IN THIS CHAPTER

- Understanding page flow
- Breaking out of the flow by floating elements
- Positioning elements on the page
- Stacking elements on top of each other

Chapter 1
Exploring Some Layout Basics

To dismiss basic contexts such as link colours, page layouts, navigation systems, and visual hierarchy as 'boring' or 'pedestrian' is akin to laughing at a car's steering wheel as unimaginative.

—JEFFREY VEEN

Why are some web pages immediately appealing, while others put the "Ugh" in "ugly"? There are lots of possible reasons: colors, typography, image quality, the density of exclamation points. For my money, however, the number one reason why some pages soar while others are eyesores is the overall look and feel of the page. We've all visited enough websites in our lives to have developed a kind of sixth sense that tells us immediately whether a page is worth checking out. Sure, colors and fonts play a part in that intuition, but we all respond viscerally to the "big picture" that a page presents.

That big picture refers to the overall layout of the page, and that's the subject you start to explore here in Book 5. In this chapter, you build a solid foundation by understanding how the web browser lays out a page by default, and then exploring a few basic CSS techniques that enable you to break out of that default layout and take control of your pages. By the time you're done mastering the nitty-gritty

of page layout, you'll be in a position to design and build beautiful and functional pages that'll have them screaming for more.

Getting a Grip on Page Flow

When a web browser renders a web page, one of the really boring things it does is lay out the tags by applying the following rules to each element type:

- **Inline elements:** Render these from left to right within each element's parent container.
- **Block-level elements:** Stack these on top of each other, with the first element at the top of the page, the second element below the first, and so on.

REMEMBER

These rules assume that the current language is one whose text reads from left to right and top to bottom (such as English). In some languages (such as Hebrew and Arabic), the default text flow is from right to left and top to bottom. In vertical languages, the default text flow is from top to bottom and then either right to left (as in Japanese and Chinese) or left to right (as in Mongolian).

This is called the *page flow*. For example, consider the following HTML code (refer to bk05ch01/example01.html in this book's example files):

```
<header>
    The page header goes here.
</header>
<nav>
    The navigation doodads go here.
</nav>
<section>
    This is the first section of the page.
</section>
<section>
    This is—you got it—the second section of the page.
</section>
<aside>
    This is the witty or oh-so-interesting aside.
</aside>
<footer>
    The page footer goes here.
</footer>
```

This code is a collection of six block-level elements — a `header`, a `nav`, two `section` tags, an `aside`, and a `footer` — and Figure 1-1 shows how the web browser renders them as a stack of boxes.

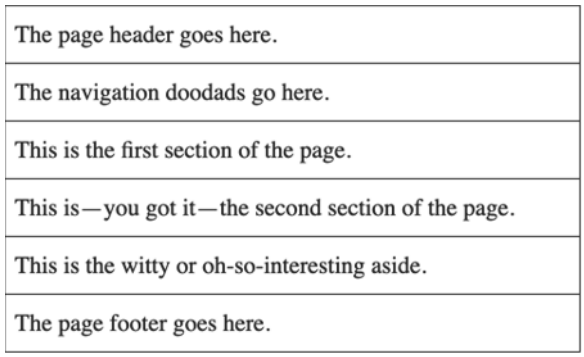

FIGURE 1-1: The web browser renders the block-level elements as a stack of boxes.

Nothing is inherently wrong with the default page flow, but having your web page render as a stack of boxes lacks a certain flair. Fortunately for your creative spirit, you're not married to the default, one-box-piled-on-another flow. CSS gives you a ton useful methods for breaking out of the normal page flow and giving your pages some pizzazz. In this chapter, you learn about three of those methods: floating, positioning, and stacking.

Floating Elements

When you *float* an element, the web browser takes the element out of the default page flow. Where the element ends up on the page depends on whether you float it to the left or to the right:

- **Float left:** The browser places the element as far to the left and as high as possible within the element's parent container.
- **Float right:** The browser places the element as far to the right and as high as possible within the element's parent container.

In both cases, the nonfloated elements flow around the floated element.

You convince the web browser to float an element by adding the `float` property:

```
element {
    float: left|right|none;
}
```

For example, consider the following code (check out bk05ch01/example02.html) and its rendering in Figure 1-2:

```
<header>
    <img src="images/notw.png" alt="News of the Word logo">
    <h1>News of the Word</h1>
    <h2>Language news you won't find anywhere else (for good
  reason!)</h2>
</header>
<nav>
    <a href="#">Home</a>
    <a href="#">What's New</a>
    <a href="#">What's Old</a>
    <a href="#">What's What</a>
    <a href="#">What's <em>That</em>?</a>
</nav>
```

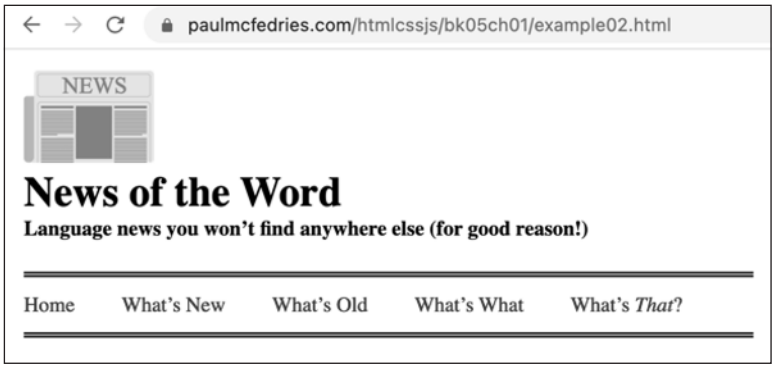

FIGURE 1-2: As usual, the browser displays the block-level elements as a stack of boxes.

In Figure 1-2, note that the web browser is up to its usual page-flow tricks: stacking all the block-level elements on top of each other. However, I think this page would look better if the title and subtitle (the `h1` and `h2` elements) appeared to the right of the logo. To do that, I can float the `img` element to the left (bk05ch01/example03.html):

```
header > img {
    float: left;
    margin-right: 2em;
}
```

Figure 1-3 shows the results. With the logo floated to the left, the title and subtitle — the `h1` and `h2` elements — now flow around (or, really, to the right of) the `img` element.

