="speaker--content">
  <div class="speaker--biography">
    {{{page-data.speaker--biography}}}
  </div>
  <ul class="speaker--links">
    {{#each page-data.speaker--links}}
    <li class="speaker--link-item">{{link--label}}:
<a href="http://{{link--target}}" class="speaker--
link">{{link--title}}</a></li>
    {{/each}}
  </ul>
</article>
</div
```

This is just one example of how templates cut down on redundancy in markup. The Handlebars site (*handlebarsjs.com*) has a nice description of semantic templates right on the home page if you'd like more information on how it works.

Of course, browsers have no idea what to do with these file formats, so before the site can be published, it needs to be built or assembled, merging all the data into the template modules and all the modules into whole web pages. That is the job of scripts and build tools like the ones introduced in this section. Hopefully, this brief example gives you an inkling of how generated sites work.

VERSION CONTROL WITH GIT AND GITHUB

If you've done any sort of work on a computer, you've probably used some sort of system for keeping track of the versions of your work. You might have come up with a system of naming drafts until you get to the "final" version (and the "final-final" version, and the "final-final-no-really" version, and so on). You might take advantage of macOS's Time Machine to save versions that you can go back to in an emergency. Or you might have used one of the professional version control systems that have been employed by teams for decades.

The king of version control systems (VCS) for web development is a robust program called Git (*git-scm.com*). At this point, knowing your way around Git is a requirement if you are working on a team and is a good skill to have even for your own projects.

> **FUN FACT**
>
> Git was created by Linus Torvalds, the creator of the Linux operating system, when he needed a way to allow an enormous community to contribute to the Linux project.

In this section, I'll introduce you to the terminology and mental models that will make it easier to get started with Git. Teaching all the ins and outs of how to configure and use Git from the command line is a job for another book and online tutorials (I list a few at the end of the section), but I wish someone had explained the difference between a "branch" and a "fork" to me when I was starting out, so that's what I'll do for you.

We'll begin with a basic distinction: Git is the version control program that you run on your computer; GitHub (*github.com*) is a service that hosts Git projects, either free or for a fee. You interact with GitHub by using Git, either from the command line, with the user interface on the GitHub website, or using a standalone application that offers a GUI interface for Git commands. This was not obvious to me at first, and I want it to be clear to you from the get-go.

GitHub and services like it (see **Note**) are mainly web-based wrappers around Git, offering features like issue tracking, a code review tool, and a web UI for browsing files and history. They are convenient, but keep in mind that you can also set up Git on your own server and share it with your team members with no third-party service like GitHub involved at all.

NOTE

Beanstalk (beanstalkapp.com), GitLab (gitlab.com), and Bitbucket (bitbucket.org) are other Git hosting services aimed at enterprise-scale projects. GitLab has a free option for public projects, similar to GitHub, and because it is open source, you can host it yourself.

Why Use Git

There are several advantages to making Git (and GitHub) part of your workflow. First, you can easily roll back to an earlier version of your project if problems show up down the line. Because every change you make is logged and described, it helps you determine at which point things might have gone wrong.

Git also makes it easy to collaborate on a shared code source. You may tightly collaborate with one or more developers on a private project, merging all of

your changes into a primary copy. As an added benefit, the sharing process is a way to get an extra set of eyes on your work before it is incorporated. You may also encourage loose collaboration on a public project by welcoming contributions of people you don't even know in a way that is safe and managed. Git is a favorite tool for this type of collaboration on all sorts of open source projects.

Git is a favorite tool for collaboration on open source projects.

Getting up to speed with GitHub in particular is important because it's what everyone is using. If your project is public (accessible to anyone), the hosting is free. For private and commercial projects, GitHub charges a fee for hosting. In addition to hosting projects, they provide collaboration tools such as issue tracking. You may have already found that some of the links to tools I mentioned in this book go to GitHub repositories. I want you to know what you can do when you get there.

How Git Works

Git keeps a copy of every revision of your files and folders as you go along, with every change (called a commit) logged in with a unique ID (generated by Git), a message (written by you) describing the change, and other metadata. All of those versions and the commit log are stored in a repository, often referred to as a "repo."

