CHAPTER 3

Understanding the CSS Box Model and Positioning

What You'll Learn in This Chapter:

- How to conceptualize the CSS box model
- How to position your elements
- How to control the way elements stack up
- How to manage the flow of text
- How fixed layouts work
- How fluid layouts work
- How to create a fixed/fluid hybrid layout
- How to think about and begin to implement a responsive design

In the preceding chapter, you learned a lot about the basic structure and syntax of HTML and CSS. In this chapter, you'll take that knowledge one step further and learn about the CSS box model, which is the guiding force behind the layout of elements on your screen—be it a desktop or mobile display.

It's important to spend some time focusing on and practicing working with the box model, because if you have a good handle on how the box model

works, you won't tear your hair out when you create a design and then realize that the elements don't line up or that they seem a little "off." You'll know that, in almost all cases, something—the margin, the padding, the border—just needs a little tweaking.

You'll also learn more about CSS positioning, including stacking elements on top of each other in a three-dimensional way (instead of a vertical way), as well as controlling the flow of text around elements using the `float` property. You'll then build on this information and learn about the types of overall page layouts: fixed and fluid (also referred to as liquid). But it's also possible to use a combination of the two, with some elements fixed and others fluid. We end the chapter with a brief mention of responsive web design, which is an important topic upon which entire books have been written.

The CSS Box Model

Every element in HTML is considered a "box," whether it is a paragraph, a `<div>`, an image, or anything else. Boxes have consistent properties, whether we see them or not, and whether the style sheet specifies them or not. They're always present, and as designers, we have to keep their presence in mind when creating a layout.

Figure 3.1 is a diagram of the box model. The box model describes the way in which every HTML block-level element has the potential for a border, padding, and margin and, specifically, how the border, padding, and margin are applied. In other words, all elements have some padding between the content and the border of the element. Additionally, the border might or might not be visible, but there is space for it, just as there is a margin between the border of the element and any other content outside the element.

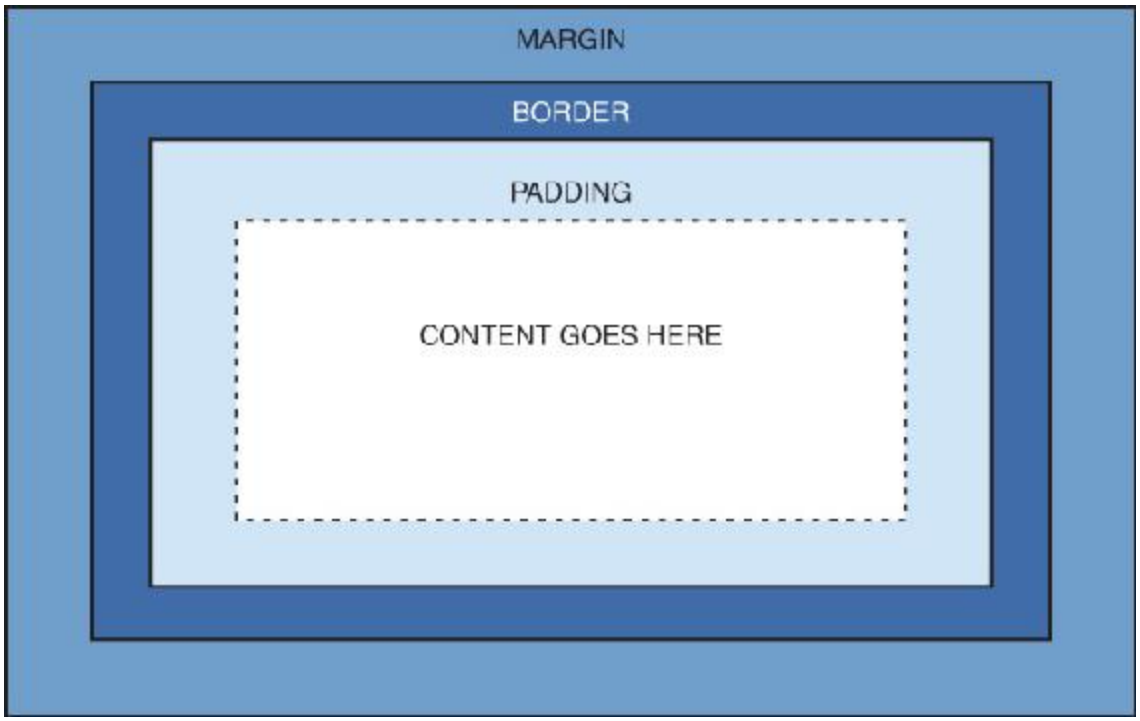

FIGURE 3.1
Every element in HTML is represented by the CSS box model.