FIGURE 1-3: When the logo gets floated left, the title and subtitle flow around it.

Example: Creating a pull quote

A *pull quote* is a short excerpt copied ("pulled") from the current page text. The excerpt should be evocative or interesting, and the pull quote is set off from the regular text. A well-chosen and well-designed pull quote can entice an ambivalent site visitor to read (or, at least, start) the article.

You create a pull quote by copying the article excerpt and placing it inside an element such as an `aside`. You then float that element, most often to the right. Style the element as needed to make it stand apart from the regular text and you're done.

Here's an example (bk05ch01/example04.html):

HTML (partial):

```
<p>
    "None of it made a lick of sense" he said.
</p>
<aside class="pullquote">
    They can't understand a word anyone is texting to them.
</aside>
<p>
    It has long been thought that teen instant messages
  contained abbreviations (such as <i>LOL</i> for "laughing out
  loud" and <i>MAIBARP</i> for "my acne is becoming a real
  problem"), short forms (such as <i>L8R</i> for "later" and
  <i>R2D2</i> for "R2D2"), and slang (such as <i>whassup</i> for
  "what's up" and <i>yo</i> for "Hello, I am pleased to meet
  your acquaintance. Do you wish to have a conversation?").
  However, the report reveals that this so-called "teenspeak"
  began to change so fast that kids simply could not keep up.
  Each teen developed his or her own lingo, and the instant
  messaging system devolved into anarchy.
</p>
```

CSS:

```
.pullquote {
    border-top: 4px double black;
    border-bottom: 4px double black;
    float: right;
    color: hsl(0deg, 0%, 40%);
    font-size: 1.9rem;
    font-style: italic;
    margin: 0 0 0.75rem 0.5rem;
    padding: 8px 0 8px 16px;
    width: 50%;
}
```

Figure 1-4 shows how everything looks.

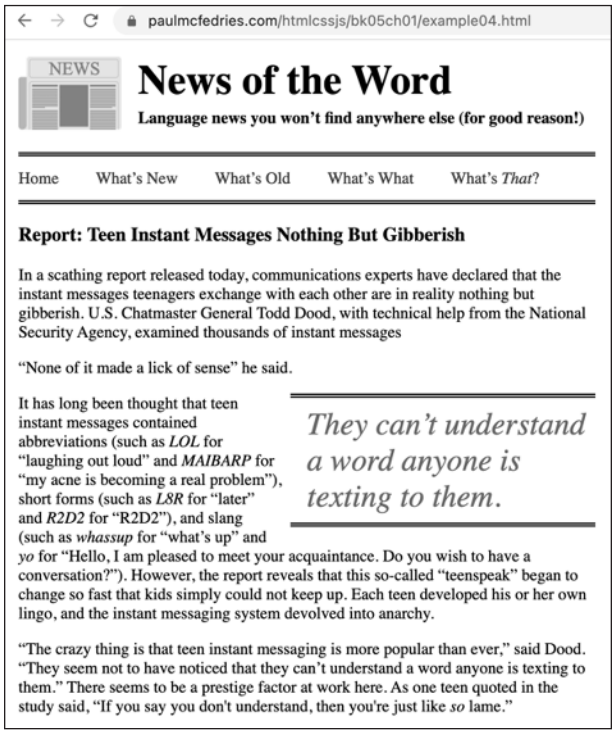

FIGURE 1-4: A pull quote floated to the right of the article text.

Clearing your floats

The default behavior for nonfloated stuff is to wrap around anything that's floated, which is often exactly what you want. However, you'll sometimes want to avoid having an element wrap around your floats. For example, consider the following code (bk05ch01/example05.html) and how it gets rendered, as shown in Figure 1-5.

```
<header>
    <h1>Can't You Read the Sign?</h1>
</header>
<nav>
    <a href="/">Home</a>
    <a href="semantics.html">Signs</a>
    <a href="contact.html">Contact Us</a>
    <a href="about.html">Suggest a Sign</a>
</nav>
<article>
    <img src="/images/keep-off-the-grass.jpg"  alt="A sign
  reading Keep Off the Grass beside a well-trodden path running
  across the grass.">
</article>
<footer>
    &copy; Can't You Read?, Inc.
</footer>
```

FIGURE 1-5: When the image is floated left, the footer wraps around it and ends up in a weird place.

With the `<img>` tag floated to the left, the rest of the content flows around it, including the content of the `<footer>` tag, which now appears by the top of the image.

You want your footer to appear at the bottom of the page, naturally, so how can you fix this? By telling the web browser to position the `footer` element so that it *clears* the floated image, which means that it appears after the image in the page flow. You clear an element by adding the `clear` property:

```
element {
    clear: left|right|both|none;
}
```

Use `clear: left` to clear all left-floated elements, `clear: right` to clear all right-floated elements, or `clear: both` to clear everything. When I add `clear: left` to the `footer` element (bk05ch01/example06.html), you can note in Figure 1-6 that the footer content now appears at the bottom of the page.

```
footer {
    clear: left;
}
```

FIGURE 1-6: Adding `clear: left` to the `footer` element causes the footer to clear the left-floated image and appear at the bottom of the page.

Collapsing containers ahead!

