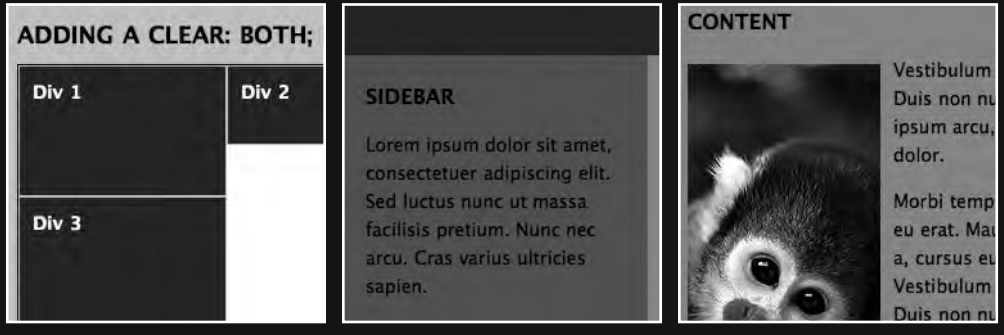
CHAPTER 11
A TWO-COLUMN CSS LAYOUT



In this chapter we create a slightly more complex, two-column layout. Building upon the one-column web page we created at the end of Chapter 10, we'll add a second `div` to sit alongside the `content div`. We'll use this to contain some additional sidebar content, which could be anything: some supporting content, a navigation list, or any number of other elements.

For the purposes of this chapter, and to keep the focus on layout, we'll keep the content of the `sidebar div` simple. Inside it we'll place a single `<h2>` followed by a short paragraph containing links to our Cheeta and Cornelius pages. In the next chapter we'll replace this `sidebar` content with a styled list of links, creating the navigation for our Famous Primates web site.

As with the other chapters, we'll be working with our King Kong web page, and you'll be working along with your Gordo page for the homework. By the end of the chapter you should have a working knowledge of two-column layouts, floating elements, and the importance of clearing floats.

The fundamental concepts discussed in this chapter will provide a solid foundation on which you can build, providing the building blocks for more complex layouts down the line.

A float-based CSS layout

There are a number of ways to create multicolumn layouts using CSS. These include floating, absolute positioning, and using negative margins. We'll be focusing on a float-based approach as we find this the most flexible method, as well as being easy for beginners to understand.

> *Although we've focused on float-based layouts in the book, we have featured a roundup of other layout methods, including absolute positioning and the use of negative margins, in the periodical section of the book's companion web site where we have also provided links to a number of additional resources that deal with CSS-based layouts.*

To get started with float-based layouts, we need to introduce the idea of floating elements and removing them from the document flow. But what exactly do we mean by the *document flow*?

Looking back at the XHTML web pages we created earlier in the book, you know that web pages are linear by nature. The elements of your web pages flow down the page starting with the first element at the top of the page and finishing with the last element at the bottom of the page. Elements can be either block-level (forcing a line break) or inline-level.

In Western languages a paragraph of text starts at the top left corner of the page and works its way down toward the bottom right corner of the page. When writing, each word is placed just to the right of the preceding word until there is no more room on the line; the next word is then placed at the very left of the next line.

Similarly, all elements on an XHTML page have a natural tendency to sit as close to the top left of the page as possible. This is the natural flow of web pages (in the Western speaking world) and is known as the **document flow**.

Using the `float` property in CSS allows us to remove elements from this flow, enabling us to position them on the page. A floated element is taken out of the normal top left to bottom right flow and moved as far as possible to the left or right (depending upon the CSS rules written to target it). This will become clearer to grasp when we introduce some examples in this chapter.

Before we introduce you to some examples of floats in action, by floating some `div`s, let's see what the W3C has to say about floats:

> *A float is a box that is shifted to the left or right on the current line. The most interesting characteristic of a float (or "floated" or "floating" box) is that content may flow along its side (or be prohibited from doing so by the "clear" property). Content flows down the right side of a left-floated box and down the left side of a right-floated box.*
>
> *A floated box is shifted to the left or right until its outer edge touches the containing block edge or the outer edge of another float. If there's a line box, the top of the floated box is aligned with the top of the current line box.*
>
> `www.w3.org/TR/CSS21/visuren.html#floats`

While that might sound a little confusing just now, it should all become apparent as we walk through some practical examples demonstrating how floats work, which we'll do in the following section. Using some simplified web pages we apply floats to our different elements and show you how the floats we add affect the normal document flow.

Floating divs

We know from Chapter 10 that `div`s are block-level elements; this means they force a line break after they close. We can, however, use the `float` property to alter their relationship to the natural document flow of a web page.

To demonstrate how floats work, we've created a basic web page with a `container div` and three nested `div` elements. We've numbered the `div`s so that you can see exactly how the different `div` elements are affected as we apply floats to them. Our markup looks like this:

```
<h1>No Floats</h1>
<div id="container">
  <div class="box"><h2>Div 1</h2></div>
  <div class="box"><h2>Div 2</h2></div>
  <div class="box"><h2>Div 3</h2></div>
</div>
```

We give the container a width and height of 400px and give the three divs nested inside it a width and height of 100px; lastly we've added 1-pixel borders and background colors on all of the div elements and the body element to make our divs easier to see. We do this using the following CSS:

```
body
{
margin: 20px;
padding: 0;
font-family: 'Lucida Grande', 'Lucida Sans', Arial, sans-serif;
background-color: #CCCCCC;
}

#container
{
width: 400px;
height: 400px;
border: 1px solid #000000;
background-color: #FFFFFF;
}

.box
{
width: 100px;
height: 100px;
border: 1px solid #CCCCCC;
background-color: #333333;
}
```

The preceding markup renders in the browser as shown in Figure 11-1.

As with all div elements, our three smaller divs are block-level; as such they force a line break so that our three divs sit one on top of the other. At this point, the three nested divs are in the normal document flow with the first box at the top and the last box at the bottom.

Let's now apply float: left to the div elements with the class of box and see how this affects the layout. We add the following additional declaration to our CSS:

```
.box
{
float: left;
width: 100px;
height: 100px;
border: 1px solid #CCCCCC;
background-color: #333333;
}
```

The result of adding this declaration is shown in Figure 11-2.

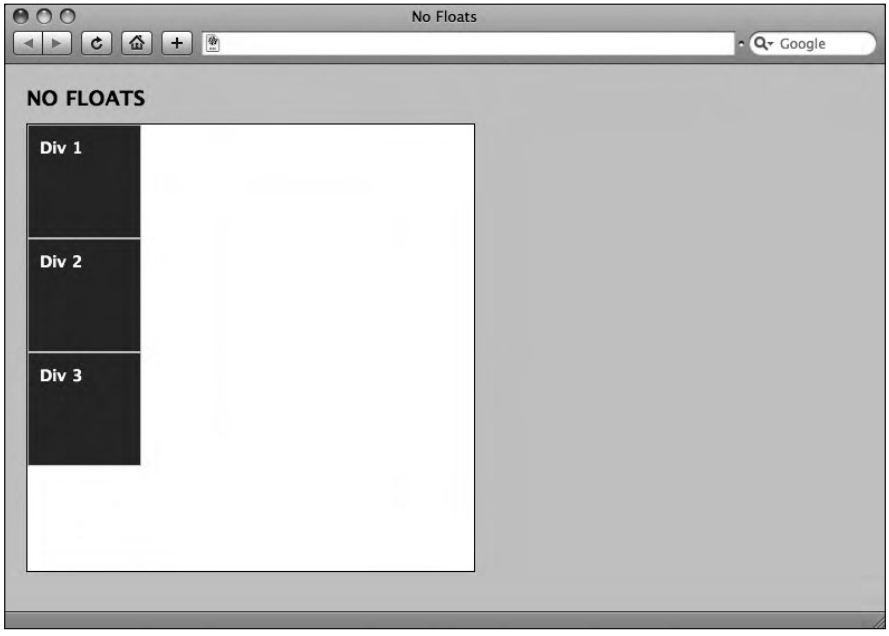


Figure 11-1. Our three nested `div` elements, each with a class of `box` as they render nested within the `container`. No floats are added at this point.

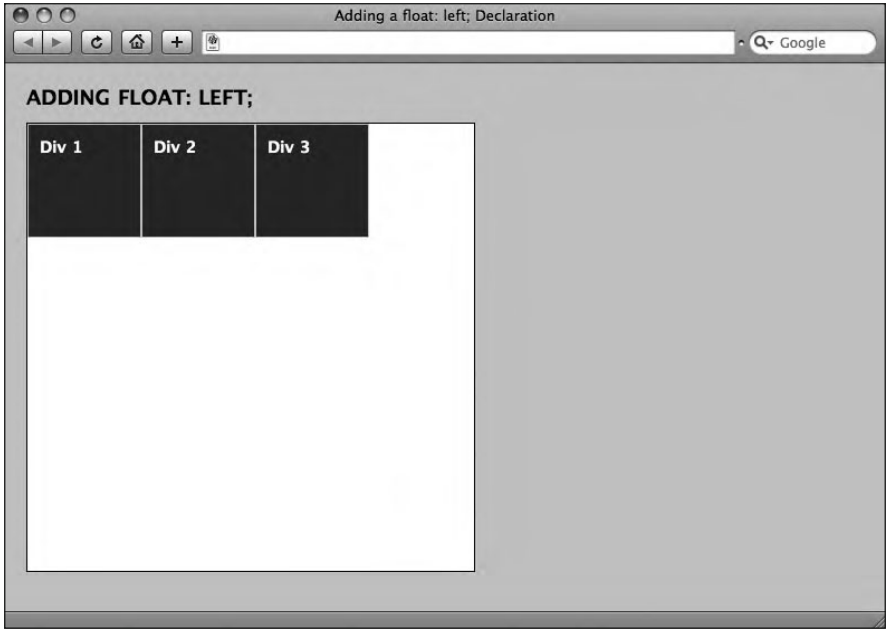


Figure 11-2. By adding a `float: left;` declaration, our three nested `div` elements, each with a class of `box`, are now removed from the normal document flow.

