CHAPTER 10
A ONE-COLUMN CSS LAYOUT

This chapter forms the next stage of our journey to build a well-structured and well-styled web page. As in the previous chapters, we'll be working on our King Kong page and walking you through a variety of practical and hands-on CSS techniques.

In this chapter we'll be adding some additional structure to our document by breaking our King Kong page down into sections or divisions. We'll introduce and use `div` elements as the basis for creating a simple one-column CSS layout. An understanding of how `div` elements work is an important part of your Web Standardistas' journey and will form the foundation for creating more complicated layouts in CSS, so we urge you to pay attention throughout this chapter.

Along the way we'll introduce adding margins, borders, and padding, a means of adding borders and space around your elements and, as a byproduct of that, we'll introduce the CSS box model. We'll also look at the humble `span`, the inline sibling of the `div`, as well as `id`s and `class`es, attributes that allow you to identify and classify individual or groups of elements to easily target them with styles.

Finally, we'll look at CSS background images and how they can be used as a simple but effective means of ensuring your designs are just a little bit more interesting.

We know you're eager to get started and to get your hands dirty, but first, some short messages.

The Cascade in Cascading Style Sheets

Before we get on to the juicy part of this chapter, it's important to introduce a few more aspects of how CSS works. We're reaching a point at which our style sheets can get quite complex, and as a consequence, conflicting CSS rules might have an impact on how our elements display.

Allow us to digress for a moment while we look in a little more depth at the cascade in Cascading Style Sheets. We'll introduce how the cascade works; we'll also take a look at what happens when your CSS contains more than one rule targeting the same element and how the browser resolves these issues.

So what exactly is the cascade?

We've talked a lot about Cascading Style Sheets, but we've yet to touch on the actual topic of **cascading**. As your pages grow in complexity, especially as you start to use internal and external style sheets in combination during the development process, an understanding of the cascading aspect of CSS becomes important.

So what exactly is the cascade? The cascade is how CSS resolves conflicts between different styles, for example, when more than one rule is applied to the same element. The best way to explain how the cascade works is to show you an example.

We mentioned earlier in the book that we'd be working on an *internal* style sheet until we had our design complete. We also mentioned that—when the time was right and we had

the style sheet we wanted—we would remove this from the XHTML page itself to create an *external* style sheet that we would link to. We also mentioned the browser's *default* style sheet in Chapter 9. Lastly, we mentioned *inline* styles in Chapter 8, where the various styles are applied within the body of the XHTML page itself, directly to the elements we're styling.

This gives us a total of four potential sources of style for the elements on a page. So what if we have multiple rules targeting the same element? What wins out?

Imagine this scenario: we have a very simple web page with just a single `h1` on it. Unlikely, but for the purpose of this exercise, a little easier to follow.

As you know, the browser's default style sheet defines default styles for all the elements on our pages, including our solitary `h1`. As you saw in the chapters covering XHTML, the browser's default style sheet sets our `h1`'s `color` to `black` by default. However, what if we also have an external style sheet with a rule targeting our `h1`, setting its `color` to `red`? (We know we haven't told you how to create an external style sheet yet, but the principle remains.) As if this weren't confusing enough for the poor `h1`, let's add an internal style sheet too, with a rule for our `h1` setting its `color` to `green`. Lastly, let's apply a style directly to the `h1` itself using an inline style, setting its `color` to `blue`.

So, our lone `h1` is being styled left, right, and center (and from above too). Which style takes precedence? What color will the `h1` display in? The answer is `blue`, because the inline style wins. Let's find out why.

First, you need to get your head around the concept that before displaying your page, a browser gathers all of the different styles together; these styles then cascade into a new "virtual" style sheet combining all the different styles as visualized in Figure 10-1.

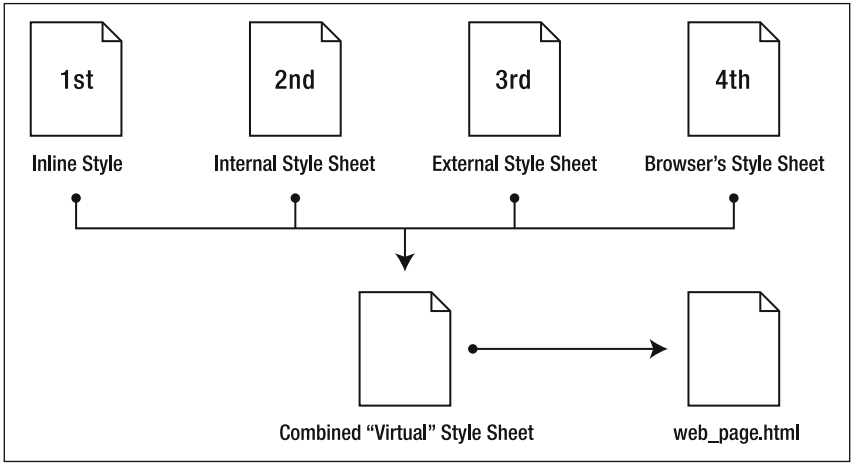

Figure 10-1. The browser gathers together the different styles, identifying which rules win in the event of any conflicts, before applying them to our web page.

Which styles "win out" in this new, "virtual" style sheet depends upon a set of complicated rules. We've simplified them here to make things a little easier to understand. The winners, in order, are as follows:

1. Inline styles
2. Internal style sheet (This is the style sheet you're currently using, situated inside the head element.)
3. External style sheet
4. Browser's default style sheet

So, internal style sheets win over external style sheets. Both take precedence over the browser's default style sheet. Inline styles—on the rare occasions you might use them—overrule everything. You can see this in action by looking at the source code of a page we've created for you to demonstrate how our solitary h1 is styled, at the book's companion web site:

```
www.webstandardistas.com/10/cascade.html
```

In fact, the rules of the cascade are a little more complicated, but this, in essence, is it. Although you're not using external style sheets yet, you might witness a potential clash between your inline style sheet and the browser's default style sheet. Hopefully an awareness of this will save you tearing your hair out when things are appearing to "go wrong" and you can't work out why.

The order of your CSS rules is important

As your style sheets get longer and more complicated, it's easy to lose track of what you've styled and find yourself writing additional rules lower in your style sheet that clash with earlier rules you've previously written. Take a look at the following example:

```
body
{
font-family: 'Lucida Grande', 'Lucida Sans', Arial, sans-serif;
font-size: 14px;
}

p
{
color: red;
}

/* Imagine another few dozen additional rules here. */

p
{
color: blue;
}
```

You've written a rule for your p elements to display all paragraphs in red as you first intended. However, after writing a few dozen additional rules, you've written another rule for your p elements to display all paragraphs in blue. Perhaps you forgot about writing the first; perhaps someone interrupted your train of thought, and it just slipped in by mistake.

So now there are two conflicting CSS rules targeting your p elements. What color will your paragraphs display in? The answer is blue. The browser takes the rule lowest in the style sheet and uses it to display your paragraphs. *The later a rule appears in a style sheet, the more weight it is given.*

Your style sheets might not be complex now, but they eventually will be, and you'd be surprised how often this happens. When an element doesn't display as you intended, it's worth spending a few minutes checking for duplicate rules in your style sheet(s) as this can often be the source of problems.

Introducing margins, borders, and padding

In Chapter 9 you saw how we could use line-height to give our paragraphs a little more spacing between the lines of text to aid legibility. We can also put space *around* our elements, using margins, borders, and padding to create space *between* our different elements. As we move toward creating more complicated layouts for our King Kong page, an understanding of how the margin, border, and padding properties work is important.

So far we've looked at styling elements in isolation, adding some line-height, and styling the typography of the different elements. In this section we look at how margins, borders, and padding are applied in CSS, affecting the relationship of our elements to each other. An understanding of this will form the basis for creating CSS layouts.

Meet the box model

Before we embark on a walkthrough, adding margins, borders, and padding to a simple paragraph to show how they affect a typical element, we need to cover a little theory.

You already know that all elements on a web page are treated as boxes—some are block-level, some are inline-level (we introduced this concept in Chapter 3). Each of these boxes is comprised of a content area and optional margins, borders, and padding. Up until this point margins, borders, and padding have been set by the browser's default style sheet; however, we can explicitly set them using CSS, overriding these defaults and specifying sizes we'd prefer.

The relationship of an element to its margins, borders, and padding is known as the **box model**. An understanding of the box model and how it works is crucial as we move forward to cover CSS layouts, so, without further ado, let's meet the box model.

Figure 10-2 illustrates the relationship between an element and any added margin, border, and padding. As you can see, the padding sits between the edge of the element and any

border added to it; the margin sits between any border added to an element and any adjacent elements.

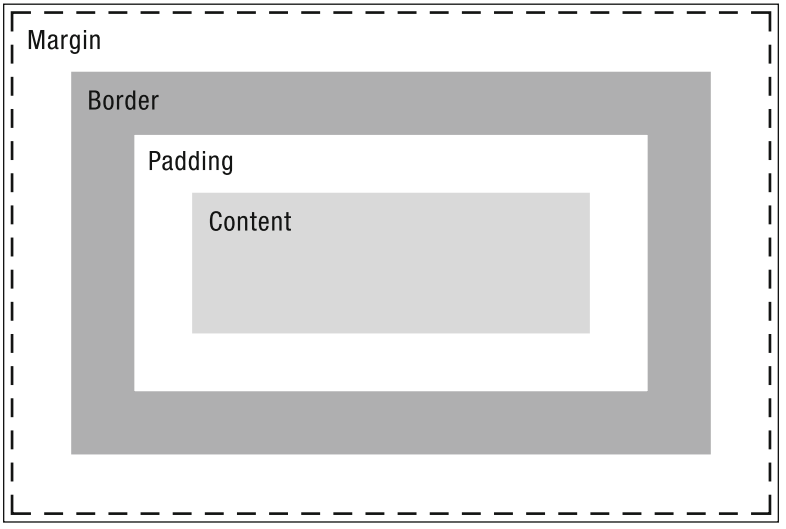

Figure 10-2. The W3C box model, showing the relationship of an element's content to its margins, borders, and padding

When `margin`, `border`, and `padding` properties are all specified, the width of our element—or the space that it occupies within the browser window—is as follows:

```
margin-left + border-left + padding-left + element  width + padding-right +
border-right + margin-right
```

We'll see this in action in the following section when we take a typical element, a paragraph, and add margins, borders, and padding to it, demonstrating the effect that this has on an element within the context of a browser window.

Applying margins, borders, and padding

Now that we've introduced you to the box model and margins, borders, and padding, we need to look at how these are applied to our elements. In this section we add `margin`, `border`, and `padding` declarations to a single `p` element and demonstrate their effect within a browser. This will give you an idea of how an element's margins, borders, and padding relate to each other and how you can use them together to structure your web page.

To start with we've created a very short web page with a single paragraph. We've written the following two CSS rules:

```
body
{
font-family: 'Lucida Grande', 'Lucida Sans', Arial, sans-serif;
font-size: 14px;
```

```
color: #000000;
background-color: #FFFFFF;
}

p
{
width: 400px;
line-height: 1.5;
}
```

In the preceding rules we've applied some basic styling to a very simple paragraph. We've set a `width` on our sole `p` element of `400px` to allow us to see the paragraph in context within a browser window and to see how adding margins, borders, and padding affects the overall width of our element within the browser. This renders in a browser as shown in Figure 10-3.