The odd behavior of CSS is apparently limitless, and floats offer yet another example. Consider the following HTML (bk05ch01/example07.html) and its result in Figure 1-7:

```
<article>
    <section>
        An awfully long time ago...
    </section>
    <aside>
        <b>Note:</b> Creating a new word by...
    </aside>
</article>
```

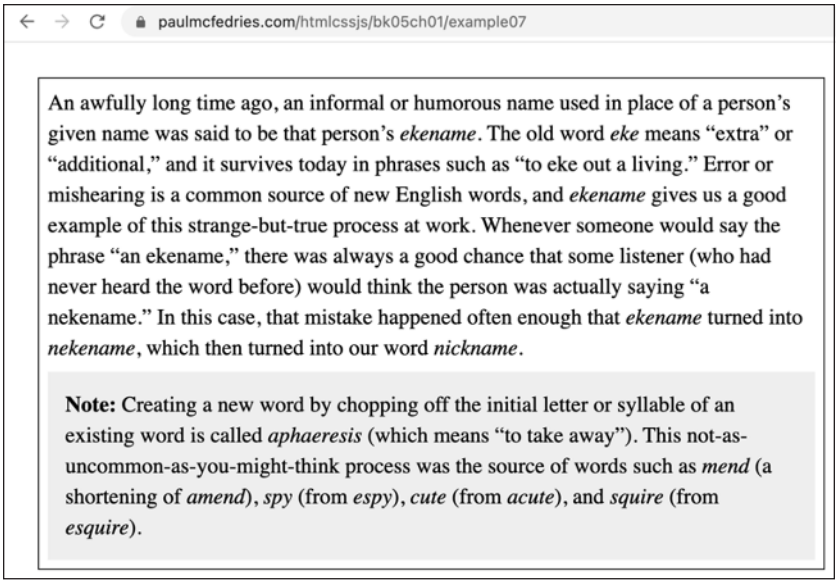

FIGURE 1-7: An `<article>` tag containing a `<section>` tag and an `<aside>` tag, rendered using the default page flow.

Note, in particular, that I've styled the `article` element with a border.

Rather than the stack of blocks shown in Figure 1-10, you may prefer to have the `section` and the `aside` elements appear side by side. Great idea! So, you add `width` properties to each, and float the `section` element to the left and the

aside element to the right. Here are the rules (bk05ch01/example08.html), and Figure 1-8 shows the result.

```
section {
    float: left;
    width: 25rem;
}
aside {
    float: right;
    width: 16rem;
}
```

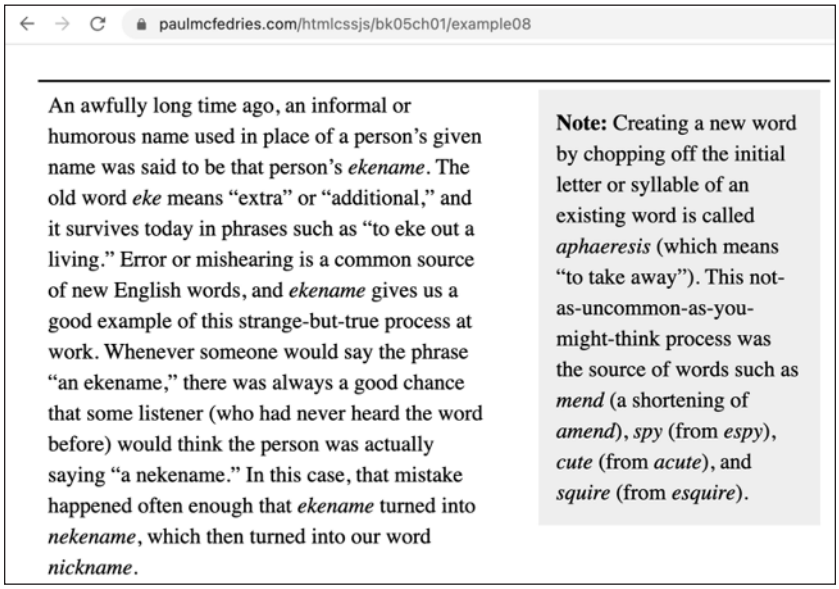

FIGURE 1-8: With its content floated, the <article> element collapses down to just its border.

Well, that's weird! The line across the top is what's left of the article element. What happened? Because I floated both the section and the aside elements, the browser removed them from the page flow, which made the article element behave as though it had no content at all. The result? A CSS bugaboo known as *container collapse.*

To fix this issue, you have to force the parent container to clear its own children.

HTML:

```
<article class="self-clear">
```

CSS:

```
.self-clear::after {
    content: "";
    display: block;
    clear: both;
}
```

The `::after` pseudo-element tells the browser to add an empty string (because you don't want to add anything substantial to the page), and that empty string is displayed as a block that uses `clear: both` to clear the container's children. It's weird, but it works, as shown in Figure 1-9 (bk05ch01/example09.html).

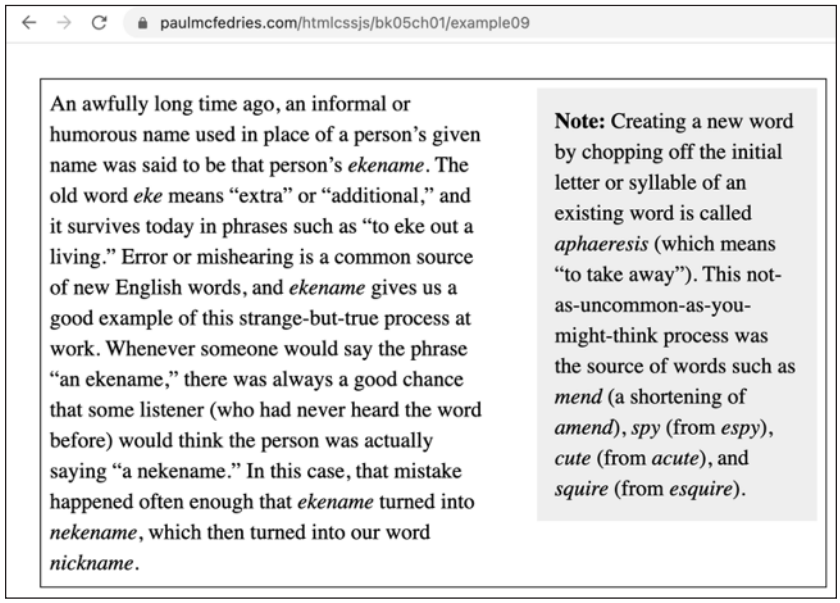

FIGURE 1-9: With the `self-clear` class added to the `<article>` tag, the `article` element now clears its own children and is no longer collapsed.