As you can see, the addition of a `float: left;` declaration removes all `div`s with a class of `box` from the normal document flow, in turn moving them as far left as possible within their containing element (in this case the `container div`). The result is that the three nested `div` elements display horizontally and are aligned—or floated—to the left.

Let's now apply `float: right` to the same `div` elements and see how this affects the layout. We amend our CSS as follows:

```
.box
{
float: right;
width: 100px;
height: 100px;
border: 1px solid #CCCCCC;
background-color: #333333;
}
```

The result is shown in Figure 11-3.

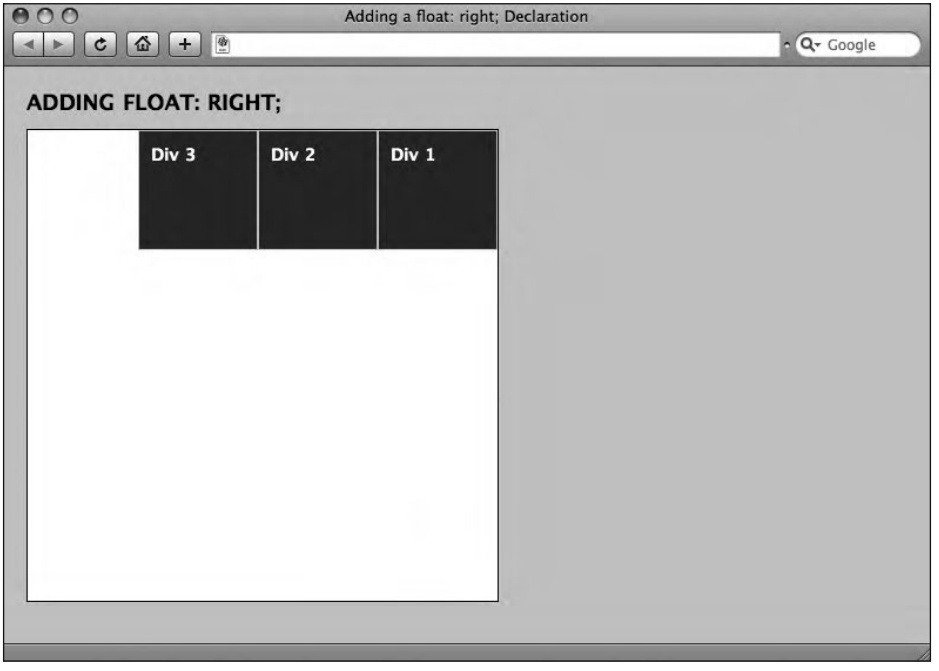


Figure 11-3. Changing our `float` value from `left;` to `right;` alters the relationship of our nested `div`s to the `container div`.

Looking at Figure 11-3, you might be forgiven for wondering why the order of the `div`s is now Div 3, Div 2, Div 1.

At first glance this might appear confusing; however, the explanation is simple. The browser is rendering Div 1 as far to the right as it will go, in this case aligning it to the top right edge of the `container` `div` it's nested in. It's then rendering Div 2 in the markup, and then Div 3. This is due to the `float: right;` declaration, which takes the `div`s with the class of `box` out of the normal document flow and floats them to the right.

This might be confusing to start with, but it's important to get an understanding of how floats remove elements from the normal flow of the document, as this will form the basis of the layouts we'll be creating later in this chapter.

Before we get on to applying this knowledge to our two-column layout, let's take a look at what happens when not all of our `div` elements are floated. To do this, we've adjusted and added to our markup as follows, introducing a different class for the first of our three smaller boxes:

```
<div id="container">
  <div class="boxspecial"><h2>Div 1</h2></div>
  <div class="box"><h2>Div 2</h2></div>
  <div class="box"><h2>Div 3</h2></div>
</div>
```

We now have two different classes—`box` and `boxspecial`. We alter our style sheet as follows, removing the `float` property from the `box` class, and adding a rule targeting the new `boxspecial` class we've just created:

```
.box
{
width: 100px;
height: 100px;
border: 1px solid #CCCCCC;
background-color: #333333;
}

.boxspecial
{
float: right;
width: 100px;
height: 100px;
border: 1px solid #CCCCCC;
background-color: #333333;
}
```

Let's try and anticipate what's going to happen. Div 1 has a class of `boxspecial`, so it's going to be floated to the `right`. Div 2 and Div 3, however, have no `float` specified, so they're going to sit within the normal flow. Let's take a look at how this renders in the browser, as shown in Figure 11-4.

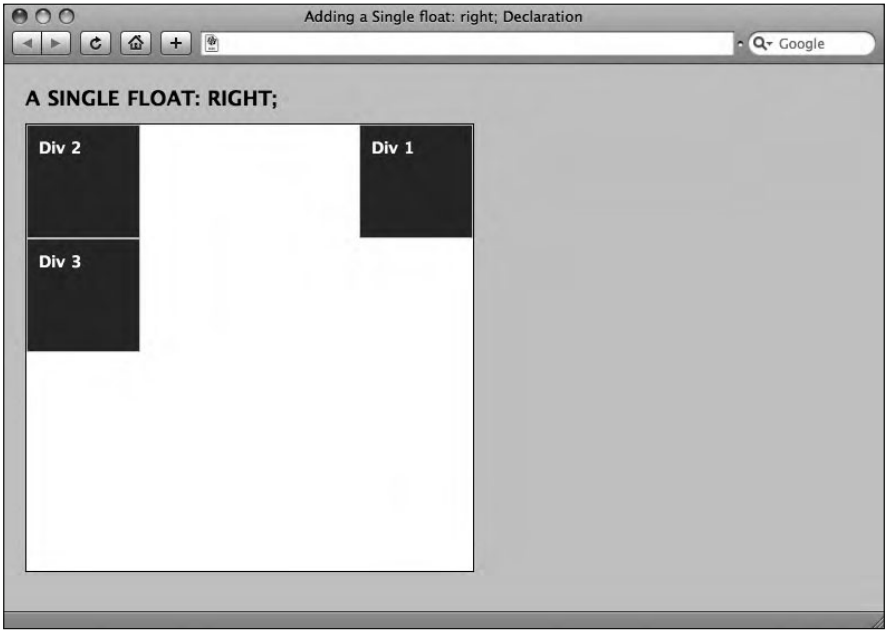


Figure 11-4. By specifiying `float: right;` for Div 1, and specifying no float for Div 2 and Div 3, Div 1 is removed from the normal flow of the document. Div 2 and Div 3, still within the normal document flow, ignore it completely.

As you can see, Div 1 has been floated to the right and *removed from the document flow*; this results in Div 2 and Div 3 occupying the space (at the top left) that Div 1 no longer occupies. Div 3 is dropping below Div 2, because it is no longer floated and is block-level.

If you return back to the idea that floated boxes are *removed from the document flow* and that boxes with no floats applied are *still within the normal document flow*, this should make sense. Div 2 and Div 3 are essentially ignoring Div 1 because it has been floated and removed from the document flow.

The preceding examples all deal with divs that are the same size. In reality, however, our different div elements are likely to be different sizes (as they will be when we deal with our two-column layout). Let's see how three divs of different size relate.

In the following example, we've amended our markup to give each div a different class (box1, box2, and box3), and we've written three rules, one for each class, that we add to our style sheet as follows:

```
.box1
{
float: left;
width: 160px;
height: 100px;
border: 1px solid #CCCCCC;
background-color: #333333;
}
```

```
.box2
{
float: left;
width: 160px;
height: 60px;
border: 1px solid #CCCCCC;
background-color: #333333;
}

.box3
{
float: left;
width: 160px;
height: 140px;
border: 1px solid #CCCCCC;
background-color: #333333;
}
```

We've now changed the size of all three divs; all are also now different heights. As in Figure 11-2, all have a `float: left;` declaration, so they are removed from the normal document flow. Figure 11-5 shows the result of our changes.

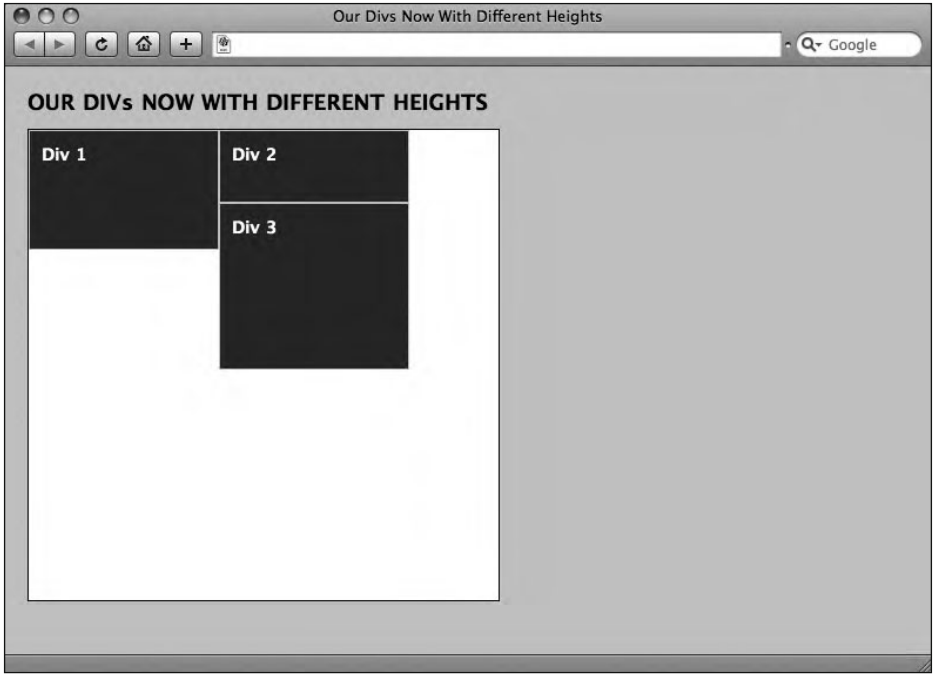


Figure 11-5. Div 1, Div 2, and Div 3 are now 160px wide, so they will not fit side by side within the 400px `container` div. Note how Div 3 occupies the first available position in the document flow, as far to the top left as it can go.