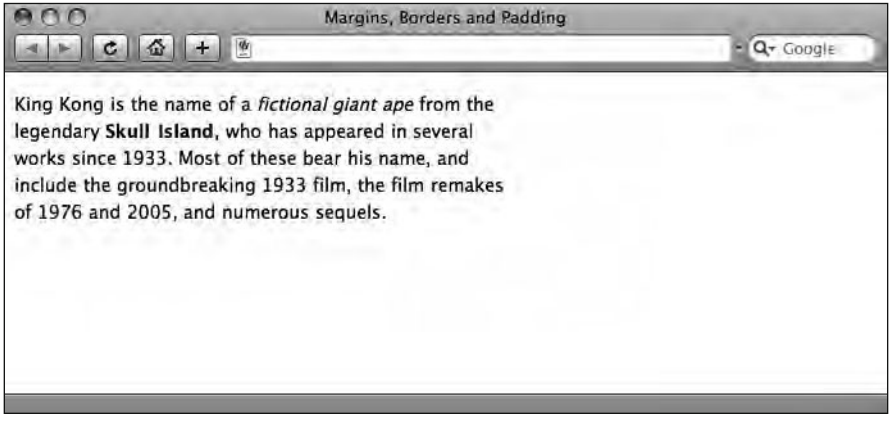

Figure 10-3. A simple paragraph with no `margin`, `border`, or `padding` declarations added

To highlight the block-level nature of our paragraph element, and to enable us to see the effect of adding margins, borders, and padding, we add a declaration to our `p` rule setting the paragraph's `background-color` to display in light gray, as follows:

```
p
{
width: 400px;
line-height: 1.5;
background-color: #CCCCCC;
}
```

The result of this is shown in Figure 10-4.

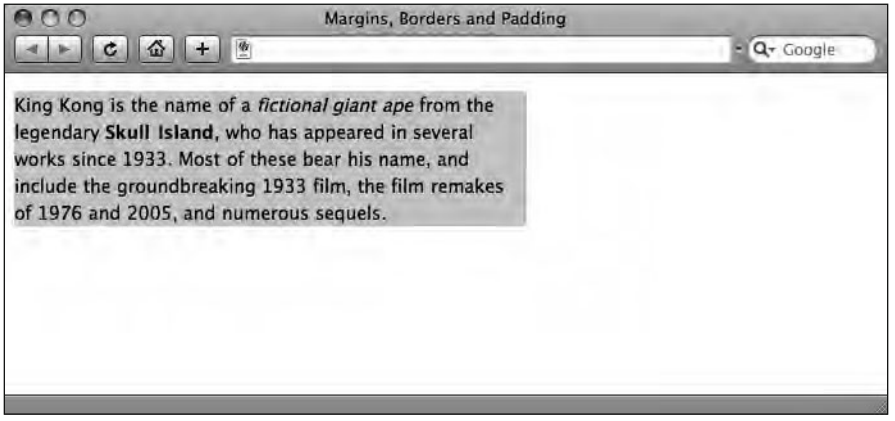

Figure 10-4. Setting a `background-color` on our paragraph reveals its underlying block-level nature.

Now it's time to add some margins; before we do that, however, we'll begin by removing some margins (this might sound odd, but all will be revealed in a moment).

As you can see in Figure 10-4, our browser's default style sheet is already applying some default `margin` to both our `body` element and our block-level `p` element, adding space above it and to the left of it (there's also space below it and to the right of it, but to all intents and purposes, it's invisible in this example). This is the default `margin` that the browser applies to our `p` and `body` elements in the event of no other style sheet specifying a different amount.

We'll remove the browser's default margin by setting the `margin` on the `body` and `p` elements to `0`. We do this by adding the following two declarations to our existing `body` and `p` rules:

```
body
{
font-family: 'Lucida Grande', 'Lucida Sans', Arial, sans-serif;
font-size: 14px;
color: #000000;
background-color: #FFFFFF;
margin: 0;
}

p
{
width: 400px;
line-height: 1.5;
background-color: #CCCCCC;
margin: 0;
}
```

Removing the browser's default `margin` results in the paragraph appearing as shown in Figure 10-5.

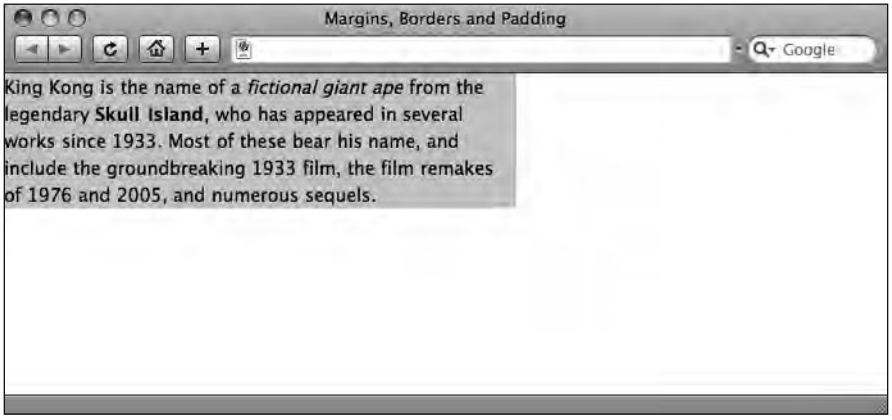

Figure 10-5. Resetting the `margin` on the `body` and `p` elements removes any default `margin` from the paragraph that would otherwise be applied by the browser's default style sheet.

By removing the browser's default margins, our paragraph now sits tight to the top left corner of the browser window. Now we're ready to start adding some margins, borders, and padding of our own and see how they affect the paragraph.

We amend the rule styling our `p` element, changing the value of our `margin` declaration and setting it to `40px`:

```
p
{
width: 400px;
line-height: 1.5;
background-color: #CCCCCC;
margin: 40px;
}
```

What this does is set the margin *on all four sides* of the paragraph to 40 pixels as shown in Figure 10-6. Although we've only written `40px` once, CSS shorthand is setting this on all four sides of the `p` element. (Rest assured, we'll be introducing you to the CSS shorthand for margins, borders, and padding shortly.)

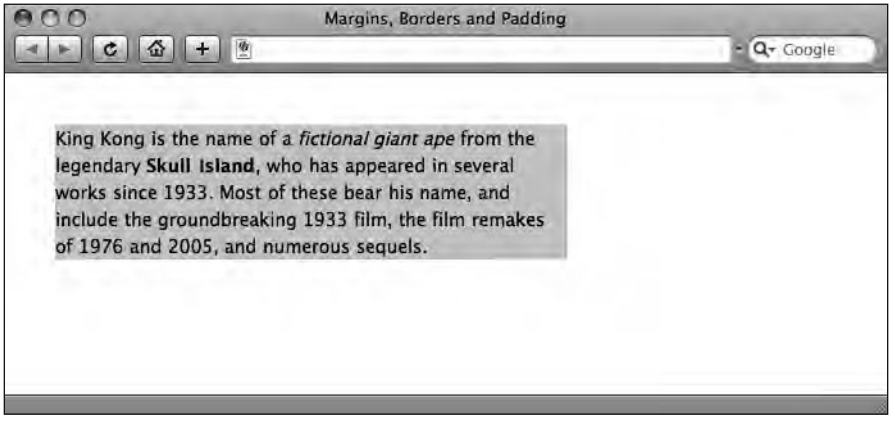

Figure 10-6. Adding 40px of margin to our paragraph moves the paragraph 40px down and 40px to the right. Although you can't see it, 40px of margin has also been added to the right edge and bottom edge of the paragraph.

Now it's time to add a `border` to our paragraph element. We introduced you to the `border` property in Chapter 9, when we added a border to our links, so the following additional declaration shouldn't be new to you:

```
p
{
width: 400px;
line-height: 1.5;
background-color: #CCCCCC;
margin: 40px;
border: 10px solid #666666;
}
```

Adding the `border` declaration results in our paragraph appearing as you see in Figure 10-7.

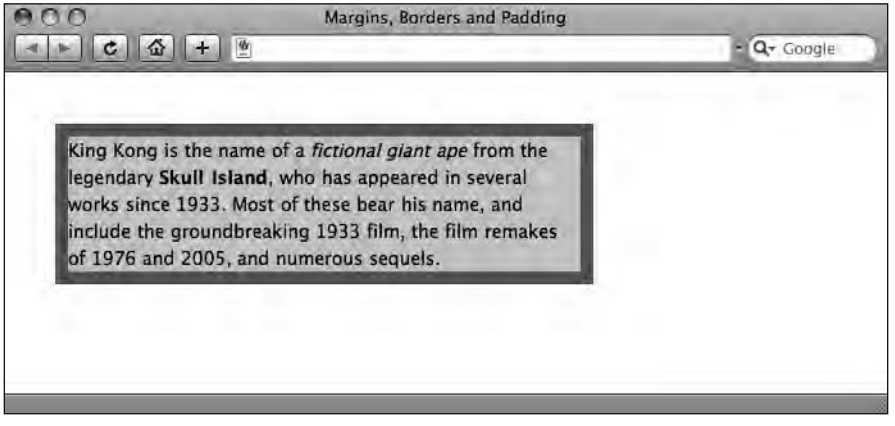

Figure 10-7. Adding a 10px dark gray border to the paragraph

Essentially we're setting a border on the paragraph to be 10px wide, solid, and a darker shade of gray. We now have a border around our paragraph; however, the text of the paragraph is sitting tight to the edge of its block-level box and sitting tight toward the border. It's time to add some padding.

We add a new declaration specifying a value for our padding as follows:

```
p
{
width: 400px;
line-height: 1.5;
background-color: #CCCCCC;
margin: 40px;
border: 10px solid #666666;
padding: 20px;
}
```

Adding this rule results in the layout shown in Figure 10-8.

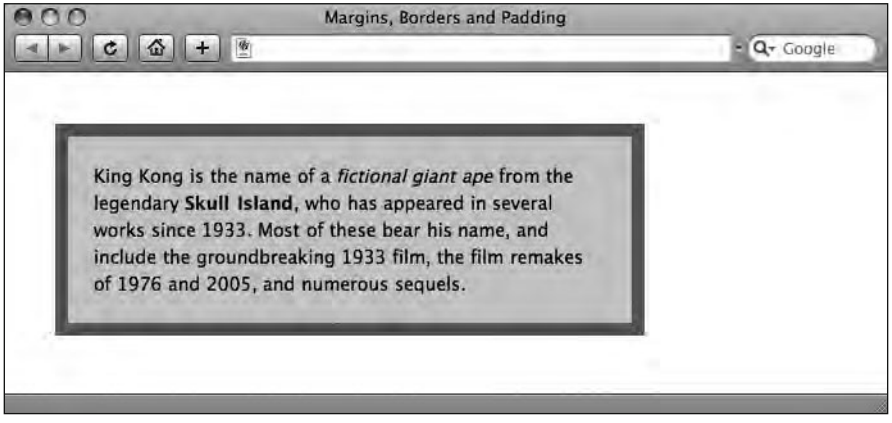

Figure 10-8. Adding some padding to our p element inserts some space between the paragraph and the border.