Positioning Elements

The second major method for breaking out of the web browser's default "stacked boxes" page flow is to position an element yourself using CSS properties. For example, you could tell the browser to place an image in the top-left corner of the window, no matter where that element's `<img>` tag appears in the page's HTML code. In the CSS world, this is known as *positioning*, and it's a very powerful tool, so much so that most web developers use positioning only sparingly.

The first bit of positioning wizardry you need to know is, appropriately, the `position` property:

```
element {
    position: static|relative|absolute|fixed|sticky;
}
```

- `static`: Places the element in its default position in the page flow.
- `relative`: Offsets the element from its default position while keeping the element in the page flow.
- `absolute`: Offsets the element from its default position with respect to its parent (or sometimes an earlier ancestor) container while removing the element from the page flow.
- `fixed`: Offsets the element from its default position with respect to the browser viewport while removing the element from the page flow.
- `sticky`: Starts the element with relative positioning until the element's parent crosses a specified offset with respect to the browser viewport (usually because the user is scrolling the page), at which point the element switches to fixed positioning. If the boundary of the element's parent block then scrolls to where the element is stuck, the element reverts to relative positioning and scrolls with the parent.

Because `static` positioning is what the browser does by default, I won't say anything more about it. For the other four positioning values — `relative`, `absolute`, `fixed`, and `sticky` — notice that each one offsets the element. Where do these offsets come from? From the following CSS properties:

```
element {
    top: top-value;
    right: right-value;
    bottom: bottom-value;
    left: left-value;
}
```

- `top`: Shifts the element down
- `right`: Shifts the element from the right
- `bottom`: Shifts the element up
- `left`: Shifts the element from the left

In each case, the value you supply is either a number followed by one of the CSS measurement units (such as `px`, `em`, `rem`, `vw`, or `vh`) or a percentage.

Using relative positioning

Relative positioning is a bit weird because not only does it offset an element relative to its parent container, but it still keeps the element's default space in the page flow intact.

Here's an example (bk05ch01/example10.html):

HTML:

```
<h1>
    holloway
</h1>
<div>
    <i>n.</i> A sunken footpath or road; a path that is enclosed
  by high embankments on both sides.
</div>
<img src="/images/holloway1.jpg" alt="Photo of a holloway">
<img src="/images/holloway2.jpg" alt="Photo of a holloway"
  class="offset-image">
<img src="/images/holloway3.jpg" alt="Photo of a holloway">
```

CSS:

```
.offset-image {
    position: relative;
    left: 200px;
}
```

The CSS defines a rule for a class named `offset-image`, which applies relative positioning and offsets the element from the left by `200px`. In the HTML, the `offset-image` class is applied to the middle image. As shown in Figure 1-10, not only is the middle image shifted from the left, but the space in the page flow where it would have appeared by default remains intact, so the third image's place in the page flow doesn't change. As far as that third image is concerned, the middle image is still right above it.

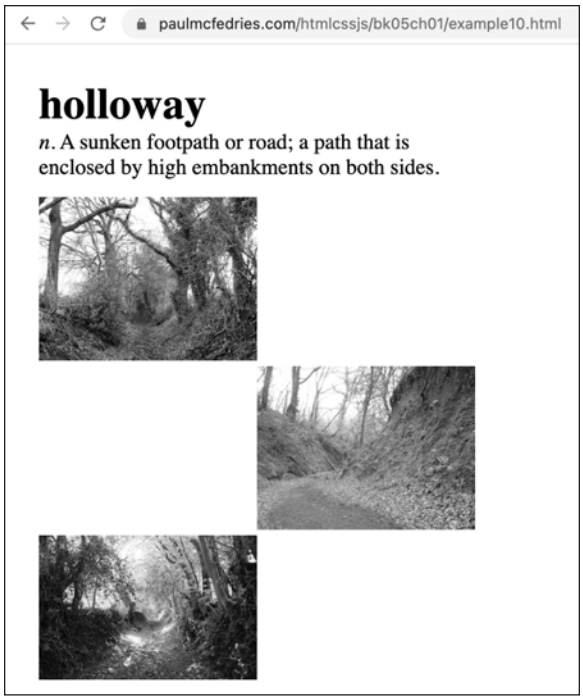

FIGURE 1-10: The middle image uses relative positioning to shift from the left, but its place in the page flow remains.

Giving absolute positioning a whirl

Absolute positioning not only offsets the element from its default position but also removes the element from the page flow. Sounds useful, but if the element is no longer part of the page flow, from what element is it offset? Good question, and here's the short answer: the closest ancestor element that uses nonstatic positioning.

If that has you furrowing your brow, I have a longer answer that should help. To determine which ancestor element is used for the offset of the absolutely positioned element, the browser goes through a procedure similar to this:

1. Move one level up the page hierarchy to the previous ancestor.
2. Check the `position` property of that ancestor element.
3. If the `position` value of the ancestor is `static`, go back to Step 1 and repeat the process for the next level up the hierarchy; otherwise (that is, if the `position` value of the parent is anything other than `static`), offset the original element with respect to the ancestor.
4. If, after going through Steps 1 to 3 repeatedly, you end up at the top of the page hierarchy — that is, at the `html` element — then use the `html` element to offset the element, which means in practice that the element is offset with respect to the browser's viewport.

I mention in the previous section that relative positioning is weird because it keeps the element's default position in the page flow intact. However, now that weirdness turns to goodness because if you want a child element to use absolute positioning, you add `position: relative` to the parent element's style rule. Because you don't also supply an offset to the parent, it stays put in the page flow, but now you have what CSS nerds called a *positioning context* for the child element.

I think an example would be welcome right about now (bk05ch01/example11.html):

HTML:

```
<section>
    <img src="images/new.png" alt="">
    <h1>
        holloway
    </h1>
    <div>
        <i>n.</i> A sunken footpath or road; a path that is
  enclosed by high embankments on both sides.
    </div>
    <div>
        There are two main methods that create holloways: By
  years (decades, centuries) of constant foot traffic that wears
  down the path (a process usually accelerated somewhat by water
  erosion); or by digging out a path between two properties and
  piling up the dirt on either side.
    </div>
</section>
```

CSS:

```
section {
    position: relative;
    border: 1px double black;
}

img {
    position: absolute;
    top: 0;
    right: 0;
}
```

In the CSS, the `section` element is styled with the `position: relative` declaration, and the `img` element is styled with `position: absolute` and `top` and `right` offsets set to `0`. In the HTML, note that the `<section>` tag is the parent of the `<img>` tag, so the latter's absolute positioning will be with respect to the former. With top and right offsets set to `0`, the image will now appear in the top-right corner of the section element and, indeed, it does, as shown in Figure 1-11.