Here's yet another explanation of the box model, going from the outside inward:

- The *margin* is the area outside the element. It never has color; it is always transparent.
- The *border* extends around the element, on the outer edge of any padding. The border can be of several types, widths, and colors.
- The *padding* exists around the content and inherits the background color of the content area.
- The *content* is surrounded by padding.

Here's where the tricky part comes in: To know the true height and width of an element, you have to take all the elements of the box model into account. Think back to the example from the preceding chapter: Despite the specific indication that a `<div>` should be 250 pixels wide and 100 pixels high, that `<div>` had to grow larger to accommodate the padding in use.

You already know how to set the width and height of an element using the `width` and `height` properties. The following example shows how to define a `<div>` that is 250 pixels wide and 100 pixels high, with a red background and a black single-pixel border:

```
div {
  width: 250px;
  height: 100px;
  background-color: #ff0000;
  border: 1px solid #000000;
}
```

Figure 3.2 shows this simple `<div>`.

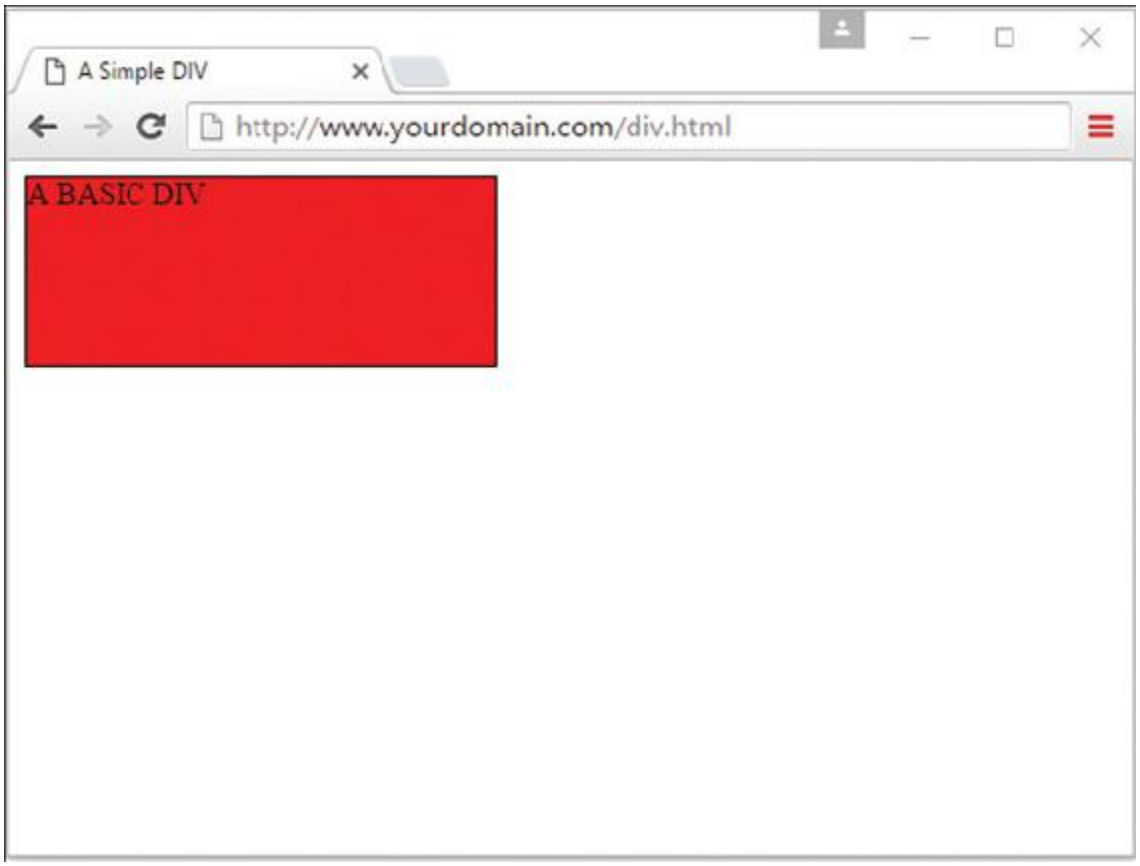

FIGURE 3.2
This is a simple styled `<div>`.