Once you have Git installed on your computer, every time you create a new repository or clone an existing one, Git adds a directory and files representing the repo's metadata alongside other files in the project's folder. Once the Git repository is initialized, you can commit changes and take advantage of the "time machine" feature if you need to get back to an earlier version. In this way, Git is a good tool for a solo workflow.

More likely you'll be working with a team of other folks on a project. In that case, a hub model is used in which there is an official repository on a central server that each team member makes a local copy of to work on. Each team member works on their own machine, committing to their local repo, and at logical intervals, uploads their work back to the central repository.

That's what makes Git a distributed version control system compared to other systems, like SVN, that require you to commit every change directly to the server. With Git, you can work locally and offline.

The first part of mastering Git is mastering its vocabulary. Let's run through some of the terminology that will come in handy when you're learning Git and the GitHub service. FIGURE 20-6 is a simplified diagram that should help you visualize how the parts fit together.

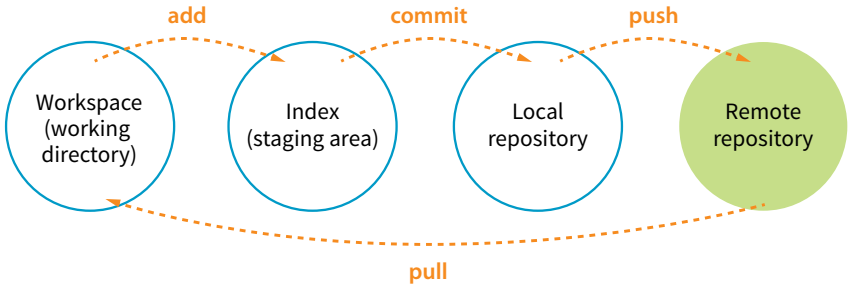


FIGURE 20-6. Visualization of Git structure.

Working directory

The working directory is the directory of files on your computer in which you do your actual work. Your working copy of a file is the one that you can make changes to, or to put it another way, it's the file you can open from the hard drive by using Finder or My Computer.

Repository

Your local Git repository lives alongside the files in your working directory. It contains copies, or snapshots, of all the files in a single project at every step in its development, although these are kept hidden. It also contains the metadata stored with each change. There may also be a central repository for the project that lives on a remote server like GitHub.

Commit

A commit is the smallest unit of Git interaction and the bulk of what you will do with Git. Git uses "commit" as a verb and a noun. You may "save" your working document frequently as you work, but you commit (v.) a change when you want to deliberately add that version to the repository. Usually you commit at a logical pause in the workflow—for example, when you've fixed a bug or finished changing a set of styles.

When you commit, Git records the state of all the project files and assigns metadata to the change, including the username, email, date and time, a unique multidigit ID number (see the **"Hashes"** sidebar), and a message that describes the change. These stored records are referred to as commits (n.). A commit is like a snapshot of your entire repository—every file it contains—at the moment in time you made the commit.

Commits are additive, so even when you delete a file, Git adds a commit to the stack. The list of commits is available for your perusal at any time. On GitHub, use the History button to see the list of commits for a file or folder.

The level of granularity in commits allows you to view the repository (project) at any state it's ever been at, ever. You *never* lose work, even as you proceed

Git Visualization Resources

Need more help picturing how all these pieces and commands work together? Try these visualization resources:

- The Git Cheatsheet from NDP Software provides a thorough interactive mapping of how various Git commands correspond to the workspace and local and remote repositories. It's worth checking out at *ndpsoftware.com/git-cheatsheet.html#loc=workspace;*.
- A Visual Git Reference (*marklodato.github.io/visual-git-guide/index-en.html*) is a collection of diagrams that demonstrate most common Git commands.
- "Understanding the GitHub Flow" (*guides.github.com/introduction/flow/*) explains a typical workflow in GitHub.

Hashes

The unique ID that Git generates for each commit is technically called a *SHA-1 hash*, more affectionately known as simply a *hash* in the developer world. It is a 40-character string written in hexadecimal (0–9 and A–F are used), so the odds of having a duplicate hash are astronomical. It is common to use short hashes on projects instead of the full 40 characters. For example, on GitHub, short hashes are seven characters long, and you'll see them in places like a project's Commits page. Even with just seven characters, the chances of collision are tiny.