WARNING

Because an absolutely positioned element now resides outside of the normal page flow, the element no longer abides by the default "rule" that no two elements should overlap. Therefore, you need to be careful when absolutely positioning an element to ensure that it doesn't accidentally end up on top of your page text or other elements.

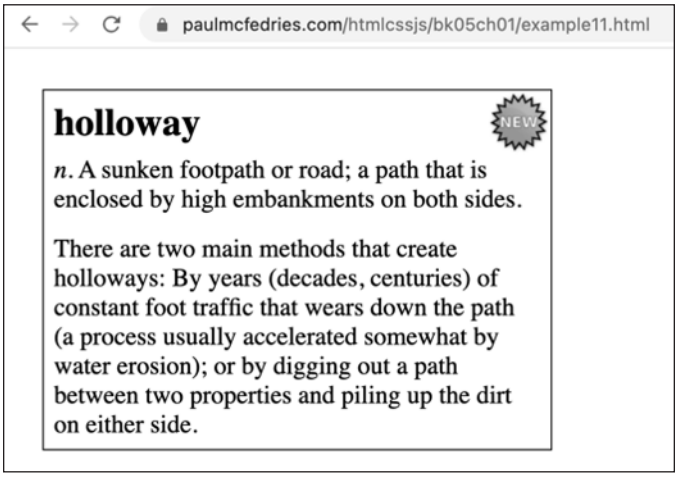

FIGURE 1-11: The `img` element uses absolute positioning to send it to the top-right corner of the `section` element.

Trying out fixed positioning

With *fixed positioning,* the element is taken out of the normal page flow and is then offset with respect to the browser's viewport, which means the element doesn't move, not even a little, when you scroll the page (that is, the element is "fixed" in its new position).

One of the most common uses of fixed positioning is to plop a header at the top of the page and make it stay there while the user scrolls the rest of the content.

Here's an example (bk05ch01/exmple12.html) that shows you how to create such a header:

HTML:

```
<header>
    <img src="images/holloway3.jpg" alt="Photo of a holloway">
    <h1>
        holloway
    </h1>
</header>
<main>
...
</main>
```

CSS:

```
header {
    position: fixed;
    top: 0;
    left: 0;
    width: 100%;
    height: 4rem;
    border: 1px double black;
    background-color: hsl(101, 38%, 63%);
}

main {
    margin-top: 4rem;
}
```

The HTML includes a `header` element with an image and a heading, followed by a longish `main` section that I don't include here for simplicity's sake. In the CSS code, the `header` element is styled with `position: fixed`, and the offsets `top` and `left` set to `0`. These offsets fix the header to the top left of the browser's viewport. I also added `width: 100%` to give the header the entire width of the window. Note, too, that I set the header `height` to `4rem`. To make sure that the `main` section begins below the header, I styled the `main` element with `margin-top: 4rem`. Figure 1-12 shows the results.

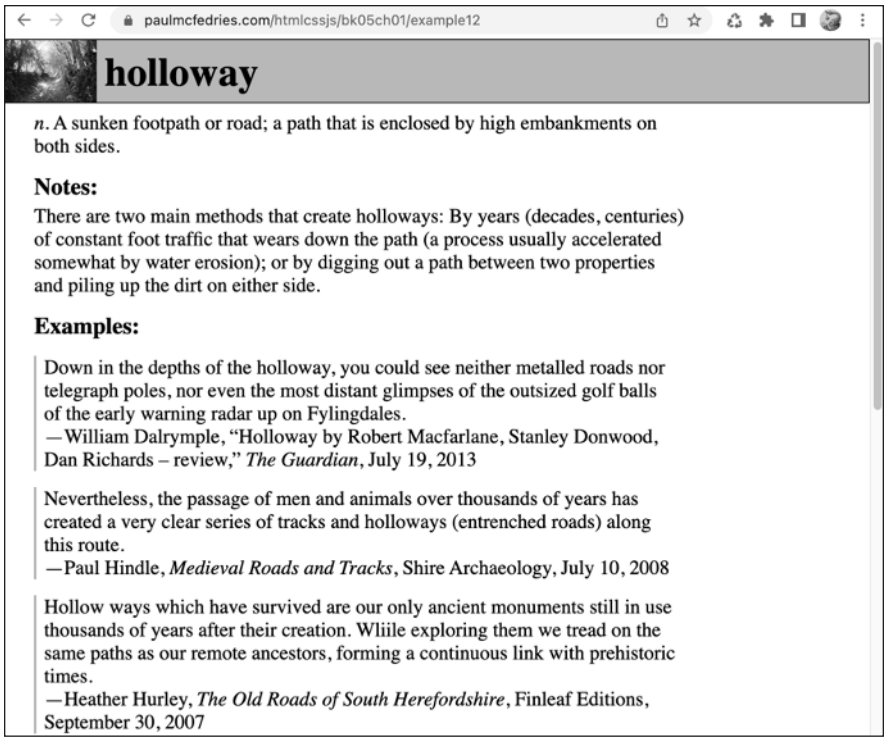

FIGURE 1-12: A page with the `header` element fixed to the top of the screen. When you scroll the rest of the page, the header remains where it is.

Making elements stick (temporarily)

Sticky positioning is a kind of combination of relative and fixed. That is, the element starts off with relative positioning until the element's containing block crosses a specified threshold (usually because the user is scrolling the page), at which point the element switches to fixed positioning. If the opposite edge of the element's containing block then scrolls to where the element is stuck, the element reverts to relative positioning and scrolls with the containing block.