As we increased the `width` of our three `div`s to `160px`, they no longer fit side by side in the `400px container div`. The result is that Div 3 drops down to the next available space within the document flow, positioned as far to the top left as it can go. It's worth noting that this *isn't* below Div 1, but below Div 2, due to the space available beneath Div 2.

What if we'd like to drop Div 3 down so that it starts afresh on a new line and sits below Div 1? The answer is we use the `clear` property to clear the floats of Div 1 and Div 2 and instruct the browser to display Div 3 below them. (We'll use this technique in due course to clear our `footer` when we create the two-column layout for our upgraded King Kong page.)

We amend our CSS as follows:

```
.box3
{
float: left;
width: 160px;
height: 140px;
border: 1px solid #CCCCCC;
background-color: #333333;
clear: both;
}
```

The result of adding the `clear` property is shown in Figure 11-6.

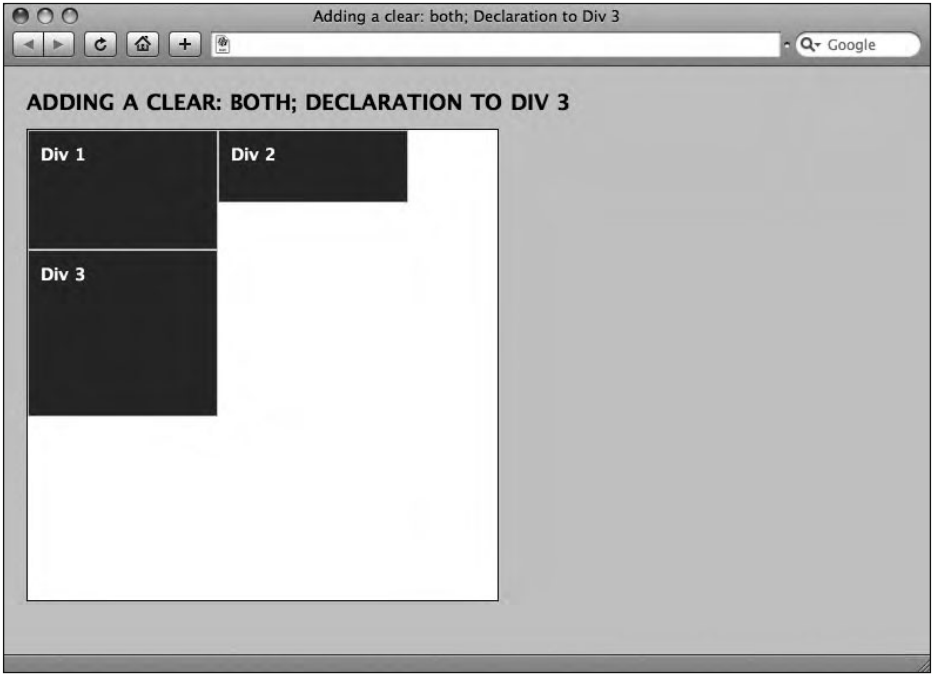


Figure 11-6. Adding a `clear: both;` declaration to the rule styling Div 3 clears both the Div 1 and Div 2 floats, dropping Div 3 beneath them.

Although all of our divs have been removed from the normal document flow by being set to `float: left;`, the addition of a `clear` declaration to Div 3 clears the above floats and positions Div 3 below Div 1 and Div 2.

In the next section we put the preceding examples into practice as we begin to build our two-column CSS layout using floats.

Applying floats to layouts

In the previous chapter we created a one-column layout and, in the process, subdivided our content into a number of divisions or divs. For the purposes of our one-column layout, we created `header`, `content`, and `footer` divs and nested them within a `container` div to hold everything together.

In this section we'll introduce a further div to our layout, a `sidebar`, which we'll use to gather some additional content to sit alongside our `content` div.

Simplified, we adjust our markup as follows, adding a `sidebar` div:

```
<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="sidebar">
      <!-- This is where the sidebar of the page is situated. -->
    </div> <!-- Closes the #sidebar div. -->

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

We now know that we can use CSS to float elements to the left or to the right. Because floated elements float until their outer edge meets the containing block they're situated within, or the outer edge of another floated element, we can nest our `header`, `content`, `sidebar`, and `footer` divs inside our `container` div and build a layout.

Figure 11-7 illustrates what we'll be building as we work through the examples in our two-column layout section. We'll float our `content` div to the left and our `sidebar` div to the right.

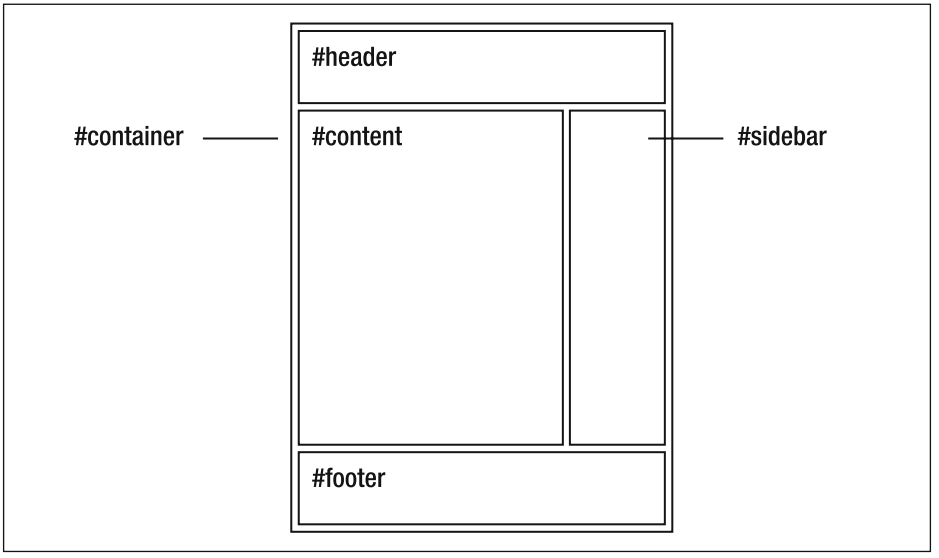


Figure 11-7. How our divs relate to each other once we add a `sidebar div` into the mix

It's worth noting, however, that, *regardless of the order our content appears in our markup*, we can float the `content` and `sidebar` divs in *either* direction, changing the visual order and display of the `content` and `sidebar` as we wish.

This allows us to organize our markup in the best way possible, putting the most important content first in the markup. As our divs can be floated in *either* direction, this means we can position the `content div` *before* the `sidebar div` in our markup, but use floats to visually position the `sidebar div` *before* the `content div` if we'd like to.

We write our markup in the optimum order, putting the `content`—which is more important than the `sidebar`—first in the markup order, as follows:

```
<body>
  ...

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

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

  ...
</body>
```

We can then control how these display in the browser by using floats. By floating the `content div` to the `left` and the `sidebar div` to the `right`, the content comes first in the visual display in the browser as in Figure 11-8.

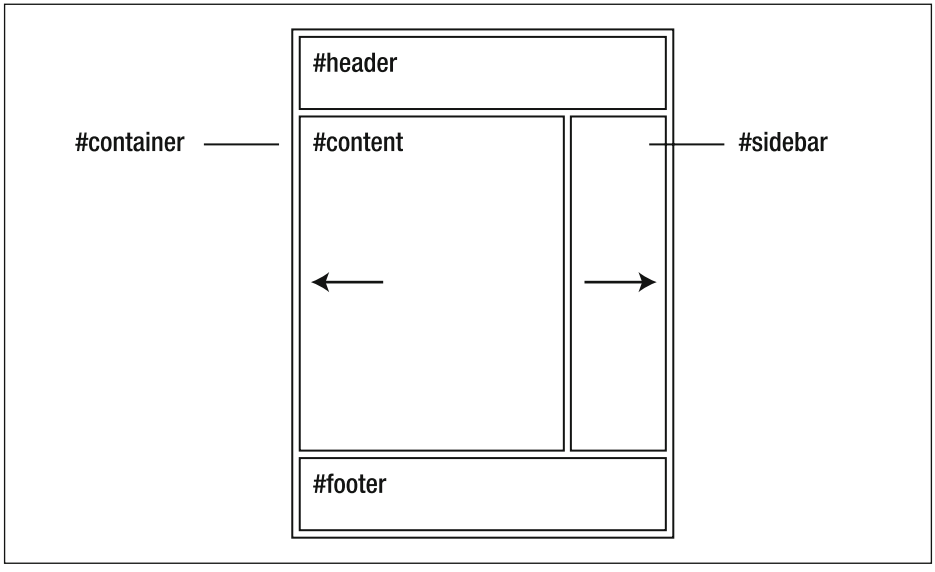


Figure 11-8. Floating the `content div` to the `left` and the `sidebar div` to the `right` changes the order our `divs` display in in the browser.

By simply changing the `float` values, we can float the `sidebar` to the left and the `content div` to the right, *without changing the order of our markup* as in Figure 11-9.

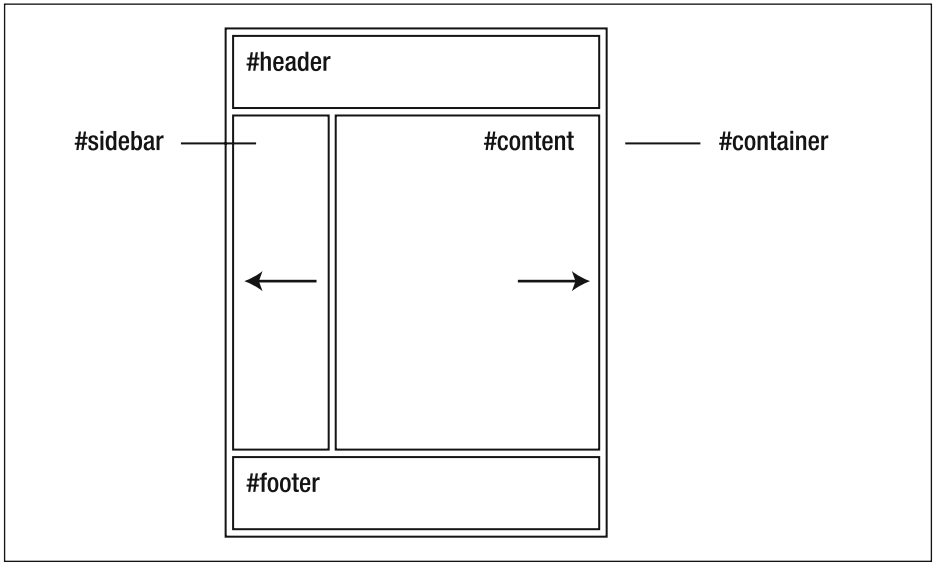


Figure 11-9. Simply switching the `float` values changes the order of our `divs` as they display in the browser.

In the next section, as we walk through the creation of our two-column layout, we'll show you how easy it is to switch our floats. Let's start building the two-column layout.

Creating our two-column CSS layout

We'll now take our markup with our `header`, `content`, `sidebar`, and `footer` divs nested within the `container` div and add some CSS to control our layout. This walkthrough develops the one-column layout we created in Chapter 10 and gives us a basic layout upon which we can build as we move forward.

We add the following rules to our style sheet:

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

#container
{
width: 790px;
background-color: #CCCCCC;
margin: 0 auto;
}

#header
{
padding: 10px 20px;
background-color: #333333;
text-align: center;
}

#content
{
width: 510px;
padding: 10px 20px;
background-color: #999999;
}

#sidebar
{
width: 200px;
padding: 10px 20px;
background-color: #666666;
}
```

```
#footer
{
clear: both;
padding: 10px 20px;
background-color: #333333;
text-align: center;
}

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

#header h1, #footer h1
{
color: #FFFFFF;
}
```