So now you've seen how an element's margins, borders, and padding relate to each other. Let's take a look at how the preceding additions measure up in the browser.

Understanding how the browser sees this, particularly how it calculates the width the element is now occupying with the added margins, borders, and padding, is important as it will have a bearing on our CSS layouts when we start to place the content of our web pages into divs, assign them a specific width, and set any margin, border and padding declarations.

In Figure 10-9 we've annotated the screenshot in Figure 10-8 to show how the added margin, border, and padding declarations affect the width of our element.

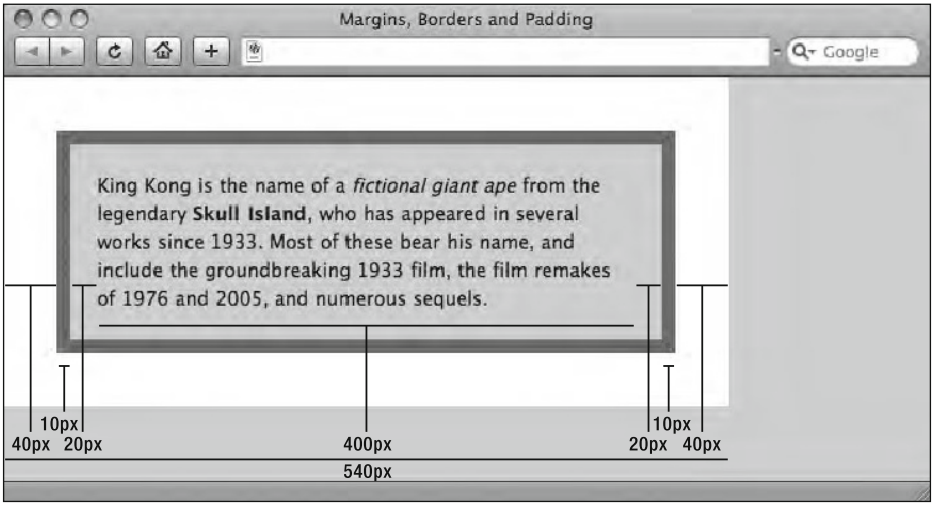

Figure 10-9. The combined width needed for our paragraph is 540 pixels; this consists of all margins, borders, padding, and the width of the paragraph itself.

Looking at Figure 10-9, you can see that when margins, borders, and padding are all specified, the width of our element—or the space that it occupies within the browser window—is as follows:

```
margin-left + border-left + padding-left + element  width  + padding-right +
border-right + margin-right
```

So, in the example in Figure 10-9, our total width is as follows:

```
40px + 10px + 20px + 400px + 20px + 10px + 40px = 540px
```

> *It's worth noting—if only as a historical footnote—that Internet Explorer 5.5 and earlier used a different method of calculating the total width of the box, which used to cause web designers no end of problems. Luckily for you, those days are now, thankfully, a thing of the past. Starting your document with a correct DOCTYPE, as we covered in Chapter 2, instructs Internet Explorer 6 and 7 to use the correct, W3C box model interpretation.*

So, now we know that the browser calculates the width an element occupies on a page by totaling up the width of the element plus any added margins, borders, and padding. It's worth noting that in this example we applied margins, borders, and padding to a `p` element; we could do exactly the same with a `div` filled with content, thereby enabling us to control layout.

Using CSS shorthand for margins, borders, and padding

By now you should be using View Source regularly to look at other designers' source code in addition to looking at the source code of the example pages we've been providing for

each of the chapters. You might have noticed that CSS rules can be written in a variety of ways, in some instances using what's known as **shorthand** to make rules more compact.

You briefly met some CSS shorthand in Chapter 9 when we introduced the `border` property to apply a `border` to the bottom of our links as follows:

```
a:link
{
border-bottom: 1px solid #9CC4E5;
}
```

We can write the same rule in longhand, styling each of the properties separately, as follows:

```
a:link
{
border-bottom-width: 1px;
border-bottom-style: solid;
border-bottom-color:  #9CC4E5;
}
```

Both of these rules style the `border-bottom` (the `border` at the bottom of our `a:link` pseudo-class) in exactly the same way; however, the first is clearly shorter.

Choosing a shorthand method over a longhand method is largely a matter of preference, and it could be argued that a longhand approach allows you to clearly see at a glance exactly what you're styling. However, it's also worth noting that in the preceding example, the shorthand version is half the length of the longhand version. Multiply that over a number of rules, and you're clearly reducing download times and bandwidth requirements.

Let's take a look at another example, from the preceding walkthrough. We added a margin to our paragraph as follows, setting a `margin` on all four sides of the `p` element with a single declaration:

```
p
{
...
margin: 0;
}
```

We could have also written this as four declarations as follows:

```
p
{
...
margin-top: 0;
margin-right: 0;
margin-bottom: 0;
margin-left: 0;
}
```

The two are functionally equivalent and will result in exactly the same display within a browser. When the value of a property is the same on all four sides, it can be specified once, and the browser will apply this value to all four sides.

> *When specifying zero as a measurement, you don't need to specify a unit of measure;* `margin: 0px;` *and* `margin: 0;` *will display identically.*

Let's take a look at a couple of other examples of CSS shorthand in action. In the following example, the values for the top and bottom `margin` are `20px` and the values for left and right `margin` are `10px`:

```
margin: 20px 10px 20px 10px;
```

When the top and bottom, and left and right values are the same, CSS allows us to shorten this even further, as follows:

```
margin: 20px 10px;
```

The first value (`20px`) styles both the top and bottom, and the second value (`10px`) styles both the left and right.

It's important to note that, when using shorthand, margins, borders, and padding are defined in the following order: top, right, bottom, left.

> *An easy way to remember the order in which margins, borders, and padding are applied to all four sides is to think of the numbers on a clock:* `top` *= 12 o'clock,* `right` *= 3 o'clock,* `bottom` *= 6 o'clock, and* `left` *= 9 o'clock.*

A longhand approach can prove useful when you're styling an element with different values on each side as follows:

```
blockquote:
{
margin-top: 20px;
margin-right: 40px;
margin-bottom: 60px
margin-left: 10px;
}
```

In this example, our `blockquote` sits `20px` from the base of any element above it, has `40px` of margin on the right-hand side, inserts `60px` of space beneath it, and has `10px` of margin on the left-hand side.

This could, however, also be shortened and written as follows:

```
blockquote:
{
margin: 20px 40px 60px 10px;
}
```

As with everything we've covered, this can be a lot to take in, but practice makes perfect. Use View Source to view others' CSS, and you'll pick up the preceding shorthand in next to no time.

Styling our <blockquote>

We've looked at specifying `margin`, `border`, and `padding` declarations for a simple `p` element. We've also looked at using CSS shorthand to specify our different rules. Let's combine these two in a real-world example that we'll apply to our King Kong page.

To show margins, borders, and padding in action we'll take our `blockquote` as it stood at the end of Chapter 9 and apply some style to it, giving it margins, a border (on one side only), and some padding.

As it stood at the end of Chapter 9, our `blockquote` was styled using only the browser's default style sheet and the rules we had set on the body to style our typography. The result of these combined rules is shown in Figure 10-10.

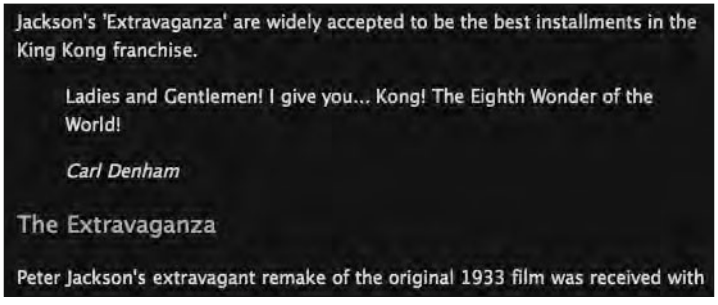

Figure 10-10. Our `blockquote` as it stood at the end of Chapter 9. The default styling isn't doing it any justice.

We've taught you a great deal over the last few chapters, and we can combine this knowledge to apply a little more style to our humble `blockquote`, helping to differentiate it from the surrounding text.

Using the `margin`, `border`, and `padding` declarations we covered earlier and adding a few more declarations we introduced in Chapter 9, we'll differentiate the `blockquote` from the surrounding text, helping to highlight it as a feature within the text. We add the following CSS rule targeting the `blockquote`:

```
blockquote
{
font-family: Georgia, sans-serif;
font-size: 18px;
```

```
font-style: italic;
letter-spacing: 0.1em;
margin-left: 40px;
border-left: 10px solid #E0DFDA;
padding: 0 20px;
}
```

The first four rules should need no introduction, as we covered them in Chapter 9; essentially they set the `blockquote` in a different typeface from that used on the rest of the page and add a little typographic style, helping to visually highlight the quote.

The `margin`, `border`, and `padding` declarations highlight how we can creatively use these properties to add some style to our `blockquote`. It's worth noting that we're not restricted to setting the margins, borders, and padding on every side; we can selectively apply these properties as in this example.

Applying a `margin-left` to the `blockquote` indents it by `40px`, setting it apart from the surrounding text. We also set a `border-left` that gives a strong visual focus to the left-hand side of the `blockquote` and creates a more striking effect, visually distinguishing the `blockquote` from the surrounding paragraphs and headings. Finally, we add some `padding` to the left and right sides to ensure our `blockquote` isn't sitting tight toward the left-hand `border` and to give it an indent on the right-hand side.

The result of our newly added `blockquote` rule is shown in Figure 10-11.

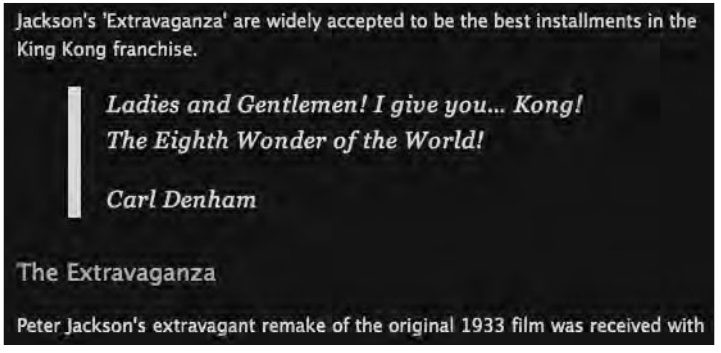

Figure 10-11. Creatively using `margin`, `border`, and `padding` declarations allows us to style our `blockquote` a little more creatively.

By simply varying the values for our `margin`, `border`, and `padding` declarations, we can apply a great deal of style to our `blockquote` element.

Dividing up your document

For the remainder of this chapter we'll focus primarily on XHTML's `div` and `span` elements and their associated `id` and `class` attributes. Used together, these offer us a powerful

means of adding further structure to our XHTML documents by enabling us to divide it up into logical sections.

Up until this point we've focused on the use of well-structured and semantic markup. We've been using the right tag for the job. All good. Now we're going to look at various methods for grouping information together into logical sections.

Take a look at any well-designed web page and you'll notice that information is generally grouped into related clusters. For example, a page may have a header, an area where the web page is branded; a content area, where the bulk of the page's information is gathered; a sidebar, for the site's navigation and any supplementary information; and a footer, for copyright and other related publishing information.