For example, suppose your page has a `section` element, and inside that `section` is an `h2` element that you've positioned as sticky. Here's an abbreviated version of the code (check out bk05ch01/example13.html for the complete version):

HTML:

```
<section>
    <h2>Cat ipsum</h2>
    <p><a href="http://www.catipsum.com/">http://www.catipsum.
  com/</a></p>
    <p>Sample:</p>
```

```
    <p class="sample-text">
        Cat ipsum dolor sit amet, prance along on top of the
  garden fence, annoy the neighbor's dog and make it bark stuff
  and things intrigued by the shower. Please stop looking at
  your phone and pet me sleep everywhere, but not in my bed get
  my claw stuck in the dog's ear and adventure always but drool
  yet roll over and sun my belly. Ooh, are those your $250
  dollar sandals?
</p>
</section>
```

CSS:

```
h2 {
    position: sticky;
    top: 0;
}
```

In the CSS, notice that for the `h2` element, I've set `position: sticky`. To specify the threshold at which the element sticks, I've set `top: 0`, which means this element will stick in place when the top edge of the `section` element hits the top of the viewport.

Here's what happens when the user starts scrolling toward the bottom of the page:

1. At first, the `section` and `h2` elements scroll up together, as shown in Figure 1-13. Note that I've added an outline around the `section` element to make it easier for you to visualize its edges.
2. When the top edge of the `section` element hits the top of the viewport (because I set `top: 0` as the sticky threshold), the `h2` stops scrolling and "sticks" in place, as shown in Figure 1-14.
3. As the user keeps scrolling, the `section` content keeps scrolling up, as shown in Figure 1-15.
4. When the bottom edge of the `section` element reaches the bottom of the stuck `h2` element, the `h2` becomes "unstuck" (that is, it goes back to relative positioning) and resumes scrolling up with the `section`, as shown in Figure 1-16.

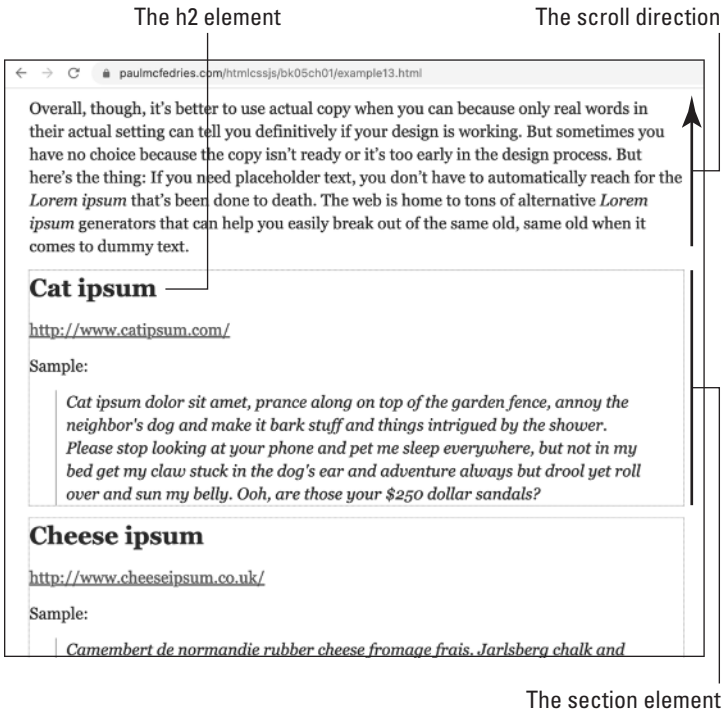

FIGURE 1-13: At first, the `section` and `h2` elements scroll up together.

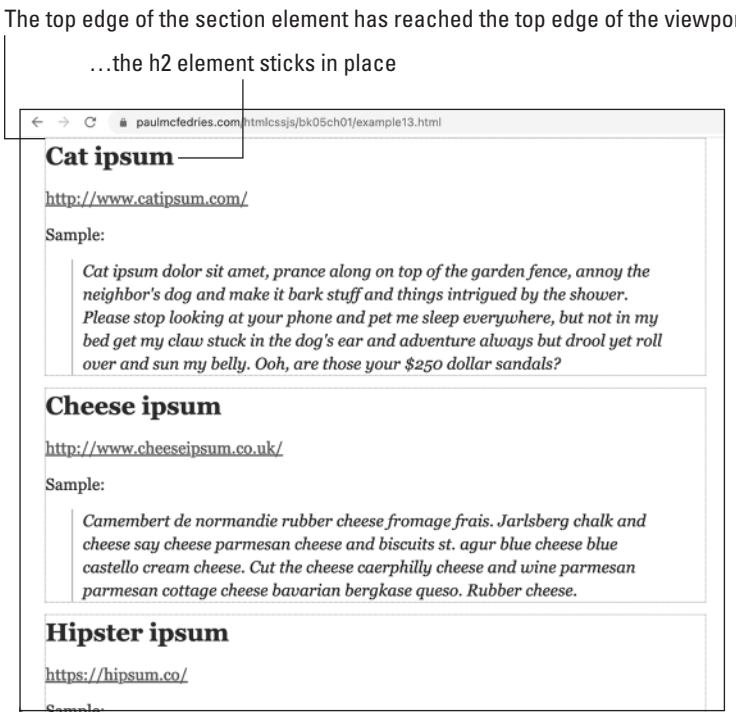

FIGURE 1-14: When the top of the `section` element hits the top of the viewport, the `h2` "sticks" in place.