With no floats applied, our document, as structured here, with our `header`, `content`, `sidebar`, and `footer` divs nested in a `container` `div`, will render using the browser's normal document flow; that is, each `div` will display one after the other, each forcing a line break. You can see the result of this in Figure 11-10.

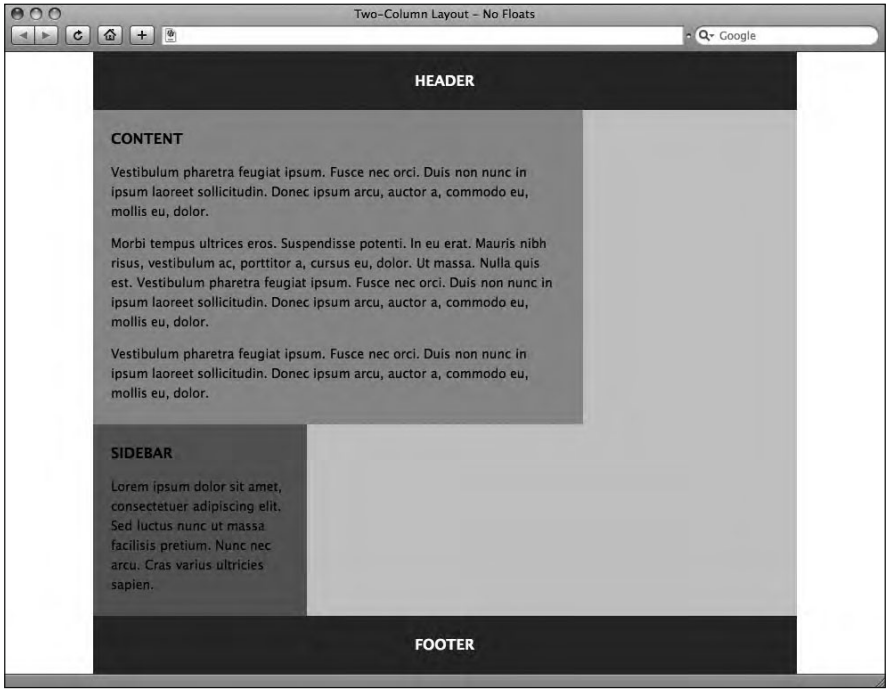


Figure 11-10. With no floats, our four nested block-level divs force line breaks and so display on separate lines.

Clearly this isn't what we want; we'd like our `content` and `sidebar` divs to float beside each other, with the `content` div floated to the `left` and the `sidebar` div floated to the `right`, as indicated in Figure 11-8 earlier. To do this we need to inform the browser to float the `content` and `sidebar` divs, and as you know from the preceding examples we can use the `float` property of CSS to align our `content` and `sidebar` divs to the `left` and `right`, respectively, effectively positioning them where we need them.

We do this by adding the following declarations to the `content` and `sidebar` rules in our style sheet:

```
#content
{
width: 510px;
padding: 10px 20px;
background-color: #999999;
float: left;
}

#sidebar
{
width: 200px;
padding: 10px 20px;
background-color: #666666;
float: right;
}
```

The result of adding these floats is shown in Figure 11-11.

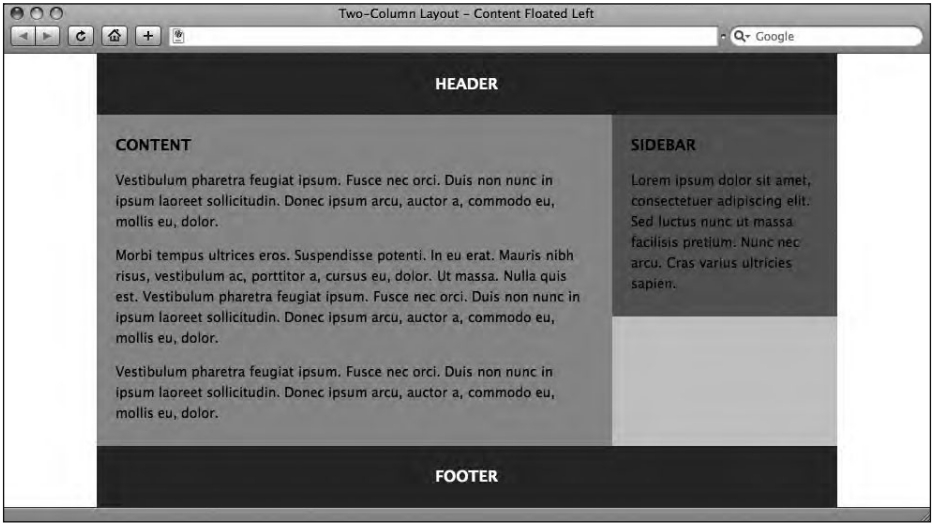


Figure 11-11. Adding `float` declarations to both the `content` and `sidebar` divs positions our `div` elements as we'd like.

What we've done is float our content to the left. Left-floated elements will move as far to the left as possible until they are in contact with the edge of the containing element (or another floated box); in this case the containing element is our container div.

Next, we've floated the sidebar to the right-hand edge of the container. Following the same principal as the left float, this pushes the sidebar as far to the right as possible, until it touches the edge of the container. As seen in the CSS markup earlier, we have explicitly specified widths on both the sidebar and content divs.

> *It's important to specify widths on floated elements. In theory a floated element without a specific width should shrink to be as wide as the content within it; however, how this is interpreted can vary between browsers. As a rule of thumb:* always specify a width on floated elements. *The only exception to this rule is when floating images, which, by their nature, already occupy a specific width.*

Imagine, however, that we'd like the sidebar div to display *before* the content div. We don't need to adjust the order of our markup; all we need to do is change our floats as follows:

```
#content
{
width: 510px;
padding: 10px 20px;
background-color: #999999;
float: right;
}

#sidebar
{
width: 200px;
padding: 10px 20px;
background-color: #666666;
float: left;
}
```

The result of changing the float value on the content and sidebar divs is shown in Figure 11-12.

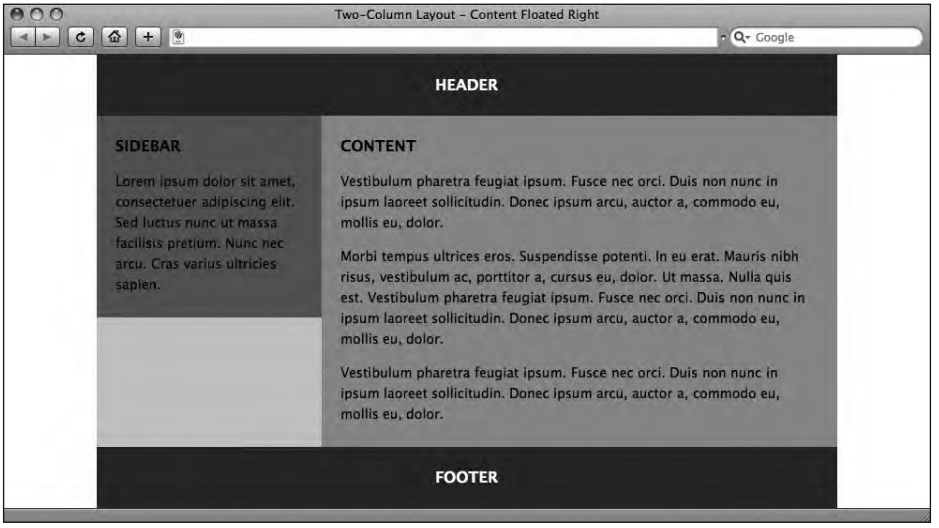


Figure 11-12. Changing the `content` and `sidebar` `float` values changes the order the `div`s display in within the browser.

Simply changing the values of the floats changes the visual display within the browser. This is a useful point to note as it allows us to create our markup in the most logical (and useful) order and then control the order of its display using CSS.

Another thing to note in our examples is the height of the `sidebar div`. We haven't specified a height for the `sidebar div`, and as you can see in Figure 11-12, the `sidebar` is only as tall as it needs to be to accommodate the content within it. This is the natural behavior of block-level elements: unless a height is specified, a block-level element will only ever occupy as much height as it needs.

As a consequence of this, the `background-color` of our `container div`, inside which our four other `div`s are nested, shows through below the `sidebar div`. We'll cover a workaround for this later in this chapter when we introduce the concept of **Faux Columns**, a term coined by noted Standardista Dan Cederholm while writing for *A List Apart* in 2004.

One final point to note: by default, elements following other floated elements will try to wrap around them, much like a paragraph of text wrapping around an image in a newspaper article. As we want our `footer` (which has no `width` set on it) to display *below* both the `content` and `sidebar` `div`s, we apply a `clear: both;` CSS declaration to the `footer` rule. This moves the `footer div` below all of the preceding floated elements on our page.