XHTML allows us to use `div` and `span` elements and their associated `id` and `class` attributes to create these document sections. The W3C states the following:

> *The* `div` *and* `span` *elements, in conjunction with the* `id` *and* `class` *attributes, offer a generic mechanism for adding structure to documents. These elements define content to be inline (*`span`*) or block-level (*`div`*) but impose no other presentational idioms on the content. Thus, authors may use these elements in conjunction with style sheets . . . to tailor HTML to their own needs and tastes.*
>
> `www.w3.org/TR/REC-html40/struct/global.html#h-7.5.4`

But what does that mean in English? It means we have two additional elements and their associated attributes to introduce. Once we've introduced them, we can begin to break our King Kong page down into logical sections of related groups of information. This will allow us to further style the document's individual sections by allocating them space and creating a CSS layout.

Identifying your document's sections

Let's take a look at our King Kong page. At this point it's one long page with some well-structured and semantic content and some basic typographic styling. A closer look reveals that this content falls into a number of sections that can be logically grouped.

We have a header area, where we have our Famous Primates brand. We have a content area, where the page's primary information is located. Finally, we have a footer area, where we have some copyright information and the date the document was written. You can see our King Kong page's key sections in Figure 10-12.

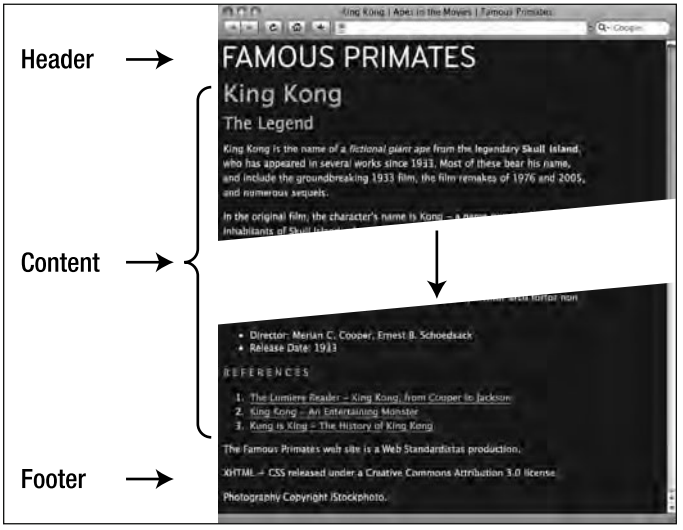

Figure 10-12. Our King Kong web page with some key sections highlighted

As the W3C states, "The `div` and `span` elements, in conjunction with the `id` and `class` attributes, offer a generic mechanism for adding structure to documents"; as such these are perfectly suited for use to divide up our King Kong page into its header, content, and footer sections.

So now that we've identified some of our web page's key divisions and identified the `div` element as a possible mechanism for adding some structure to our web page, how do we target the different sections we've identified specifically? The answer is by using a combination of `div`s and `span`s, and `id`s and `class`es, which we introduce next.

Using div and span elements with id and class attributes

Welcome to the wonderful world of `div`s and `span`s, two useful elements for providing additional markup and meaning within your documents. As you saw earlier, the W3C states that these elements are "a generic mechanism for adding structure to documents. These elements define content to be inline (`span`) or block-level (`div`) but impose no other presentational idioms on the content." This makes them perfect for adding additional structure to our King Kong page.

Where these elements differ from the ones we've introduced so far lies in the fact that they are *generic*. All the elements we've introduced to this point have had an inherent meaning: a `ul` is an unordered list, a `p` is a paragraph, and a `blockquote` is a quotation. `div`s and `span`s are different; they have no inherent meaning and might therefore be described as semantically neutral.

As div *elements are generic elements that impart no deeper semantic meaning to the content nested within them, it's important to ensure that the content you group within your* div *elements is itself marked up using meaningful and semantic elements:* h1–h6, p, ul, li, strong, em, *etc.*

Herein lies their power. As neutral or generic elements, the humble div and span are extremely versatile items in the Web Standardistas' toolkit. We can wrap a div element or a span element around our existing markup and target style at that wrapping element, allowing us to create document sections, for example, the header, content, and footer we identified earlier.

Before we introduce divs *and* spans *properly, a note of caution. It's easy to fall under their spell, wrapping everything in a* div *or a* span *regardless of content.*

The overuse of div *and* span *elements—known as* divitis *and* spanitis, *respectively—is itself a minor form of tag soup to be avoided. In the words of Spiderman's Uncle Ben (or was it Stan Lee?):* "With great power comes great responsibility." *Use these elements sparingly and only where necessary.*

Let's take a look at div and span elements and their associated id and class attributes in a little more detail.

div and span elements

You now know that div elements are block-level and span elements are inline-level. Here we'll give you a look at some examples of both, introducing the markup needed to wrap our content in either of these elements.

Let's take a look at two different uses of the div element: the first used to wrap and identify a single section of a web page, the content section; the second used to wrap and classify one of a number of sections of a web page, a number of blog entries.

To identify a unique part of a web page, we wrap its contents in div tags and use an id attribute as follows:

```
<div id="content">
  <!-- This is where the main content of the web page is situated.
  There can be only one div with an id of content on this page. -->
</div>
```

We're not limited to using div elements for main structural sections of a document, however. We can also use div elements to gather together groups of related information that may occur more than once on a page, blog entries, for example. When styling multiple instances of a div, we use a class attribute instead of an id attribute as in the following example:

```
<div class="blog_entry">
  <!-- This is where one of our blog entries is situated. We can have
 more than one div with a class of blog_entry on this page. -->
</div>

<div class="blog_entry">
  <!-- This is another blog entry. Notice how we've used the class of
 blog_entry more than once on this page. -->
</div>
```

The first example has an id, the second a class. The id is unique; the class can be used more than once. We'll explain the difference between id and class attributes in a little more detail in the next section, but first let's introduce the span element.

Where span elements differ from div elements lies in their inline-level nature. We can use a span *within* a block-level element to differentiate that section from the surrounding section. For instance, in the following example, we can write a CSS rule targeting the span with the class highlighted to style it differently from the surrounding paragraph:

```
<p>I am a paragraph with <span class="highlighted">some inline
text</span> that is highlighted in a different color.</p>
```

We'll show you some examples of spans in action later in this chapter.

id and class attributes

In the previous section we saw two examples of div elements in action, one used an id, one used a class. What's the difference between the id and class attributes in the examples?

The answer is simple: ids are unique, classes aren't. We can have only one element with a specific id attribute on a page, but we can have multiple elements with the same class attribute on a page. This is an important point to grasp and worth spending some time on.

The id attribute is about *identification*. Think of your identity—your id identifies you and you only. Just like there's only one of you, so too can there be only one element with a particular id on a page.

The class attribute is about *classification*. You can have as many elements with a particular class on a page as you like.

An id is unique; there's only one person with a specific id in a group of people. Within that same group, however, are lots of people who belong to that group: id = one; class = many. In short, an id attribute can only be used once per page, whereas a class attribute can be used multiple times.

Looking at the examples in the section titled "div and span elements" earlier in the chapter, you'll notice we used an id of content when creating our content div, but we used a class of blog_entry when creating our blog entry divs. Think about it—on a page there will very likely be only one content area, but there might be several blog entries.

classes and ids are not restricted to divs and spans. In fact, any HTML element including headings, paragraphs, and images can have a class or an id added. We'll see this in action later when we apply a class to our portrait of King Kong, applying a little style to it. For now, let's return to the examples we introduced earlier.

In our example markup earlier, we identified a div and gave it an id as follows:

```
<div id="content">
  <!-- This is where the main content of the web page is situated. -->
</div>
```

We target this div's id with the following CSS:

```
#content
{
/* This is where the rules styling the content div are situated. */
}
```

Note how we use a # (hash) to indicate that the CSS rule is targeting an id. To indicate that a CSS rule is targeting a class, we use a . (period) as shown in the following example:

```
<div id="blog_entry">
  <!-- This is where each blog entry is situated. -->
</div>
```

We target that div's class with the following CSS:

```
.blog_entry
{
/* This is where the rules styling the blog_entry div are situated. */
}
```

As you'll see in the following sections, the div and span elements and their associated id and class attributes form the building blocks on which we build CSS layouts.

It's all in a name

When creating our ids and classes, we're not restricted to set terms like header, content and footer, or blog_entry and diary_entry. In fact, we can choose any names we like. However, when choosing names for ids or classes, it's a good idea to choose names that have semantic meaning, words that can give you pointers when you return to a project after some time has elapsed. Consider the following two class names:

```
<span class="red">Some important content here.</span>
```

and

```
<span class="important">Some important content here.</span>
```

Although the first class name, red, might be presentationally accurate—this year you've decided to highlight important words in red—the second class name, important, is semantically accurate. Ask yourself, what are you *really* highlighting? The fact that the words are in red? Or the fact that they're important?

Next year, after your company's extensive rebranding, important words might be highlighted in blue to reflect your company's new (blue) corporate identity. The following rule won't make quite as much sense:

```
.red
{
color: blue;
}
```

Try to describe the content or function of your elements: primary_navigation is a more meaningful name than right_column, and content is better than box. The point is to use names for ids or classes that are meaningful and semantic; you'll be grateful for this in the long run.

When naming id *or* class *names, we recommend only using the letters a–z in upper or lowercase, the numbers 0–9, and underscores or hyphens. An* id *or a* class *name must* always *start with a letter. It's also worth noting that* id *and* class *names are case sensitive—to a browser,* #Header *and* #header *are different.*

Keeping your id *and* class *names in lowercase and making sure they always start with a letter will help to keep you on the one true path.*

Using div elements to create CSS layouts

Earlier in the chapter we ascertained that our King Kong page could be broken down into three logical sections, a header, content, and footer. By creating divs for the different sections of the document and nesting these within a container div, we can begin to create a layout for our King Kong page using CSS. At this point, to create a single column layout, we break our web page down into the key sections shown in Figure 10-13.

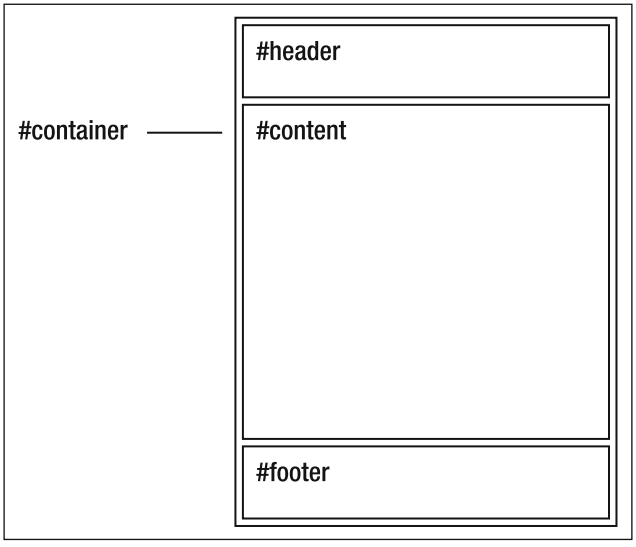

Figure 10-13. The basic div structure that we'll use to create our single column layout

Let's take a look at the markup required to create Figure 10-13. We've created a simplified version of a web page as follows, wrapping the key sections in div elements, adding comments to show where each div closes. (You'll notice the div elements are nested using the First In, Last Out approach we introduced in Chapter 2.) The markup we need is as follows:

```
<body>
  <div id="container">
    <div id="header">
      <!-- This is where the header information is situated. -->
    </div> <!-- Closes the #header div. -->

    <div id="content">
      <!-- This is where the main content of the page is situated. -->
    </div> <!-- Closes the #content div. -->

    <div id="footer">
      <!-- This is where the footer information is situated. -->
    </div> <!-- Closes the #footer div. -->
  </div> <!-- Closes the #container div. -->
</body>
```

Once we've wrapped the key sections of our document in div elements, we can use CSS to apply style to these different div elements just as we would any other element. This includes controlling layout, giving the divs different widths and heights, changing their background color, and positioning them within the browser window.