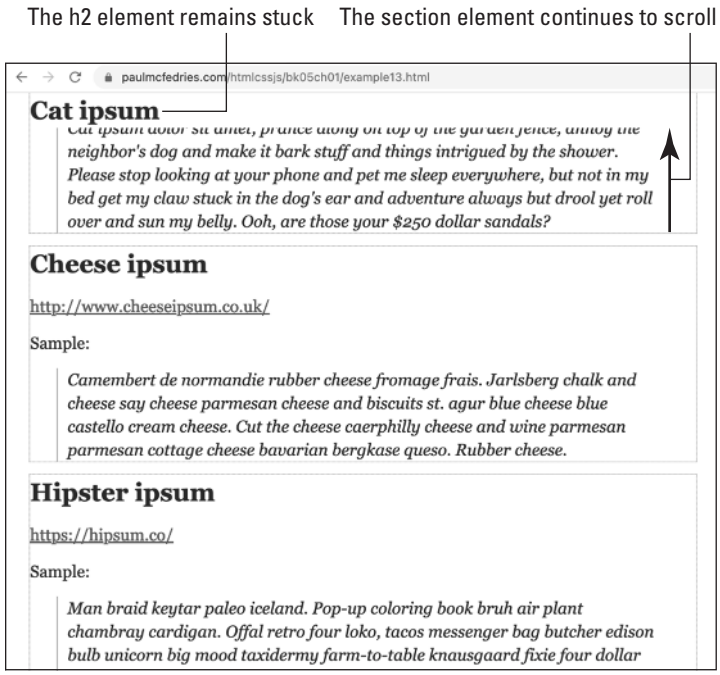

FIGURE 1-15: As you keep scrolling, the `section` content keeps scrolling up.

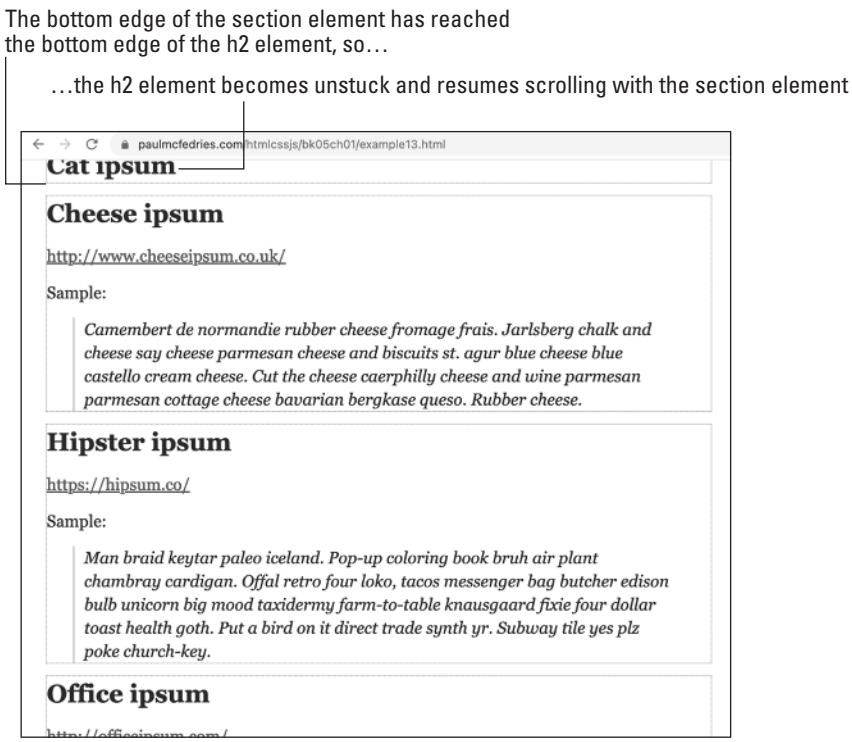

FIGURE 1-16: When the `section` bottom reaches the stuck `h2` element, the `h2` becomes "unstuck."

Stacking Elements

When you position an element using `position: fixed`, or `position: sticky`, as I describe in the previous section, a weird thing happens: When you scroll past the fixed or stuck element, the subsequent text and other page knickknacks slide *under* the element.

Similarly, if you position an element with `position: relative` or `position: absolute`, it's possible to place the element on the page (by manipulating the `top`, `right`, `bottom`, and `left` properties) so that it sits on *top* of some other elements.

How does the browser know which elements go on top of the other elements? The browser uses the following default layering:

1. The background and borders of the `html` element are rendered on the bottom layer.
2. All nonpositioned elements (that is, all elements where the `position` property is `static`) are placed on the next layer.
3. All positioned elements (that is, elements with a `position` value of `relative`, `absolute`, `fixed`, or `sticky`) are placed on subsequent layers in the order they appear in the HTML.

These layers represent the browser defaults, but CSS offers a way to layer stuff the way *you* want, which is the topic of the next section.

Layering elements with z-index

If you think back to high school geometry (my apologies if this is a painful ask), you'll recall the idea of the two-dimensional Cartesian plane where the x-axis represents horizontal values and the y-axis represents vertical values. You may also have come across the three-dimensional version that added a z-axis perpendicular to the plane representing points that are, relative to you as the observer, closer to you (positive) or farther away from you (negative).

This idea of having points closer to or farther away from you can be applied to the browser's rendering layers, with the page background farthest away, a nonpositioned element one layer closer to you, and a positioned element yet another layer closer to you.

CSS enables you to override the browser's default layers by setting up a *stack*, where elements on higher stack levels are rendered above elements on lower stack

levels. You specify an element's stack level by setting the `z-index` property on a positioned element (`z-index` has no effect on nonpositioned elements):

```
z-index: value;
```

» *value*: An integer that specifies the stack level. `0` (or `auto`) is the default level. A positioned element with a larger `z-index` value is rendered above a positioned element with a lower `z-index` value. Negative values are allowed.

Here's an example (bk05ch01/example14.html):

HTML:

```
<body>
    <div id="div1">
        <span>div1</span>
    </div>
    <div id="div2">
        <span>div2</span>
    </div>
</body>
```

CSS:

```
#div1 {
    position: relative;
    z-index: 2;
}
#div2 {
    position: relative;
    bottom: 100px;
    left: 100px;
    z-index: 1;
}
```

The HTML creates two `div` elements with ids `div1` and `div2`. In the CSS, `div2` is positioned relatively and shifted up by `100px` and to the left by `100px`. Because `div2` comes after `div1` in the HTML, `div2` should appear on top of `div1` by default. However, I set `z-index: 2` on `div1`, which is higher than the `z-index: 1` declared on `div2`, so `div1` now appears on top of `div2`, as shown in Figure 1-17.