It's worth noting that floats can be applied to other elements, not just `div`s like our `content` and `sidebar`. Later in the chapter, in the section "Applying a float to an image," we'll take a look at this in action and see how we can use floats to control the display of images on our web pages. But first, a little mathematics . . .

Calculating the width of your elements

As we move on to two-column layouts, one thing worth noting is the importance of calculating the total `width` our elements occupy when floated side by side to ensure that they fit within their containing boxes. This once again brings us to the box model.

We touched on the box model—how we calculate the width of boxes in CSS—in Chapter 10, but it's worth a short refresher here in the context of our new two-column layout. Now that we're dealing with our `content` and `sidebar` `div`s sitting side by side nested in a `container` `div`, a recap of the box model is well worth including.

A short box model recap

Let's revisit the box model briefly; it has a bearing on two-column layouts (and more advanced multicolumn layouts) so we want to ensure you fully understand how nested `div`s relate to the `div`s they may be nested within.

As illustrated in Figure 11-13, the total width a box occupies in CSS is calculated by adding the declared width of the element and adding any margins, borders, and padding.

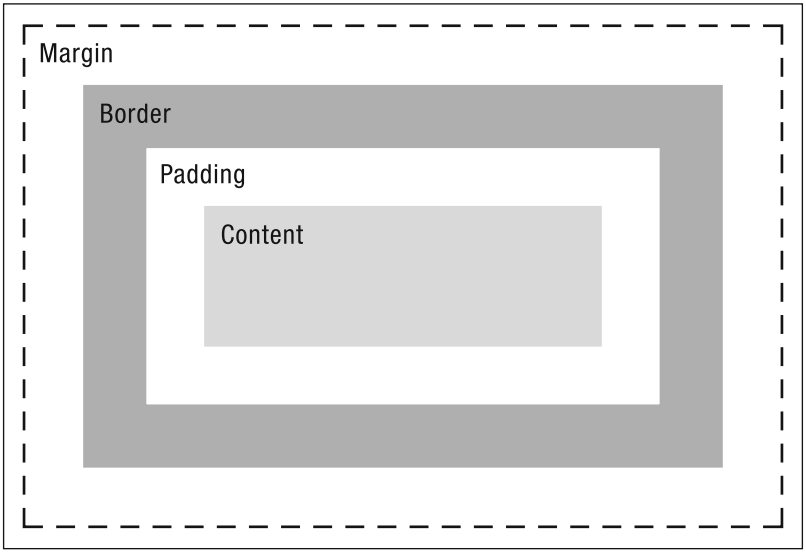


Figure 11-13. The box model we acquainted you with in Chapter 10

Calculating the total width boxes occupy is the point at which beginners often stumble. They give a `div` a `width`, and assume that this is the *total width* the `div` occupies, forgetting to add any margins, borders, and padding.

Let's apply the box model theory we introduced in Chapter 10 to our two-column layout. Cue a little mathematics.

We'd like our content and sidebar divs to sit side by side within the container div. To do this we need to allow enough room in our container div to accommodate these divs. Let's take a look at the figures. First let's look at the container, content, and sidebar rules in our CSS, in particular the relevant measurements. These are as follows:

```
#container
{
width: 790px;
background-color: #CCCCCC;
margin: 0 auto;
}

...

#content
{
width: 510px;
padding: 10px 20px;
background-color: #999999;
float: left;
}

#sidebar
{
width: 200px;
padding: 10px 20px;
background-color: #666666;
float: right;
}
```

Our container div is 790px wide. Our content and sidebar divs (with any added margins, borders, and padding) need to fit within this div, so we need to ensure their combined width is less than 790px. Our content div is 510px wide with 20px of padding on the left and right (remember our CSS shorthand—padding: 10px 20px;—establishes a padding-top and padding-bottom of 10px, and a padding-right and padding-left of 20px). Our sidebar div is 200px wide with 20px of padding on the left and right. Figure 11-14 shows how these measurements add up.

Let's take a look at the measurements. The value of our container width is 790px. Our content and sidebar divs have no margins or borders; this has the added benefit of making our calculations a little bit easier. We need the content and sidebar divs to fit inside the container div.

The total width the content div occupies is as follows:

padding-left + element width + padding-right or 20px + 510px + 20px = 550px

The total width the sidebar div occupies is as follows:

padding-left + element width + padding-right or 20px + 200px + 20px = 240px

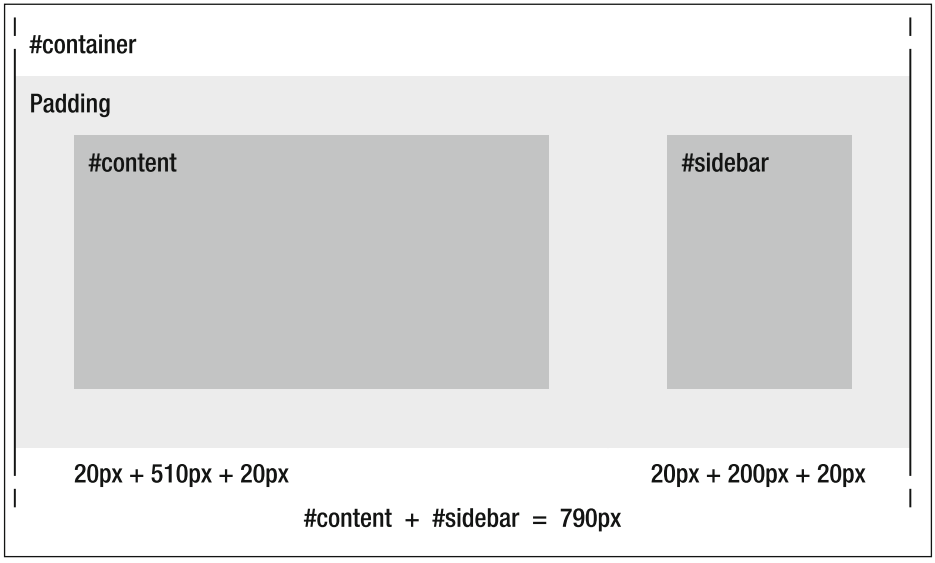


Figure 11-14. Calculating the actual widths of our `content` and `sidebar` divs, including any applied margins, borders, and padding, gives us the total width of `790px` that we need to allocate for our `container` div.

This gives us a combined `width` for our `content` and `sidebar` of `790px`, so the value of our `container` `width` is perfect. In the following section we'll show you what happens if the divs we're nesting occupy more space than the element that contains them, demonstrating the importance of getting your calculations right before you begin building your layouts.

What happens when your elements are too wide?

What happens when the `div` elements we're nesting occupy more space than the divs they're nested within? Let's take a look.

We've created two pages; both have a `container` `div` with a `width` of `790px`. On the left we've repeated Figure 11-11 where the combined `width` of our `content` and `sidebar` divs fits neatly within the `container` `div` they're nested within. On the right, however, we've increased the `width` of our `sidebar` from `200px` to `240px`, increasing the combined `width` of the `content` and `sidebar` divs to `830px`, too large for our `790px` `container` `div`.

The result is that the `sidebar` `div` drops beneath the `content` `div` and is floated to the left as specified in the rule that is controlling it. You can see the two results in Figure 11-15.

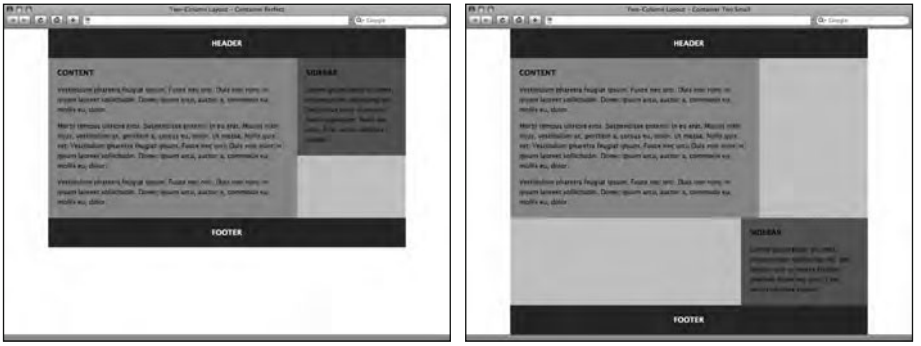


Figure 11-15. On the left our `content` and `sidebar` divs fit within our `container` div. On the right the combined `width` is too large to fit side by side, forcing the `sidebar` to display below the `content` div.

Let's take a look at the CSS behind the example on the right, simplified for brevity:

```
#container
{
width: 790px;
}

...

#content
{
width: 510px;
padding: 10px 20px;
float: left;
}

#sidebar
{
width: 240px;
padding: 10px 20px;
float: right;
}
```

Again, we need the `content` and `sidebar` divs to fit inside the `container` div. Let's take a look at the figures. The total width the `content` div occupies is as follows:

`padding-left` + element `width` + `padding-right` or 20px + 510px + 20px = 550px

The total width the `sidebar` div occupies is as follows:

`padding-left` + element `width` + `padding-right` or 20px + 240px + 20px = 280px

This gives us a combined `width` for our `content` and `sidebar` of 830px, so the value of our `container` `width`—specified as 790px—is too small.

The result is that the sidebar, which we'd like to float to the right of our content, floats to the right, but *beneath* the content. When a floated box is taken out of the normal flow of the document, it will move as far to the left or right as possible, remaining in the same position horizontally. However, if there isn't enough space to accommodate the floated box, it will move downward line by line until there is room for it—in this case, sitting beneath the content div.

You can take a look at the source code for both of the preceding web pages at the Web Standardistas web site:

```
www.webstandardistas.com/11/container_perfect.html
www.webstandardistas.com/11/container_too_small.html
```

Throughout the last section we looked at the importance of calculating widths when creating layouts. In the next section we'll look at the measurement of heights, in particular introducing you to the topic of collapsing margins.

Collapsing margins

One aspect of CSS-based layouts that can prove a little confusing for the beginner is the concept of **collapsing margins**. To explain what collapsing margins are and how they work, let's revisit the example we created in Chapter 10 to show how margins, borders, and padding work when applied to elements.