We'll see div elements in action in the following section when we take the preceding simplified markup and apply basic layout properties to it, demonstrating the effect that this has within the context of a browser window.

A one-column CSS layout

The first stage in our walkthrough is to create a simple web page with `header`, `content`, and `footer` divs nested in a `container` div. Although this page's content is simplified, the essential document sections remain, mirroring our King Kong page.

You'll notice the markup that follows is the same as that in the previous example, but with some of the comments removed for the purpose of simplicity. You can see all of the stages in the walkthrough at the book's companion web site:

```
www.webstandardistas.com/10/walkthrough
```

Use View Source to look at each of the stages. Let's get started. We create a page with the following basic structure:

```
<body>
  <div id="container">
    <div id="header">
      <!-- This is where the header information is situated. -->
    </div>

    <div id="content">
      <!-- This is where the main content of the page is situated. -->
    </div>

    <div id="footer">
      <!-- This is where the footer information is situated. -->
    </div>
  </div>
</body>
```

We add the following CSS to style some basic properties:

```
body
{
font-family: 'Lucida Grande', 'Lucida Sans', Arial, sans-serif;
font-size: 14px;
line-height: 1.6;
background-color: #FFFFFF;
}

h1
{
font-size: 16px;
text-transform: uppercase;
}
```

This markup and CSS renders in the browser as shown in Figure 10-14.

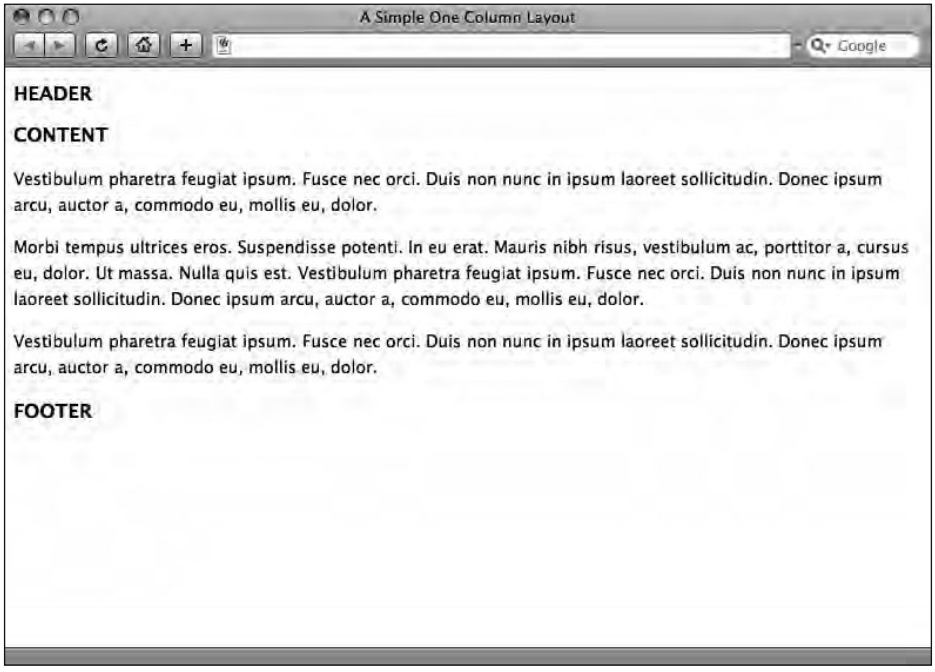

Figure 10-14. Our simplified markup with CSS styling the typography only

Although the content of the web page we're working on for this walkthrough has been simplified, the page is structured identically to our King Kong page with a `header`, `content`, and `footer` section. At the end of this walkthrough, we'll replace the generic content with our King Kong content, resulting in a one-column CSS layout for the page.

We'll now walk through the process of applying some layout to this page using CSS, specifically adding CSS rules to target the four `div` elements we've added to the markup. We add the following four CSS rules:

```
#container
{
width: 550px;
background-color: #FFFFFF;
}

#header
{
padding: 10px 20px;
background-color: #999999;
}

#content
{
padding: 10px 20px;
background-color: #CCCCCC;
}
```

```
#footer
{
padding: 10px 20px;
background-color: #999999;
}
```

These rules set a `width` on our `container div` of `550px`. Looking again at the preceding markup, you can see that the three remaining `div` elements—our `header`, `content`, and `footer`—are nested within the `container`, so setting the width to `550px` on the `container` defines the `width` of all of our `div` elements in this example. The other rules we've added set a `background-color` and add some `padding` on our other `div` elements; this is simply to ensure the effect of adding our rules is easier to see.

The result of adding these rules is shown in Figure 10-15.

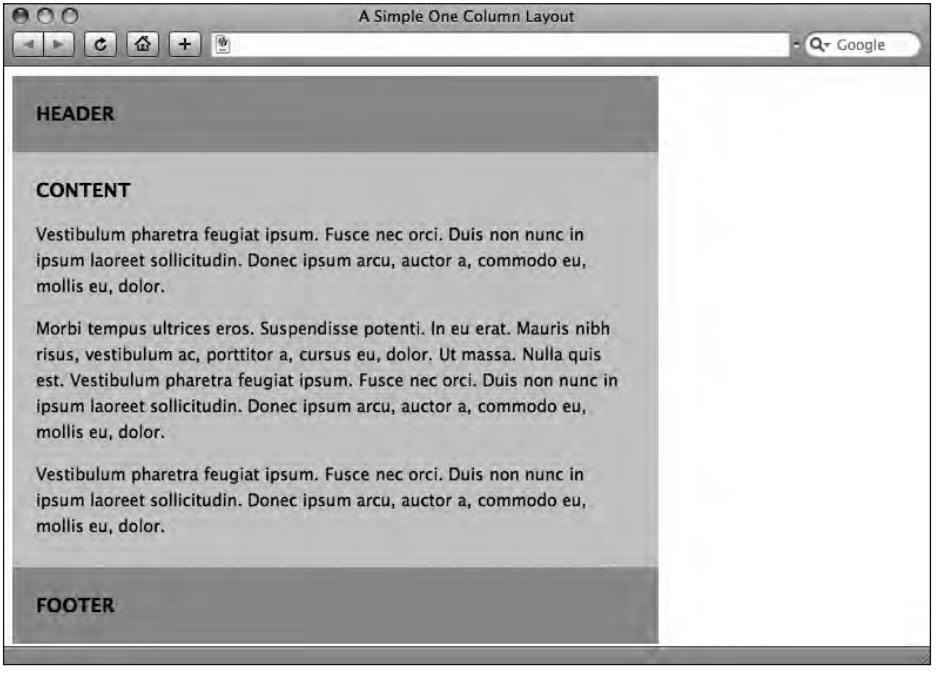

Figure 10-15. Nested within the `container div`, our `header`, `content`, and `footer` now occupy 550 pixels of horizontal space.

The next step in the process is to add a declaration to the `body` rule to remove the `margin` added by browser's default style sheet. We add the following declaration, resetting the `margin` on the `body` to `0`:

```
body
{
font-family: 'Lucida Grande', 'Lucida Sans', Arial, sans-serif;
font-size: 14px;
line-height: 1.6;
```

```
background-color: #FFFFFF;
margin: 0;
}
```

The result of adding this rule is shown in Figure 10-16.

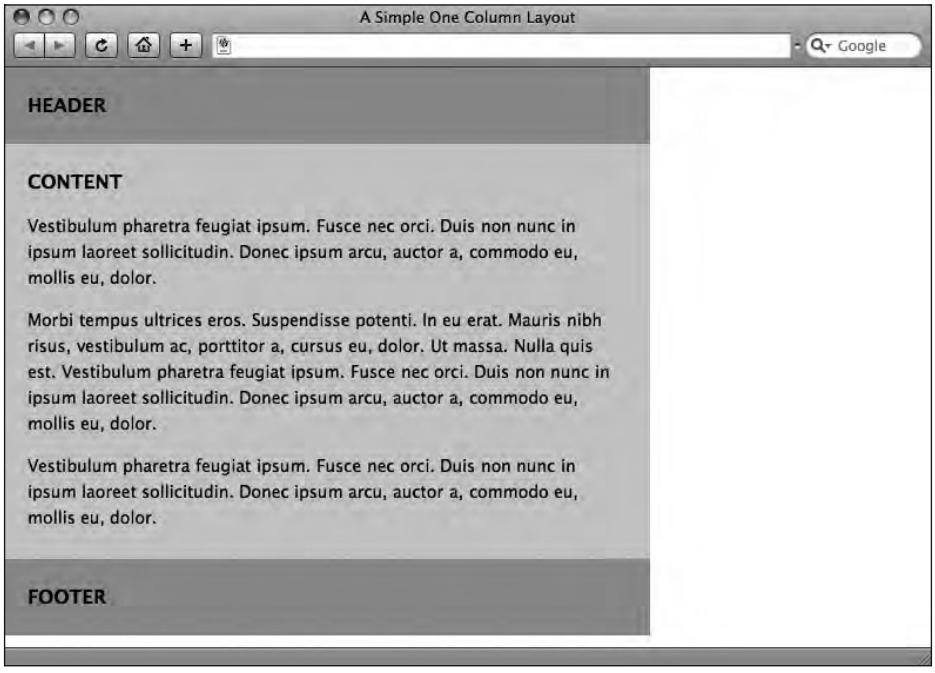

Figure 10-16. Removing the default `margin` added by the browser's default style sheet results in our `container` sitting tight toward the top left corner of the browser window.

Now that we've reset the default `margin`, the next stage is to center the `container div` within the browser window. We do this by adding a declaration to the rule styling the `container`. By adding `margin: 0 auto;` the browser centers the `container div`, giving it a top and bottom `margin` of `0px` and centering the `div` as a result of the `auto` value, which controls the right and left `margin`.

```
#container
{
width: 550px;
background-color: #FFFFFF;
margin: 0 auto;
}
```

The result of this is shown in Figure 10-17.