further and further. It's a great safety net. Indirectly this also means that there's nothing you can do with Git that you can't undo—you can't ever get yourself into an impossible situation.

Staging

Before you can commit a change, you first have to make Git aware of the file (or to track it, to use the proper term). This is called staging the file, accomplished by adding it to Git. In the command line, it's **`git add`** *`filename`*, but other tools may provide an Add button to stage files. This creates a local index of files that you intend to commit to your local repository but haven't been committed yet. It is worth noting that you need to "add" any file that you've changed, not just new files, before committing them. Staging as a concept may take a little while to get used to at first because it isn't especially intuitive.

Branch

A branch is a sequential series of commits, also sometimes referred to as a stack of commits. The most recent commit on any given branch is the head (see **Note**). You can also think of a branch as a thread of development. Projects usually have a primary or default branch, typically (although not necessarily) called master, which is the official version of the project. To work on a branch, you need to have it checked out.

NOTE

There are exceptions, as it is possible to reorder commits; however, it is almost always true that the head commit is also the most recent.

When working in a branch, at any point you can start a new branch to do a little work without affecting the source branch. You might start a new branch to experiment with a new feature, or to do some debugging, or to play around with presentation. Branches are often used for small, specific tasks like that, but you can create a new branch for any purpose you want.

For example, if you are working on "master," but want to fix a bug, you can create a new branch off master and give the branch a new descriptive name, like "bugfix." You can think of the bugfix branch as a copy of master at the point at which bugfix was created (FIGURE 20-7), although that's not exactly what is happening under the hood.

To work on the bugfix branch, you first need to check it out (**`git checkout bugfix`**), and then you can go about your business of making changes, saving them, adding them to Git, and committing them. Eventually, the new branch ends up with a commit history that is different from the source branch.

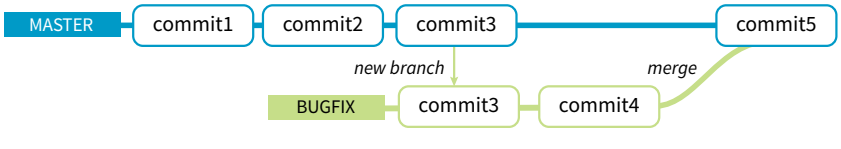


FIGURE 20-7. Creating and merging a new branch.

When you are done working on your new branch, you can merge the changes you made back into the source branch and delete the branch. If you don't like what's happening with the new branch, delete it without merging, and no one's the wiser.

Merging

Merging is Git's killer feature for sharing code. You can merge commits from one branch into another (such as all of the commits on a feature branch into master) or you might merge different versions of the same branch that are on different computers. According to the Git documentation, merging "incorporates changes from the named commits (since the time their histories diverged from the current branch) into the current branch." Put another way, Git sees merging as "joining two histories together," so it useful to think of merging happening at the commit level.

Git attempts to merge each commit, one by one, into the target branch. If only one branch has changed, the other branch can simply fast-forward to catch up with the changes. If both branches have commits that are not in the other branch—that is, if both branches have changes—Git walks through each of those commits and, on a line-by-line basis, attempts to merge the differences. Git actually changes the code inside files for you automatically so you don't have to hunt for what's changed.

However, if Git finds conflicts, such as two different changes made to the same line of code, it gives you a report of the conflicts instead of trying to change the code itself. Conflicts are pointed out in the source files between ======= and <<<<<<< characters (FIGURE 20-8). When conflicts arise, a real person needs to read through the list and manually edit the file by keeping the intended change and deleting the other. Once the conflicts are resolved, the files need to be added and committed again.