You might recall that the example we created in Chapter 10 demonstrated the effect of applying margins, borders, and padding to a single paragraph. We'll expand upon that example now and use two paragraphs to see how they interrelate, looking in particular at how the vertical margins between them interrelate.

To start with we've created a very short web page with two short paragraphs. 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;
margin: 0;
}

p
{
width: 400px;
line-height: 1.5;
background-color: #CCCCCC;
margin: 20px;
}
```

Our two paragraphs styled with these simple rules are displayed in Figure 11-16.

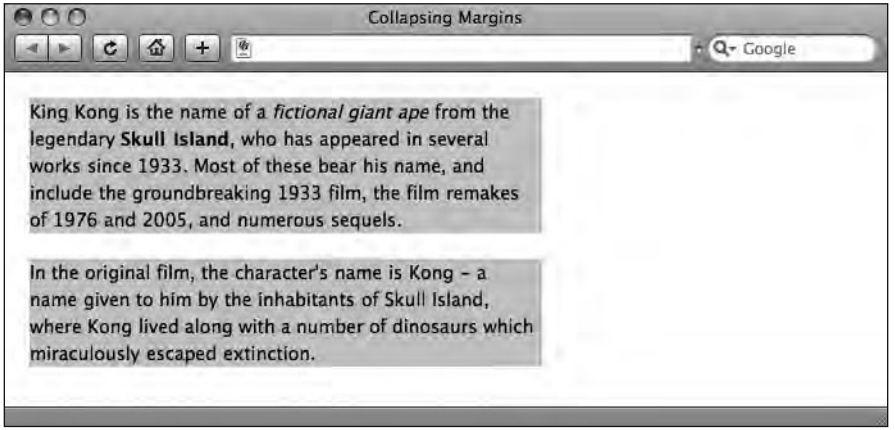


Figure 11-16. Note the distance between the top edge of the browser and the first paragraph (20px). You might be forgiven for wondering why the margin between the two pararaphs is 20px and not 40px.

In the rules we established earlier, we set a `margin: 20px;` on our p elements. As we know from Chapter 10, this CSS shorthand sets a `margin` on *all four sides* of our p elements of 20px. However, as you can see from Figure 11-16, the distance between the top left edge of the browser and the first paragraph appears to be 20px, yet the distance between the two paragraphs appears to be the same, also 20px.

Surely, if we've set a `margin` on *all four sides* of our paragraphs of 20px the distance between the two paragraphs should be 40px? (20px `margin-bottom` on the first paragraph plus 20px `margin-top` on the second paragraph equals 40px.) Why is the space between the two paragraphs only 20px?

The answer to this conundrum is that we are witnessing collapsing margins in action. Put simply, when the top and bottom margin of two elements within the normal document flow touch vertically, the smaller of these margins collapses to zero, leaving only the larger of the two margins separating the elements.

But why do collapsing margins collapse in the first place?

The reason this feature was implemented is that on most occasions, collapsing margins make perfect sense from a design perspective, making your style sheets easier to write, saving you from having to carefully calculate the top and bottom margin on each of the elements of your page in relation to each other to achieve consistent margins between the different elements in your layout.

Let's expand on our preceding example to illustrate why collapsing margins can be helpful when creating layouts. We've created two web pages in which we've added some additional elements to our previous example, adding an h1 element to both and including a couple of additional paragraphs. We've set a `margin: 20px;` on both the h1 and the p elements.

In Figure 11-17, the example on the left shows how collapsing margins should work, resulting in even spacing between our elements. In the example on the right we've added an additional class to override the collapsing margins and demonstrate how the same web page would look if the margins didn't collapse. The doubled-up margins between the elements result in too much vertical space separating the elements.

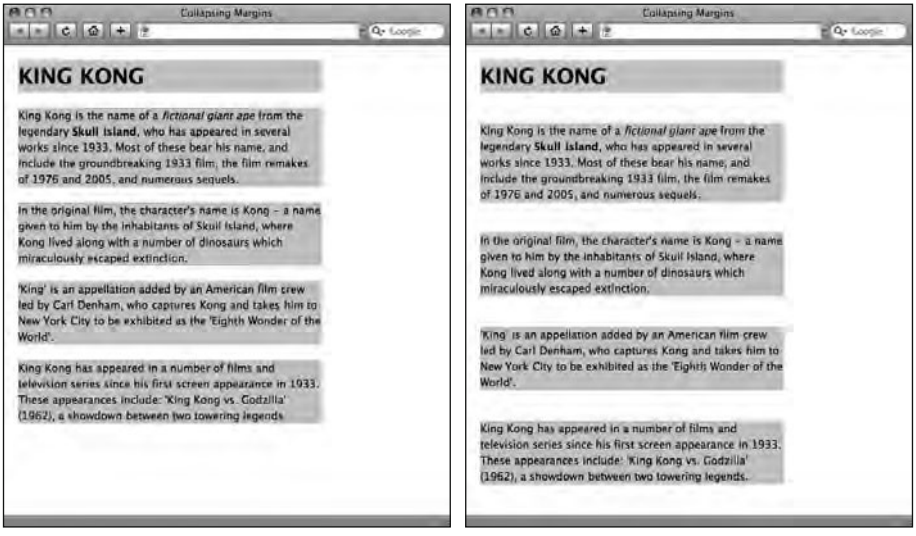


Figure 11-17. The example on the left shows how collapsing margins work normally, improving the visual consistency of the design. The example on the right shows how the page would look if the margins didn't collapse.

Collapsing margins can be a difficult topic to grasp at first, but hopefully these examples will give you some basic understanding of the principles. We recommend the chapter "Visual Formatting Model Recap" in *CSS Mastery: Advanced Web Standards Solutions* by Andy Budd (friends of ED, 2006) for anyone wishing to earn a black belt in advanced CSS visual formatting models.

Applying a float to an image

In the section "Creating our two-column CSS layout" earlier in this chapter, we promised we'd show you how to apply floats to elements other than `div` elements. In this section we'll look at applying floats to images.

In the earlier examples in this chapter, we floated our `content` and `sidebar` `div`s. In this example, we've created a simple page layout where we've added an image to our `content` `div`; we'll use a `float` to control the relationship of the image to the remainder of the content within the `content` `div`. Before we look at how this image displays in the browser, let's take a look at the markup we're using. Our simplified markup is as follows:

```
<div id="content">
  <h1>Content</h1>
  <img src="tall_squirrel_monkey.jpg" width="150" height="350"
  alt="Gordo prepares to meet Miss Baker." />
  <!-- This is where the remaining paragraphs are situated. -->
</div><!-- Closes the content div -->
```

By default the image is positioned within the normal document flow as shown in Figure 11-18.

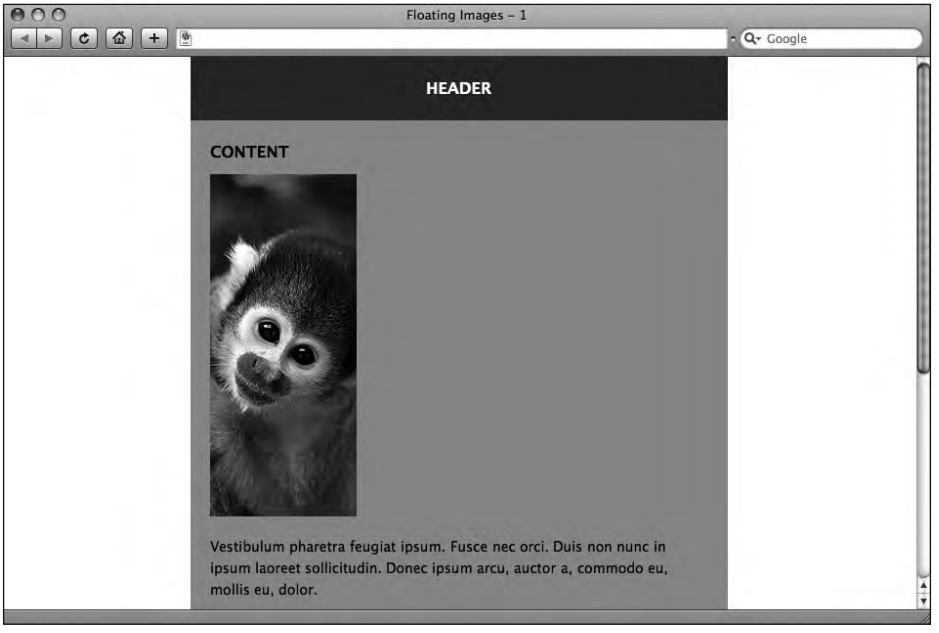


Figure 11-18. With no floats our image is placed within the normal flow of the document.

By default the image is positioned within the normal document flow with our paragraphs appearing immediately after it. However, what if we'd like to wrap our text around the image? Good news—we can do this by applying a `float` to our image.

In Chapter 10 we introduced a `class` of `portrait` for our primates' different portraits; in this example we have a tall, tightly cropped image. We'll create a class for any tall primate images like this one, by adding the following CSS rule:

```
.tallprimate
{
float: left;
padding: 10px 10px 10px 0;
}
```

We then add the class to our markup as follows:

```
<img src="cropped_squirrel_monkey.jpg" width="150" height="350"
alt="Gordo prepares to meet Miss Baker." class="tallprimate" />
```

What this rule does is target any images on our web page with a `class` of `tallprimate` and float them to the left. As you can also see, we've added `10px` of `padding` (to the `top`, `right`, and `bottom`) of our `tallprimate` class to create some space between the image and the paragraph text that will now flow around it.

You can see the results of adding these rules in Figure 11-19.

Figure 11-19. Applying a `float: left;` results in our image being removed from the normal document flow and the text wrapping around it.

Perfect, the paragraph text is now wrapping around the image just as we'd like. However, what happens if the floated element, in this case our image, is taller than its containing element? In other words, what would happen if we would remove half of our paragraphs? Let's take a look.

We remove all but two of our paragraphs from our `content div`. The result of this is shown in Figure 11-20.

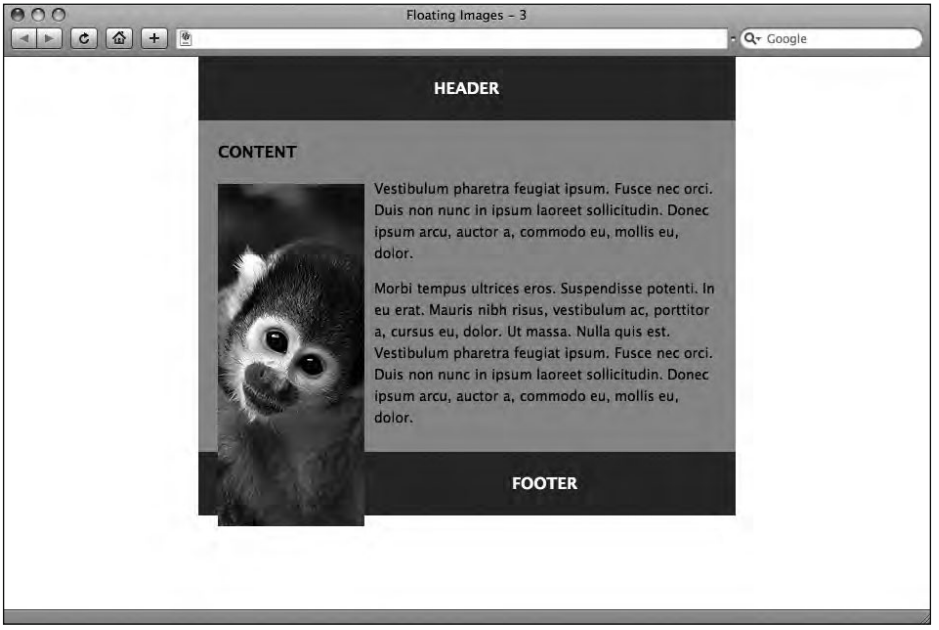


Figure 11-20. By removing our `img` with the `tallprimate class` from the normal document flow, it is not counted when the height of the containing element is calculated, which is now only tall enough for the two paragraphs.

Needless to say, this isn't so great. Why is the image breaking out of the layout?

The first thing to note is that our image is placed inside our `content div`. The height of the `content div` is determined by the height of the content inside it (unless we have specified a height for the element in the CSS). Since we've floated the image by specifying `float: left` in our CSS, this image is taken out of the normal document flow and as a result is not taken into account when the browser calculates the height of the `content div`. Instead, the height of the `content div` is determined by the two paragraphs of text, which have no floats applied.

The end result is that our floated image element now breaks out of our `content div` and into the `footer`. This is clearly not what we want, so we need to add a few additional declarations to our CSS to solve the problem.

Earlier in the chapter we introduced you to the `clear` property, which we can use to clear floats. By adding a `clear` declaration to the `footer div`, we're taking care of our `footer` problem. We add the following declaration to our `footer` rule:

```
#footer
{
...
clear: both;
}
```

Adding this rule results in the changes shown in Figure 11-21.

Figure 11-21. Adding a `clear: both;` declaration drops the footer beneath the content above it.

Applying a `clear: both;` declaration to an element moves it below all preceding floated elements. This results in our image no longer breaking into the `footer div`; however, our image is still breaking out of its containing element, the `content div`. We'll resolve that issue next . . .

The `clear: both;` declaration that we applied to the `footer` won't help us here as the `clear` property moves an element to a position *below* the preceding floated elements, and our image is placed not *above* but *inside* the `content div`.

This looks like a sticky situation, and over the years many solutions involving advanced CSS trickery and extra XHTML markup have been devised to crack this particular nut. There is, however, one easy way of solving this problem. Applying one simple CSS declaration to the containing element (in this case our `content div`) will ensure that the floated element no longer breaks out of its container. We add the following declaration to our `content` rule:

```
#content
{
...
overflow: auto;
}
```

Figure 11-22 shows how our page looks after applying this declaration to our `content div`.

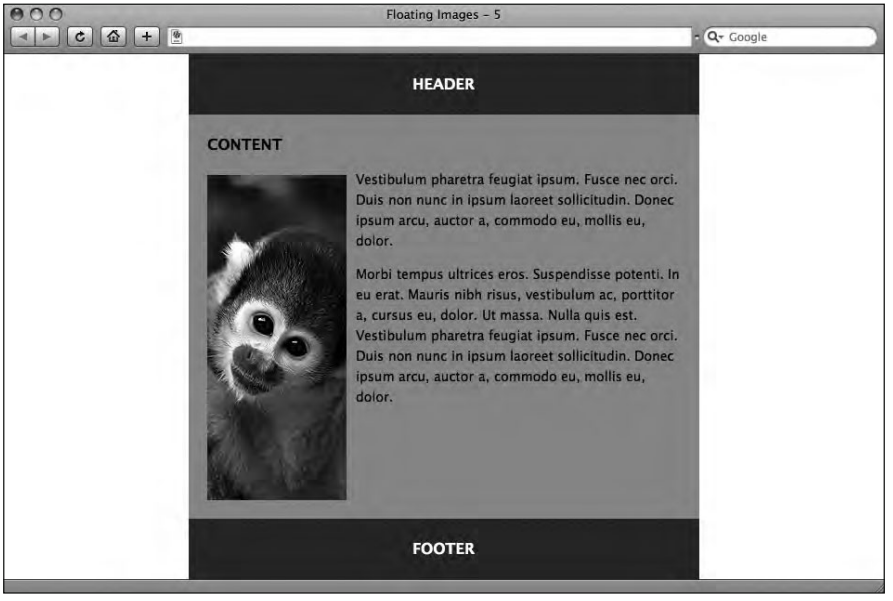


Figure 11-22. Applying `overflow: auto` to the `content div` ensures our floated image no longer breaks out of it—perfect.

That's it, a combination of `float: left;` to float our image to the left, `clear: both;` to clear the footer, and `overflow: auto;` to ensure our `content div` allows for the image being larger than the paragraph text in the `content div` solves the problem.

The CSS `overflow` *property determines how content inside a* `div` *or other block-level element should be displayed if its width or height exceeds that specified for the containing element. The* `overflow: auto;` *declaration serves the primary purpose of instructing the browser to add scrollbars if the dimensions of the containing element are less than the declared dimensions of the content within. As a side effect, applying this declaration has the benefit of solving our issue with a floated element breaking out of its container.*

Faux Columns

One of the frustrating aspects of creating multicolumn CSS layouts is the fact that our columns only expand to fill the content that they occupy. In Figure 11-23, we've created two typical web pages, one that has a long `content div` and a short `sidebar div`, and a second that has a long `sidebar div` and a short `content div`.

Ideally we'd like the backgrounds of the shorter columns to extend down so that they occupy the same vertical space as the longer columns. As things stand the `div`s only occupy as much space as the content within them. There is, however, a deceptively simple solution to the problem using a technique known as Faux Columns (a term originally coined by bon vivant Dan Cederholm in a 2004 article for *A List Apart*).

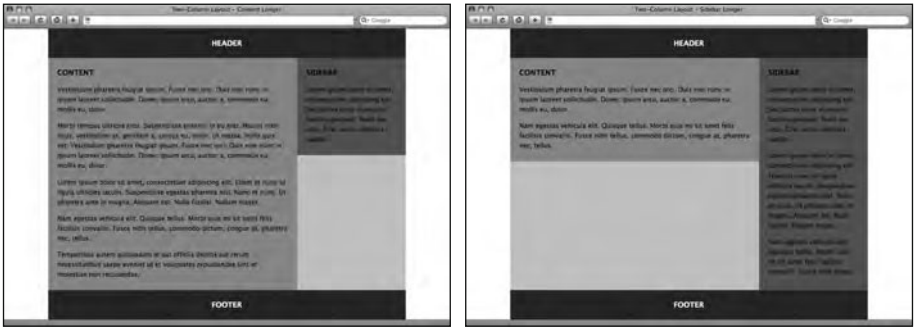


Figure 11-23. The backgrounds on the shorter columns only occupy as much vertical space as the respective columns' contents.

By defining a `background-image` for our `container div` that we tile vertically behind the two columns, we can create the illusion that both columns are the same height. This is best demonstrated with an example.

With no `background-image` defined on our `container div`, our page displays as shown in Figure 11-24.

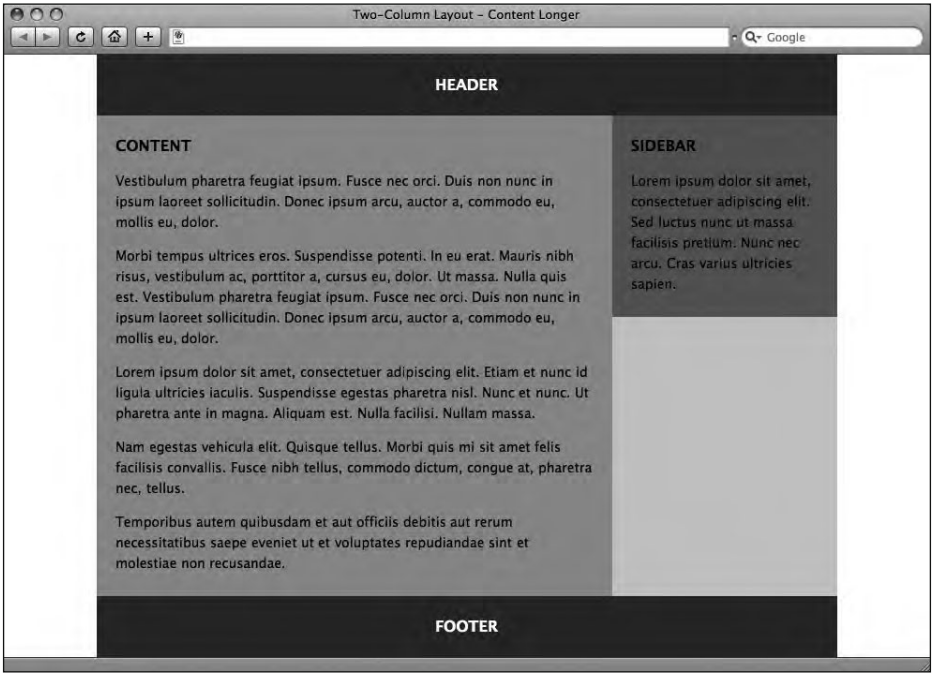


Figure 11-24. Our layout with no `background-image` applied to the `container div`. The result is that the columns are of unequal height.

The first thing we need to do is to create a background-image that is 790px wide (the width of our container div) and that visually indicates the two columns. We've created a 790px × 10px image as shown in Figure 11-25.

Figure 11-25. Our background-image, which we'll tile vertically within the container div, gives the illusion of two columns.

We adjust our container, content, and sidebar divs as follows, adding the background-image to the container and removing the background-color on both the content and sidebar divs:

```
#container
{
width: 790px;
background-color: #CCCCCC;
margin: 0 auto;
background-image: url(container_bg.png);
background-repeat: repeat-y;
}

...