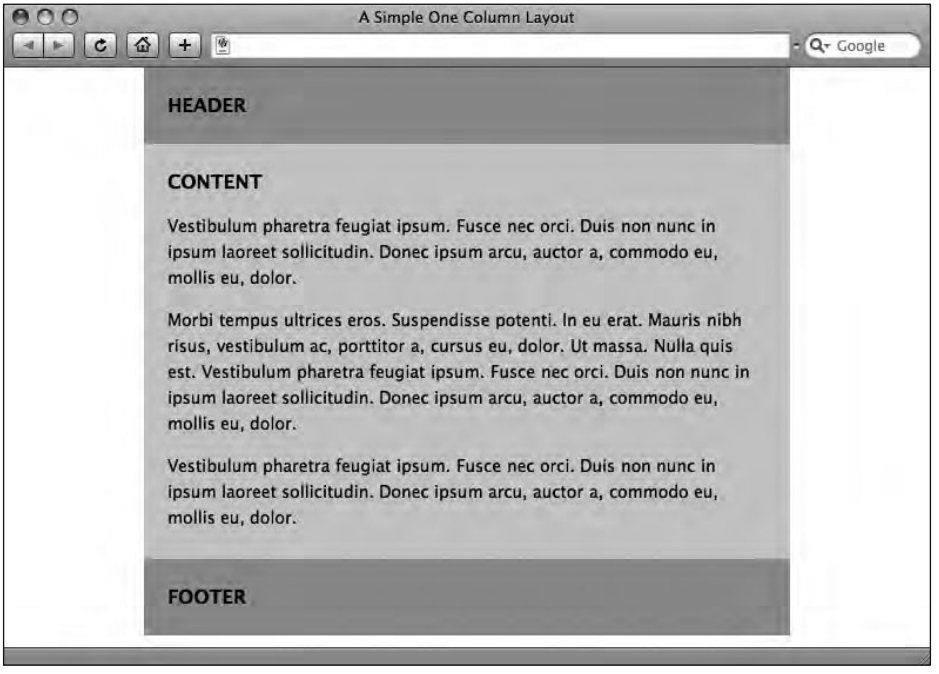

Figure 10-17. Adding a `margin: 0 auto;` declaration centers our `container div` within the browser window.

Our page is beginning to take shape. We'll now center the contents of the `header` and `footer` by using the `text-align` property. We add `text-align` declarations to both the `header` and `footer` rules as follows:

```
#header
{
padding: 10px 20px;
background-color: #999999;
text-align: center;
}

...

#footer
{
padding: 10px 20px;
background-color: #999999;
text-align: center;
}
```

The result of adding these two declarations is shown in Figure 10-18.

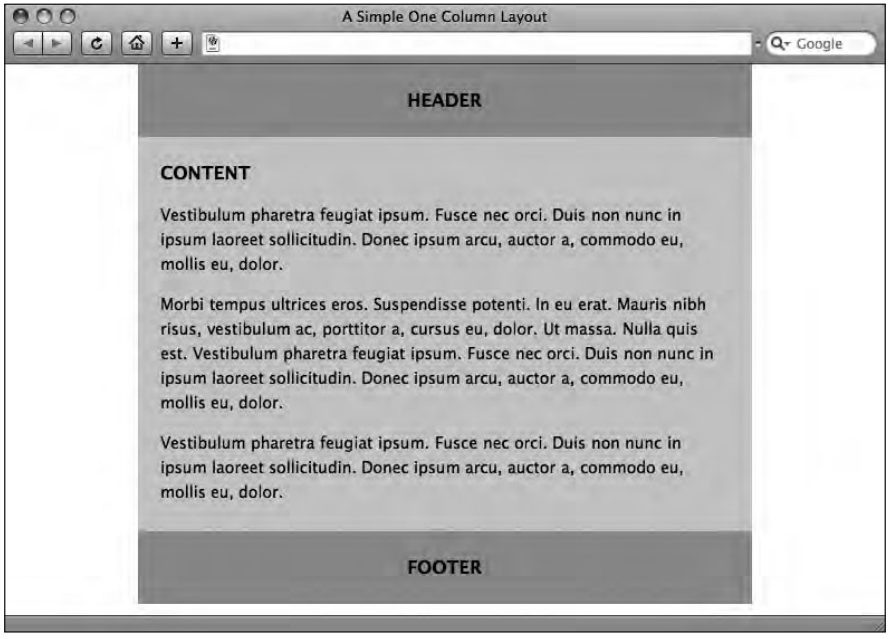

Figure 10-18. Adding a `text-align: center;` declaration to our header and footer centers the content of these `div` elements.

The next stage in the process is to change the `background-color` we set on the `header`, `content`, and `footer`. For the final layout we'd like the `content div` to form a focal point with a darker `background-color` and set the `background-color` of the `header` and `footer` to be the same as the `background-color` of the `body`.

We change the `header`, `content`, and `footer` `background-color` values as follows:


```
#header
{
padding: 10px 20px;
background-color: #FFFFFF;
text-align: center;
}

#content
{
padding: 10px 20px;
background-color: #999999;
}

#footer
{
padding: 10px 20px;
background-color: #FFFFFF;
text-align: center;
}
```

The result of this is shown in Figure 10-19.

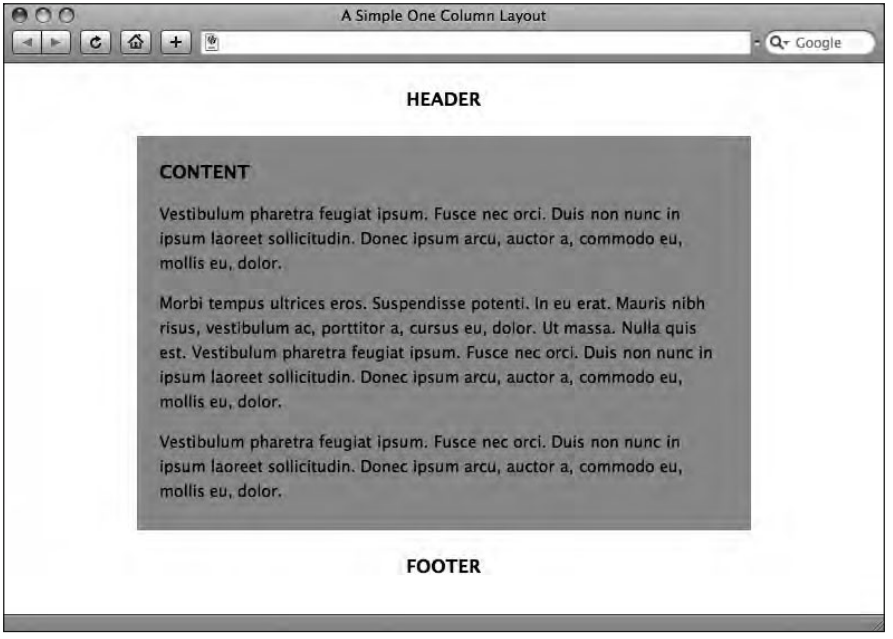

Figure 10-19. Changing the `background-color` of the `header` and `footer` visually distinguishes them from the `content div`.

The final stage in the process is to add a `border` to the top of the `body` and to the base of the `container`. We do this by adding the following two declarations:

```
body
{
font-family: 'Lucida Grande', 'Lucida Sans', Arial, sans-serif;
font-size: 14px;
line-height: 1.6;
background-color: #FFFFFF;
margin: 0;
border-top: 5px solid #000000;
}

...

#container
{
width: 550px;
background-color: #FFFFFF;
margin: 0 auto;
border-bottom: 5px solid #000000;
}
```

The result of adding these two declarations is shown in Figure 10-20.

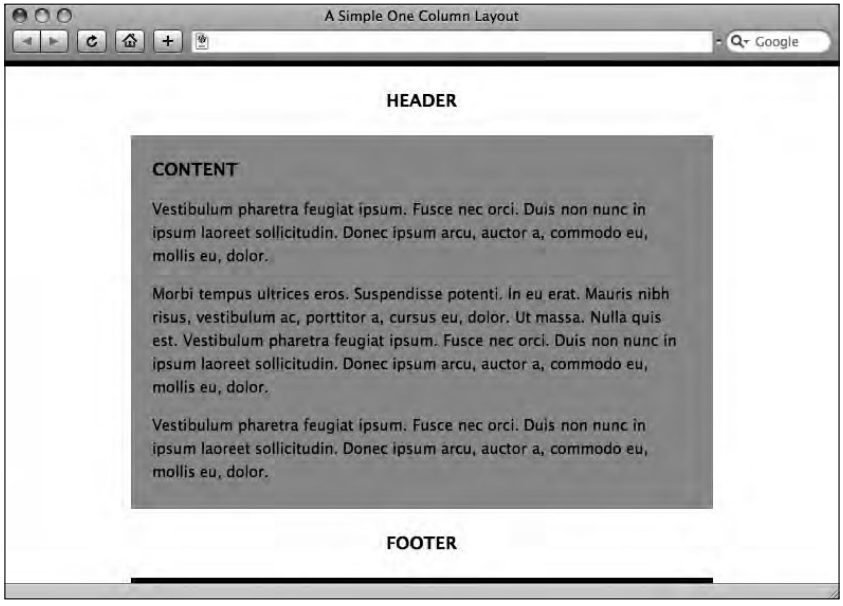

Figure 10-20. Note how the `border` at the top spans the entire width of the body, whereas the `border` at the bottom spans only the width of the `container div`.

Note how the `border` on the body expands to fill the entire width of the browser window, but the `border` on the `container` only occupies 550 pixels (the width of the `container div`).

That's it! It might not seem like much, but once we populate this layout with the content of our King Kong page as it stood at the end of Chapter 9, you can see that things are beginning to take shape. Simply adding the content we styled in the last chapter (and adding a new Famous Primates brand that we'll be supplying for you) results in Figure 10-21.

In just three chapters we've moved from a well-structured web page to a well-styled web page using a single-column layout. Equally importantly, the web page we've been building is accessible and is designed to display across a wide variety of devices, no small achievement.

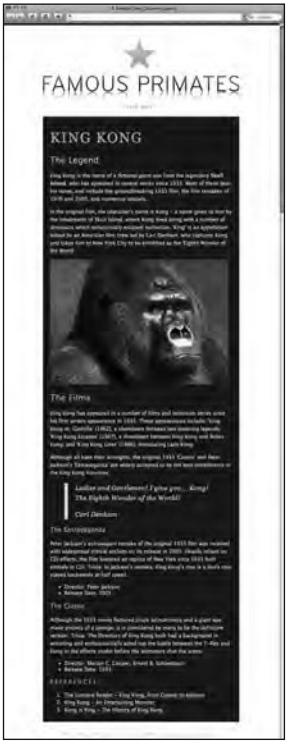

Figure 10-21. Our King Kong page as it now stands using the preceding layout

Using descendant selectors to minimize markup

Astute readers will notice just one thing wrong with the web page as it currently stands in Figure 10-21. The p elements in our footer are inheriting the color we specified on the body element, a shade of creamy white (#E0DFDA). This is fine for the text in the main content section, which has a dark brown background-color (#25201C); however, against the light background of the footer, the text lacks contrast as shown in Figure 10-22.

Figure 10-22. The text in our footer suffers from a lack of contrast, rendering it very difficult to read.

In order to fix this we need to write a rule that targets *only p elements in the footer*. The good news is we can use this fix to introduce another extremely useful aspect of CSS, namely **descendant selectors** (sometimes referred to as **contextual selectors**).

Before we introduce you to descendant selectors, let's look at one way we could resolve the problem of our p elements that we *don't* recommend. By showing you two ways of achieving the same goal—one that relies on bad practice and one that relies on good practice—we can highlight the better approach.