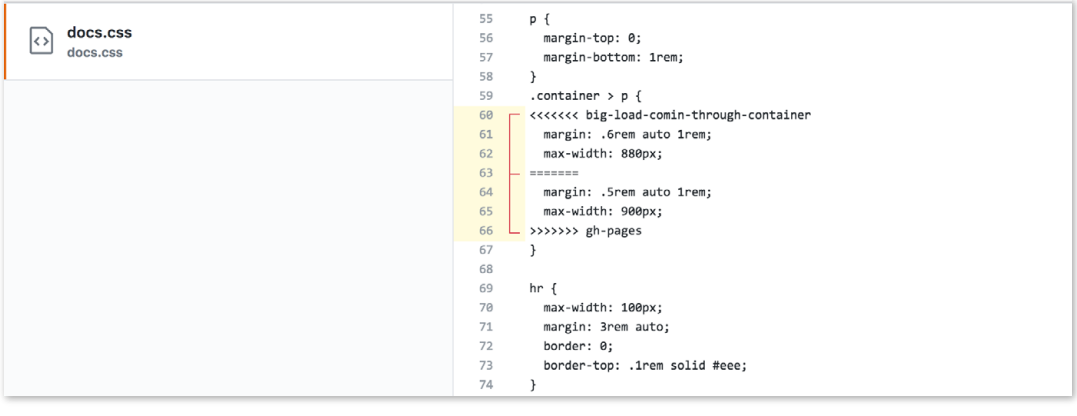


FIGURE 20-8. GitHub conflict report.

Remotes

All of the features we've looked at so far (commits, branches, merges) can be done on your local computer, but it is far more common to use Git with one or more remote repositories. The remote repo could be on another computer within your organization, but it is likely to be hosted on a remote server like GitHub. Coordinating with a remote repository opens up a few other key Git features.

Clone

Cloning is making an exact replica of a repository and everything it contains. It's common to clone a repo from a remote server to your own computer, but it is also possible to clone to another directory locally. If you are getting started on an existing project, making a clone of project's repo is a logical first step.

Push/pull

If you are working with a remote repository, you will no doubt need to upload and download your changes to the server. The process of moving data from your local repository to a remote repository is known as pushing. When you push commits to the remote, they are automatically merged with the current version on the server. To update your local version with the version that is on the server, you pull it, which retrieves the metadata about the changes and applies the changes to your working files. You can think of pushing and pulling as the remote version of merging.

■ GIT TIP

Always *pull* before you *push* to avoid conflicts.

It is a best practice to pull the remote master frequently to keep your own copy up-to-date. That helps eliminate conflicts, particularly if there are a lot of other people working on the code. Many GUI Git tools provide a Sync button that pulls and pushes in one go.

Fork

You may hear talk of "forking" a repo on GitHub. Forking makes a copy of a GitHub repository to your GitHub account so you have your own copy to play around with. Having the repo in your account is not the same as having a working copy on your computer, so once you've forked it, you need to clone (copy) it to your own computer (FIGURE 20-9).

NOTE

Forking is most often used for contributing to an open source project. For commercial or personal projects, you generally commit directly to the repository shared by your team.

People fork projects for all sorts of reasons (see **Note**). You might just want to have a look under the hood. You may want to iterate and turn it into something new. You may want to contribute to that project in the form of pull requests. In any scenario, forking is a safeguard for repository owners so they can make the project available to the public while also controlling what gets merged back into it.

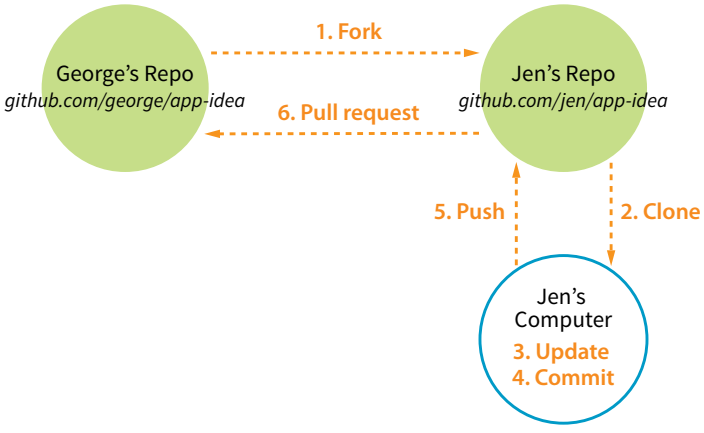


FIGURE 20-9. Once you fork a repository on GitHub, you need to clone it to get a local working copy. (Based on a diagram by Kevin Markham.)