#content
{
width: 510px;
padding: 10px 20px;
float: left;
}

#sidebar
{
width: 200px;
padding: 10px 20px;
float: right;
}
```

The result of amending our container, content, and sidebar rules is shown in Figure 11-26.

Figure 11-26. Our `background-image` tiles vertically within the `container div`, giving the illusion of two columns of equal height.

That's it. Simply adding a `background-image` to the `container div` and tiling it vertically using the `background-repeat` property gives the impression of two columns of equal height.

Wrapping up with King Kong

By tying together the different elements we've covered in this chapter, we can create a two-column CSS layout for our King Kong page, using floats to control the layout of the `content` and `sidebar divs`. In Figure 11-27, we've combined the two-column CSS layout walkthrough with our Faux Columns walkthrough to create a two-column version of our of King Kong web page. By applying a `background-image` to the `container div` that features a subtle gradient effect, we've distinguished both the `content` and `sidebar divs`, visually differentiating them. All of this is built on a solid foundation of well-structured and well-formed markup.

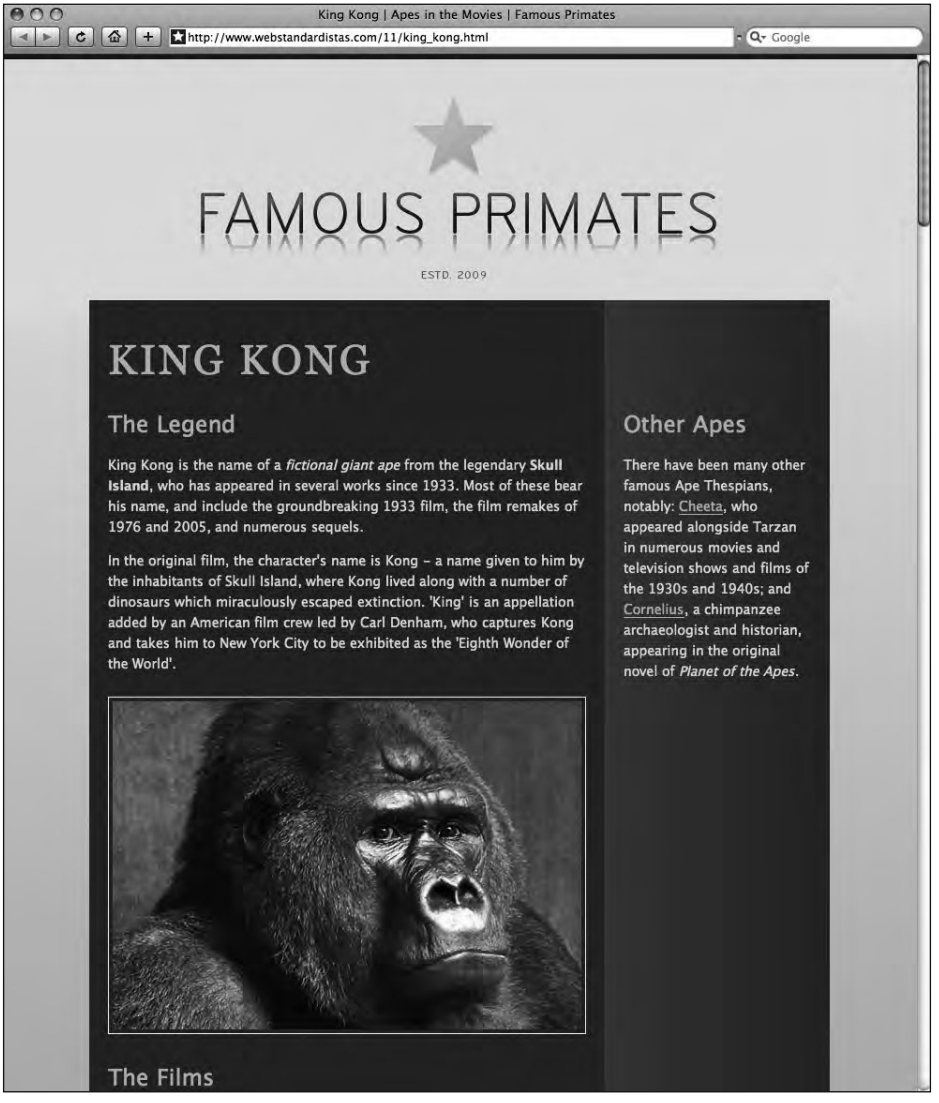


Figure 11-27. By drawing together the different topics covered in this chapter our King Kong page is really beginning to take shape.

Our King Kong page is now really coming together and, as you'll see in the next chapter, we'll repurpose our `sidebar div` to act as the perfect location for our Famous Primates web site's navigation, adding the web site's one missing ingredient.

Summary

So what have we covered? In this chapter we introduced you to two-column CSS layouts, demonstrating how we can use the `float` property to remove elements from the ordinary flow of our document. We used a float-based approach as the basis for creating a two-column CSS layout.

We also refreshed your memory of the box model, particularly looking at its importance when working out widths when nesting `div` elements. Building on our walkthough from Chapter 10 on adding margins, borders, and padding, we introduced the topic of collapsing margins, demonstrating how they can make the layout process a little easier.

Finally, we covered the topic of Faux Columns, showcasing how we could creatively use a `background-image` to visually distinguish our `content` and `sidebar divs`.

In the next chapter we'll build on this layout by replacing the `h2` and `p` elements in our `sidebar` with a navigation list styled using CSS. This will form the backbone of our Famous Primates web site's navigation. Good times.

Homework: Adding a second column to Gordo's web page

In this chapter our primary focus was the creation of a two-column CSS layout. Building upon the one-column CSS layout we created in Chapter 10, we introduced you to floats, explaining how they can be used to remove elements from the normal document flow. Once we'd demonstrated how floats worked, we showed you how to create a two-column float-based layout and used Faux Columns to create the illusion that your `content` and `sidebar divs` were of equal height.

This chapter's homework is to add a second column to your Gordo web page and include some content we've provided you to temporarily occupy the `sidebar` until we repurpose it in Chapter 12. Once you've added your second column, we'd like you to add Faux Columns to your `container div` to tie together the `content` and `sidebar divs` of your Gordo page.

1. Add a sidebar div

The first change we'd like you to make to your Gordo page is to add a `sidebar div` to your XHTML markup. Add this after your `content div` and before the `footer div` as demonstrated in our King Kong example in the chapter.

As with the previous chapters we've created everything you need to complete the homework, including providing some ready-made content for your brand-new `sidebar div` so you can focus on creating the two-column layout. You can download the assets here:

`www.webstandardistas.com/11/assets.zip`

Once you've downloaded the preceding files, transfer the Faux Column image we've supplied to your images folder, you'll be using it shortly. Add the content we've provided to your new sidebar div; we'll leave it as an exercise for you to add the links to the Albert I and Miss Baker pages where appropriate.

2. Add a sidebar rule to your style sheet

Now that you've created a sidebar div for your Gordo page, you'll need to add a rule to apply some style to it in your style sheet. We'll fill the rule with some declarations shortly. First we'll need to pause for a little mathematics.

3. Measure up the divs

In the last chapter, when we were creating our one-column layout, we only needed to set a width on our container div; the other divs nested within the container expanded to fill the available space within it. In this chapter, however, we'll need to consider the relative widths of the container div to the content and sidebar divs.

As you'll have noticed from our example in this chapter, you'll need to widen your container div and set a width on both the content and sidebar divs, ensuring that the container div is wide enough to accommodate your two columns. We could ask you to take a look at the figures from our example earlier in the chapter, but we'll save you some time and provide them for you.

The first thing you'll need to do is widen the container; remembering it needs to fit the content and sidebar divs, increase its width to 790px. You'll now need to add a width to your content and sidebar rules. Add a width: 510px; declaration to the content div and a width: 200px; declaration to the sidebar div.

4. Add the floats

In our example we floated our content div to the left and our sidebar div to the right. Add float declarations to both your content and sidebar rules and float them to the left and right, respectively.

You might also like to try floating the content to the right and the sidebar to the left and refreshing the page. This should help underline that the order of your markup can be changed visually with your CSS.

5. Add padding to your sidebar

Once you've floated your content and sidebar divs, we'd like you to add a declaration to your sidebar rule to adjust its padding. Pay particular attention to the padding at the top of the sidebar to ensure the h2 in the sidebar lines up with the h2 in the content div. You might like to refer to our example as you do this. You'll also notice from our example that we've added 20px of padding to the left and right of our sidebar; this is to ensure everything adds up to fill the new 790px-wide container div.

6. Add the Faux Columns

As with the previous chapters we've provided an image for you to use to create your Faux Columns. Following along with our example in the chapter, set the image provided as a `background-image` on your `container div` and use a `background-repeat: repeat-y;` declaration to repeat the image vertically within the `container div`. This will act as a background image to create the illusion that your `content` and `sidebar divs` are occupying the same vertical height.

As usual, to help you with the different stages of this chapter's homework, we've created our own, two-column CSS layout for King Kong featuring our newly added `sidebar div` and Faux Columns `background-image`. 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/11/king_kong.html
```

Once you've created your two-column CSS layout for your Gordo web page and added the Faux Columns `background-image` we provided, put the kettle on and enjoy a cup of *Java Malabar* as you prepare yourself for the next chapter.