Earlier in the chapter we mentioned the twin evils of *divitis* and *spanitis*, or the overreliance on div and span elements. Another problem that beginners often fall into is an overreliance on class attributes, using class attributes left, right, and center, resulting in bloated and overcomplicated markup.

We could solve the problem of the footer by styling the p elements that are situated in the footer, giving each of them a specific class as follows (we've removed the markup for the links for the purpose of simplicity):

```
<p class="footer_text">The Famous Primates web site is a➥
Web Standardistas production.</p>
<p class="footer_text">XHTML + CSS released under a Creative Commons➥
Attribution 3.0 license.</p>
<p class="footer_text">Photography &copy; iStockphoto</p>
```

By adding class="footer_text" to each of our p elements, we can then write a CSS rule, as follows, that takes care of all instances of p with a class of footer_text:

```
.footer_text
{
font-size: 11px;
```

```
color: #383330;
}
```

While this will certainly work and will take care of styling the p elements in our footer, differentiating them from the rest of the p elements on the page, it's not the most efficient solution to the problem. As you can see, it relies on us adding class="footer_text" to each of our p elements. Not exactly the most efficient approach to writing lean and mean markup, certainly not the Web Standardistas' way.

Good news, CSS offers us a far better solution, which allows us to selectively target just these p elements based on their context in the footer. We can write a descendant selector that targets all p elements in the footer as follows:

```
#footer p
{
font-size: 11px;
color: #383330;
}
```

Essentially what this does is inform the browser to "look for any instance of the element p within the div footer and apply this rule." This allows us to target *just* these p elements with laserlike precision. This does away with the need for the additional markup—class="footer_text"—on each of our p elements in our footer. The result: less markup = less maintenance = faster downloads. The result of our new rule is shown in Figure 10-23.

Figure 10-23. The p elements in our footer are now styled using a descendant selector, differentiating them from all other instances of p on the page.

Now the paragraphs in our footer are much more legible, and the type is smaller than the main p elements in our content section (which is appropriate given that this is the "small print"). However, the links in light blue need a little more contrast. We'll use a descendant selector to style these differently from the other links on the page. We add the following rule:

```
#footer a:link, #footer a:visited
{
color: #383330;
border-bottom: solid 1px #383330;
}
```

By using *descendant selectors* in conjunction with *grouped selectors* (introduced in Chapter 9), we can really minimize our markup, styling multiple elements with a single rule and improving the efficiency of our style sheets considerably.

Essentially the preceding rule targets all instances of `a:link` and `a:visited` (i.e., links and visited links) situated in the `footer` and sets their `color` to the same as the `p` elements in the `footer`, differentiating them from the `p` elements through the inclusion of a `border-bottom`. The result of this is shown in Figure 10-24.

Figure 10-24. The `a:link` and `a:visited` pseudo-classes are now styled differently, set in the same `color` as the `p` elements in the `footer`.

An understanding of how descendant selectors work can significantly help to reduce markup. Applying style to different elements based on their context reduces the number of CSS rules required and saves peppering your markup with redundant `classes`. Both things the aspiring Web Standardista should be striving for.

Styling details with the span element

Where `div` elements are *block-level*, `span` elements are *inline-level*. Imagine you have a paragraph that contains some content you'd like to style differently from the rest of the paragraph—enter the `span`. Wrapping a `span` element around the inline content you'd like to style differently allows you to target your CSS at the `span` to achieve the effect you'd like.

In the next two sections we'll take a look at this in action. In the first example, we'll use a `span` to style a specific part of our `footer`. In the second example, we'll show how noted Standardista Dan Cederholm uses a `span` to style part of his SimpleBits web site's strapline without needing to resort to image replacement techniques to create an elegant ampersand.

Using a span to style inline content

Let's take a look at a `span` in action. We'll work with the first line of our `footer`, which contains our publishing and copyright information. At present it's rendered as in Figure 10-24; however, we'd like to differentiate the words *Famous Primates* in the first line, using the `font-weight` property to highlight the words in `bold` and transforming their case to `uppercase` using the `text-transform` property. We add the following to our markup (again, we've removed the markup for the link for the purpose of simplicity):

```
<p>The <span class="primate">Famous Primates</span> web site is a
Web Standardistas production.</p>
```

By wrapping the words *Famous Primates* in a span and giving it a `class` of `primate`, we can write a rule targeting all instances of the `class` `primate`. We add the following CSS rule to our style sheet:

```
.primate
{
text-transform: uppercase;
font-weight: bold;
}
```

Figure 10-25 shows the result of wrapping the span around the words *Famous Primates* and targeting a CSS rule at the span—exactly what we wanted, differentiating the words from the rest of the paragraph.

Figure 10-25. The words *Famous Primates* are now styled in bold and uppercase, thanks to the added span.

Where possible it's best to eschew the use of the span element in favor of more meaningful markup; however, in some instances a span is the best option available.

Astute readers will notice one other change in the `footer`. At the end of Chapter 9, our final line read "Photography Copyright iStockphoto"; in the preceding example, we've now included a copyright (©) symbol. We do this by replacing the word *Copyright* with the character entity for the copyright symbol:

```
&copy;
```

The result is a copyright symbol, more clearly highlighting the fact that the copyright of the photography we're using belongs to iStockphoto.

In the next section, we take a look at another example of a span in action at noted Standardista Dan Cederholm's SimpleBits web site.

Dan Cederholm's illustrious ampersand

One elegant example of a span in action is noted Standardista Dan Cederholm's illustrious ampersand, as featured in the strapline of his SimpleBits web site (`www.simplebits.com`). Wrapping the ampersand in a span allows it to be differentiated from the surrounding text to create a sophisticated, graceful, and eye-catching feature as shown in Figure 10-26.

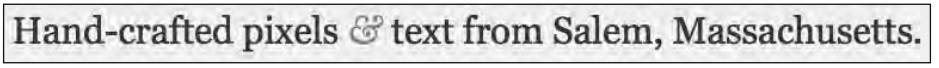

Figure 10-26. Dan Cederholm's illustrious ampersand, achieved by wrapping the ampersand in a span element

By using a semantically neutral span element, the heading still displays perfectly in a non-CSS environment, as shown, unstyled, in Figure 10-27.

Figure 10-27. Unstyled, the illustrious ampersand is perfectly displayed in a non-CSS environment.

So you've now met divs and spans, two generic methods of marking up and adding structure to our documents. Of the two, the div is the one you'll be using the most as we move forward. Gathering information together in a div allows us to give that division a size, add some style to it, and apply layout to it by giving it a width and a height.

The ability to create accurately sized divisions within our web pages allows us to embark on layout using CSS. This is another area where CSS can prove powerful.

We're not restricted to using id and class attributes with div and span elements, however. We can apply both to other elements, thereby helping us to avoid the overuse of div and span elements, as you'll see in the following section when we style our King Kong image differently through the use of an added class attribute.

Styling with class attributes

At this point our King Kong web page features two images, the Famous Primates brand at the top of the page and the portrait of the mighty King Kong. In this section we'll look at using a class attribute to give the King Kong portrait its own look and feel.

We'd like our portrait of King Kong to render in the browser with some additional style, using margins, borders, and padding. To do this we need to differentiate our King Kong image from the Famous Primates brand at the top of the page so that we can target our CSS at *just* the King Kong image.

We add a class attribute to the King Kong image as follows:

```
<img src="images/king_kong.jpg" width="500" height="350" alt="The
mighty King Kong, a fearsome giant ape." title="King Kong
contemplates scaling yet another tall building." class="portrait" />
```

By adding a class to the portrait of King Kong, we can now write a rule targeting *any* image with a class of portrait. We add the following rule to our style sheet:

```
.portrait
{
margin: 10px 0;
```

```
border: 1px solid #FFF7D7;
padding: 4px;
}
```

You should by now be able to tell that this rule is adding margins (`10px` top and bottom; `0px` right and left), a border (`1px solid #FFF7D7;`) and padding (`4px`) to our King Kong portrait. The result of this is shown in Figure 10-28.

Figure 10-28. Our King Kong image with a 1 pixel border and 4 pixels of padding around it

As you know by now, we're not restricted to styling the image in this way; we could have a thicker border, less padding—we could have anything we want—by changing our CSS rule.

Enhancing your design by adding background images with CSS

We can use CSS to apply a `background-image` to almost any element in XHTML including our page's `body` element (this can be useful for creating page backgrounds to give character to a web page), our `div` elements, blockquotes, paragraphs, lists, and even inline-level elements.

Creative use of background images can help to break up a page and, as you'll see in this section, help to create more dynamic layouts. In the following section we'll demonstrate how to add a `background-image` to our `body` element, in the process introducing you to how to apply background images using CSS. We'll then apply a `background-image` to our `blockquote` to further enhance its presentation.

Adding a background image to the body

We'll start with a simplified web page with a single paragraph with 40px of `margin` applied. We add the following rules to our style sheet to establish some basic style:

```
body
{
font-family: 'Lucida Grande', 'Lucida Sans', Arial, sans-serif;
font-size: 14px;
color: #000000;
background-color: #FFFFFF;
margin: 0;
}

p
{
width: 400px;
line-height: 1.5;
margin: 40px;
}
```

The result of this can be seen in Figure 10-29.

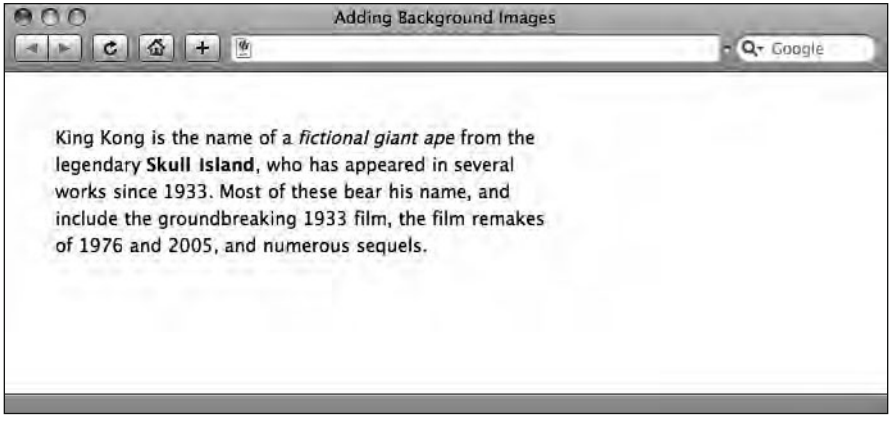

Figure 10-29. A simple paragraph with minimal style added

We create a simple 8 × 8 pixel tile in our image editor as shown in Figure 10-30. This is designed to tile seamlessly, repeating horizontally and vertically across the background of our web page like wallpaper.

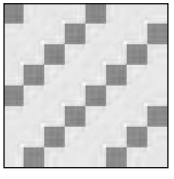

Figure 10-30.
We create a simple image in our image editor (magnified here) that we can tile as the background of our web page by setting the image as a `background-image` on our `body` element.