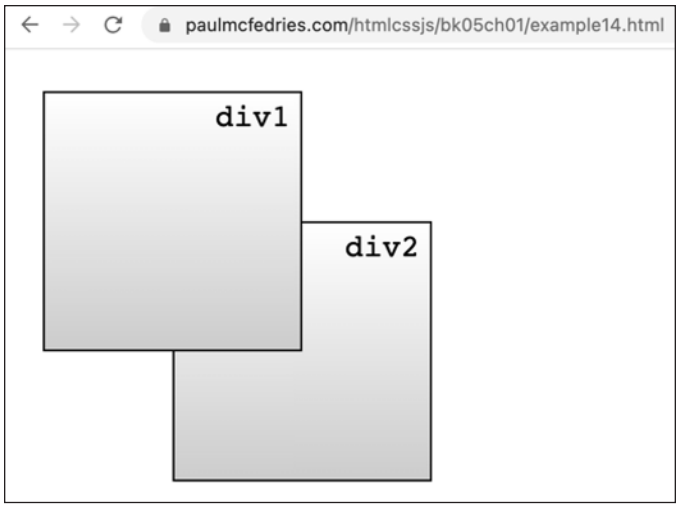

FIGURE 1-17: The element `div1` now appears on top of `div2` because its `z-index` value (2) is higher.

Getting your head around stacking contexts

The general idea that an element with a higher `z-index` value gets rendered on top of an element with a lower `z-index` value seems pretty straightforward. Ah, but here be dragons! To learn how `z-index` can get mightily weird, here's a look at some code (bk05ch01/example15.html):

HTML:

```
<body>
    <div id="div1">
        <span>div1</span>
    </div>
    <div id="div2">
        <span>div2</span>
        <aside>
            <span>aside</span>
        </aside>
    </div>
</body>
```

CSS:

```
#div1 {
    position: relative;
    z-index: 2;
}
```

```
#div2 {
    position: relative;
    bottom: 100px;
    left: 100px;
    z-index: 1;
}
aside {
    position: relative;
    top: 25px;
    left: 80px;
    z-index: 3;
}
```

This is the same code as in the previous section, except for two things:

- The HTML adds an `aside` child to the second `div` element.
- The `aside` CSS positions the element relatively, adds vertical and horizontal offsets, and sets the `z-index` property to 3.

The `aside` now has the highest `z-index` value of the three elements, so you'd expect that the `aside` would get rendered on top of everything. Figure 1-18 shows what actually happens.

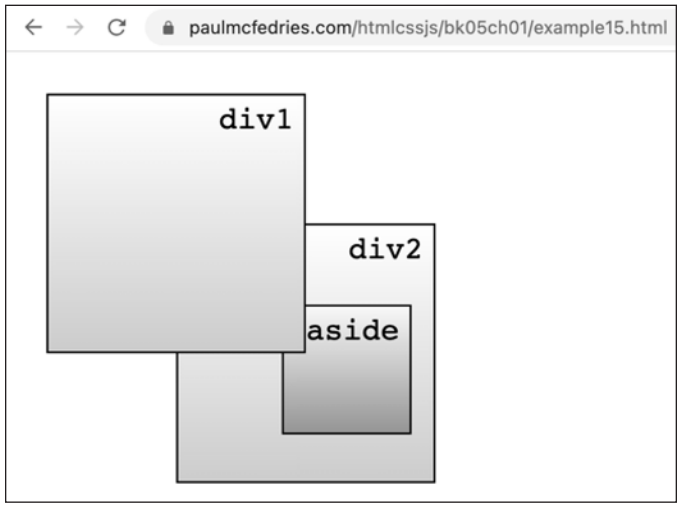

FIGURE 1-18: The new `aside` element appears behind the `div1` despite having a higher `z-index` value (3).

Wait, what!? The browser is rendering the `aside` element, which has a `z-index` value of `3`, *behind* `div1`, which has a `z-index` value of `2`! How can that be?

To understand what's going on here, you need to become fast friends with a concept known as the *stacking context*, which is the ability of a parent element's children to be stacked on top of each other.

The default stacking context is created by the `html` element, and within this stacking context the following rules apply:

- Nonpositioned elements are rendered at the bottom of the stacking order.
- Positioned elements are rendered above the nonpositioned elements in stack levels that reflect the order in which the positioned elements appear in the HTML.
- Positioned elements can use `z-index` to move up or down in the stacking order.

All this would be no big deal if you had just the one stacking context to worry about. Ah, if only life on Planet CSS were that simple! In fact, CSS creates *new* stacking contexts whenever either one of the following is true:

- An element uses `position: relative` or `position: absolute` and sets its `z-index` value to anything other than `auto`.
- An element uses `position: fixed` or `position: sticky`.

TECHNICAL STUFF

There are actually a *lot* more scenarios in which a stacking context is created. To eyeball the complete list, check out `https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Positioning/Understanding_z_index/The_stacking_context`.

You have two main points to take away from all this:

- When an element creates a new stacking context, any `z-index` values you apply to positioned children or descendants of the element will be relative *only* to other children and descendants of the element. In other words, within a stacking context, `z-index` values can't "see" outside that context.
- In the overall stack order of the page, the positioned children or descendants of any element that creates a stacking context can never be placed higher than that element's stack level.

Take look at the code once again, but now with fresh eyes:

HTML:

```
<body>
    <div id="div1">
        <span>div1</span>
    </div>
    <div id="div2">
        <span>div2</span>
        <aside>
            <span>aside</span>
        </aside>
    </div>
</body>
```

CSS:

```
#div1 {
    position: relative;
    z-index: 2;
}
#div2 {
    position: relative;
    bottom: 100px;
    left: 100px;
    z-index: 1;
}
aside {
    position: relative;
    top: 25px;
    left: 80px;
    z-index: 3;
}
```

The `div2` element uses `position: relative` and `z-index: 1`, so it creates a stacking context. The `aside` element is a child of the `div2` element, so its declaration of `z-index: 3` is relative only within the `div2`. And because the `div2` is declared with `z-index: 1`, the `aside` can never go higher than that in the overall page stack order. That's why the aside appears behind the `div1` element in Figure 1-18.