Pull request

It is important to keep in mind that your forked copy is no longer directly connected to the original repository it was forked from. You will not be able to push to the original. If you come up with something you think is valuable to the original project, you can do what is called a pull request—that is, asking the owner to pull your changes into the original master.

You can also do a pull request for a repo that you have access to, not just one that you've forked. For example, if you've made a branch off the main project branch, you can do a pull request to get your team to review what you've done and give you feedback before merging your changes back in. In fact, pull requests may be used earlier in the process to start a discussion about a possible feature.

Git Tools and Resources

Most Git users will tell you that the best way to use Git is with the command line. As David Demaree says in his book *Git for Humans*, "Git's command-line interface is its native tongue." He recommends typing commands and seeing what happens as the best way to learn Git. The downside of the command line, of course, is that you need to learn all the Git commands and perhaps also tackle the command-line interface hurdle itself. The following resources will help get you up to speed:

- *Git for Humans* by David Demaree (A Book Apart) is a great place to start learning Git via the command line (or however you intend to use it!).
- *Pro Git* by Scott Chacon and Ben Straub (Apress) is available free online (*git-scm.com/book/en/v2*).

- "Git Cheat Sheet" from GitHub is a list of the most common commands (*services.github.com/on-demand/downloads/github-git-cheat-sheet.pdf*).
- The Git Reference Manual on the official Git site provides a thorough listing of commands and features (*git-scm.com/docs*).

There are also several graphical Git applications available for those who prefer icons, buttons, and menus for interacting with their repositories, and there's no shame in it. I know many developers who use a graphical app and Terminal side by-side, choosing the tool that most easily allows them to do the task they need to do. If you feel more comfortable getting started with a graphical Git tool, I recommend the following:

- GitHub Desktop (from GitHub) is free and available for Mac and Windows (*desktop.github.com*).
- Git Tower 2 (Mac and Windows) costs money, but it is more powerful and offers a thoughtfully designed interface, including visualizations of branches and merges (*www.git-tower.com*).

Many code editors have built-in Git support or Git/GitHub plug-ins as well.

If you go to the GitHub.com site, they do a good job of walking you through the setup process with easy-to-follow tutorials. You can set up an account and gain some basic GitHub skills in a matter of minutes. Their online documentation is top-notch, and they even have a YouTube channel with video tutorials aimed at beginners (*www.youtube.com/githubguides*).

And speaking of GitHub, for a good introduction to the ins and outs of the GitHub interface, I recommend the book *Introducing GitHub: A Non-Technical Guide* by Brent Beer (O'Reilly).

When you are ready to get started using Git for version control, you'll find all the support you need.

CONCLUSION

This concludes the web developer "power tools" chapter. We began with an introduction to the command line, and looked at some strong incentives for learning to use it. You can write CSS faster and make it more cross-browser compliant. You can take advantage of task runners and build tools that automate a lot of the repetitive grunt work you come across as a developer. Finally, although the command line is not required to use Git, it may make learning Git easier and will give you repo superpowers as you begin to master it.

We've talked a fair amount about JavaScript in this chapter. In **Part IV**, I hand over the keyboard to JavaScript master Mat Marquis, who will introduce you to JavaScript and its syntax (also somehow managing to make it very entertaining). I'll be back in **Part V** to talk about web images.

TEST YOURSELF

It's time to test your knowledge of the topics introduced in this chapter. See **Appendix A** for the answers.

1. In the computer world, what is a *shell*?

2. Why might you want to learn to use the command line?
   a. It is a good way to manipulate files and folders on your own computer.
   b. It is a good way to manipulate files and folders on a remote server.
   c. It is required for many useful web development tools.
   d. All of the above.

3. What is a *prompt*?

4. What would you expect to happen if you type `mkdir newsite` after a command-line prompt?

5. Name the two primary functions of CSS processors.

6. Name one advantage of learning Sass.

7. Name two features you might use a CSS postprocessor for.

8. What is a task (in relation to a build tool/task runner)?

9. What does “Grunt is watching this file” mean?

10. What makes Git a *distributed* version control system?

11. In Git, what does it mean if a file is *staged*?

12. What is the difference between a *branch* and a *fork*?

13. Why should you pull before you push?

14. What is a pull request?