We add background-image and background-repeat declarations to our body rule as follows:

```
body
{
font-family: 'Lucida Grande', 'Lucida Sans', Arial, sans-serif;
font-size: 14px;
color: #000000;
background-color: #FFFFFF;
margin: 0;
background-image: url(diagonal_tile.png);
background-repeat: repeat;
}
```

The background-image declaration provides a relative link to the image file we're using, in this case with the relative location of our diagonal_tile.png image file. It's worth noting that the URL specifies the location of the image file in relation to the style sheet. In this case we're using an internal style sheet, so the preceding rule assumes the diagonal_tile.png file is in the same folder as the web page.

The background-repeat declaration instructs the browser to repeat the image along both the x and y axes. (Using repeat-x would repeat the image only along the x axis; using repeat-y would repeat the image only along the y axis; using no-repeat would display the background-image once only, not repeating it.)

The result of adding the preceding declarations is shown in Figure 10-31.

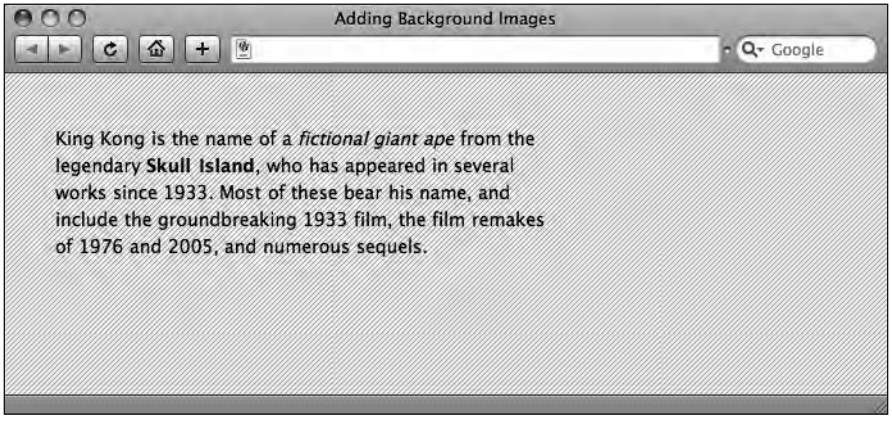

Figure 10-31. A simple background-image tiled along both the x and y axes

Although we've only used a simple, repeating tile in this example, tiling across both the x and y axes, it's possible to creatively use the background-image property to help break up the underlying grid of a simple web page and create layout effects.

One point worth noting is the importance of specifying a background-color *in addition to* a background-image. Although the background-color sits *behind* the background image,

its good practice to specify a background-color (and a contrasting text color) when using background images, in case a user is browsing your web site with images switched off.

> *It's important to note that using background images in CSS is not the same as adding images to a page using the* img *element. Images included with the* img *element are seen as* content; *images included using the* background-image *properties of CSS are seen as* presentation.

Background images can add to a design considerably. One excellent resource for well-designed background images that are free to use and inspirational is Kaliber10000's Pixel Patterns Collection (www.k10k.net/pixelpatterns/).

Using background images with other elements

In the margins, borders, and padding walkthrough earlier in this chapter, we added a background-color to our p element to highlight the space the paragraph occupied as a block-level element. Although we used this to demonstrate the effects of adding our margin, border, and padding declarations, we could have also employed the background-color property as a design element, perhaps using it to distinguish different parts of our document by setting the background-color to a different color.

Let's take a look at our blockquote again, now setting a complementary background-color to further highlight the quotation. We add the following background-color declaration to our CSS rule:

```
blockquote
{
font-family: Georgia, sans-serif;
font-size: 18px;
font-style: italic;
letter-spacing: 0.1em;
margin-left: 40px;
padding: 0 20px;
border-left: 10px solid #E0DFDA;
background-color: #3E322B;
}
```

The result of adding this rule is shown in Figure 10-32.

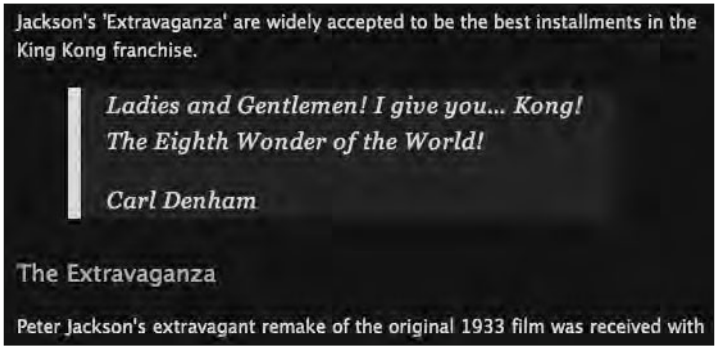

Figure 10-32. Adding a `background-color` declaration to our `blockquote` element helps to further differentiate it from the surrounding text.

With CSS we can control much more than background colors for our elements, however. As we mentioned in the last section, we can also add background images to our different elements using CSS to create a variety of presentational outcomes.

Let's take a look at this in action as applied to our `blockquote` again. We've created an image with a gradient, using the shades of brown employed on the King Kong page; we'll use this to create a `background-image` for our `blockquote`.

We add a `background-image` declaration to our CSS rule as follows:

```
blockquote
{
font-family: georgia, sans-serif;
font-size: 18px;
font-style: italic;
letter-spacing: 0.1em;
margin-left: 40px;
padding: 0 20px;
border-left: 10px solid #E0DFDA;
background-color: #3E322B;
background-image: url(blockquote_bg.png);
}
```

The result of this is shown in Figure 10-33.

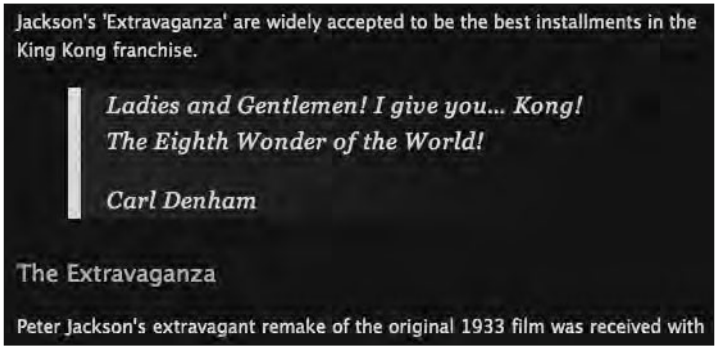

Figure 10-33. Adding a `background-image` declaration to our `blockquote` element helps to break up the grid on the page, while still differentiating the `blockquote` from the surrounding text.

Both of the preceding examples of the `background-image` property barely scratch the surface. Used creatively, background images can help to break up the grid in a typical CSS layout, resulting in more interesting and innovative layouts.

We've added two background images to our King Kong web page—one on the `body`, one on our `blockquote`—to give a glimpse of what's possible using background images. You can see these at the book's companion web site:

```
www.webstandardistas.com/10/king_kong.html
```

To give you some experience of the `background-image` property, you'll be adding background images to your Gordo web page for this chapter's homework.

Summary

So what have we covered? It's been a busy and important chapter, certainly one that's well worth reading again. We've covered a lot of fundamentals including dividing up complex web pages into logical divisions using `div` elements, an understanding of which forms the basis of creating CSS layouts.

We introduced the CSS box model and looked at adding margins, borders, and padding to our elements with a specific focus on how these properties could be used creatively when applied to our page's different elements. Lastly, we looked at using background images in CSS to help break up our web pages a little.

In the next chapter we'll build on the fundamentals introduced in this chapter, showing you how to create two-column layouts to develop your CSS layout skills considerably.

Homework: Creating a one-column CSS layout

In this chapter we added some additional structure to our document by breaking our King Kong page down into key sections or divisions. The primary focus of the chapter was the creation of a one-column CSS layout to give our King Kong page a little more style and presence. By following along with the examples covered throughout the chapter, you should be capable of creating a single-column layout for your Gordo page in addition to developing the design of its different elements.

We introduced the cascade in Cascading Style Sheets and looked at adding margins, borders, and padding to your elements, introducing you to the concept of the box model. As a by-product of this we introduced CSS shorthand, which will enable you to write shorter and more compact style sheets.

Using our King Kong page as an example, we applied our knowledge of margins, borders, and padding to improve our `blockquote`, adding a `background-image` to it also, to further differentiate it from the rest of the page.

The major focus of the chapter was on the introduction of `div` elements, which we used to break our page down into different sections, which we then positioned and controlled using CSS. Along the way, we looked at how we could use `span` elements to style details, enabling you to zero in on specific inline-level sections to apply style to. We also introduced the concept of descendant selectors, a powerful means of minimizing markup.

Finally we looked at enhancing your design by adding background images with CSS, both to the `body` and to other elements.

Your homework for this chapter will be to apply what you've learned to your Gordo page, improving its layout considerably.

1. Add the div tags

If you've been following along with the homework, your Gordo page should feature a similar structure to our King Kong page. For the first stage of this chapter's homework, we'd like you to identify the key sections of your Gordo web page and add `div` tags where appropriate.

Once you've wrapped your `header`, `content` and `footer` sections in `div`s, we'd like you to wrap everything in a `container div`, which you'll use to center your design.

2. Write the CSS

Referring to the examples in this chapter, create rules for your `header`, `content`, `footer`, and `container div` elements to control the layout of the page.

Once you've taken care of the layout, we'd like you to address the issue of the `footer`. You'll need to create a rule that styles the text in the `footer`, overriding the `font-size` and `color` declaration inherited from the `body` rule. Write this rule using a descendant selector to minimize your markup.

3. Upgrade the Famous Primates brand

As with the previous chapters, we've created everything you need to complete the homework. This includes an all-new-and-improved Famous Primates brand, in addition to two pre-baked gradient images to use as a background-image for your Gordo page and your blockquote. You can download the assets here:

```
www.webstandardistas.com/10/assets.zip
```

Once you've downloaded these files, transfer the images to your images folder and upgrade the Famous Primates brand.

4. Style the blockquote

Styling the blockquote is a two-stage process. Firstly we'd like you to apply margins, borders, and padding to distinguish the blockquote from the surrounding text. Once you've achieved that, we'd like you to specify the image we supplied as a background-image for your blockquote.

5. Style Gordo's image

Style your image of Gordo by creating a class to differentiate it from the Famous Primates brand and create margin, border, and padding declarations for the image. We added a 1 pixel border with 4 pixels of padding; you might like to experiment with an alternative, for example, try adding a 5-pixel border with no padding in the light shade of blue we've been using (#9CC4E5) as follows:

```
.portrait
{
margin: 10px 0;
border: 5px solid #9CC4E5;
}
```

Remember, you'll need to add a class="portrait" attribute to the markup of your Gordo image for this rule to take effect.

6. Add a background-image

The last part of the homework is to add the background-image we've supplied. This image needs to be repeated along the x axis, which will tile it horizontally. You'll also notice from our King Kong example page that we've included an additional declaration background-attachment: fixed;. The background-attachment property sets whether a background-image is fixed or scrolls with the rest of the page. Try commenting out this declaration and testing its effect on the background-image within a browser.

As usual, to help you with the different stages of this chapter's homework, we've created our own, similarly styled, page about King Kong featuring all of the enhancements we covered in this chapter. You can refer to this, using your browser's View Source menu command to see how we've updated our CSS, here:

```
www.webstandardistas.com/10/king_kong.html
```

Once you've completed the rollout of your one-column CSS layout for Gordo, put the kettle on and enjoy a cup of *Robert Fortune Blend 41* as you prepare yourself for the next chapter.