If we define a second element with these same properties, but also add `margin` and `padding` properties of a certain size, we begin to see how the size of the element changes. This is because of the box model.

The second is defined as follows, just adding 10 pixels of margin and 10 pixels of padding to the element:

```
div#d2 {
  width: 250px;
  height: 100px;
  background-color: #ff0000;
  border: 5px solid #000000;
  margin: 10px;
  padding: 10px;
}
```

The second , shown in Figure 3.3, is defined as the same height and width as the first one, but the overall height and width of the entire box surrounding the element itself is much larger when margins and padding are put in play.

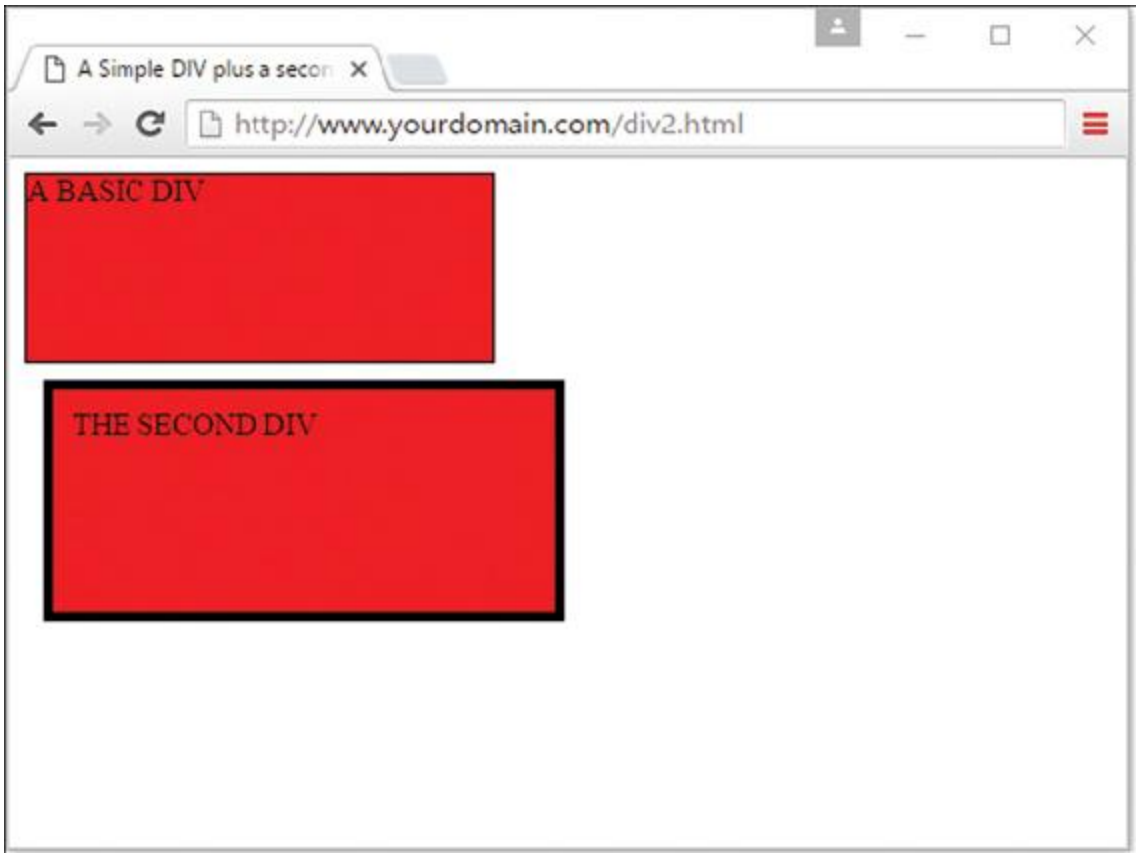

FIGURE 3.3
This is supposed to be another simple , but the box model affects the size of the second .

The total *width* of an element is the sum of the following:

Click here to view code image

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

The total *height* of an element is the sum of the following:

Click here to view code image

```
height + padding-top + padding-bottom + border-top + border-bottom
+
margin-top + margin-bottom
```

Therefore, the second `<div>` has an actual width of 300 (250 + 10 + 10 + 5 + 5 + 10 + 10) and an actual height of 150 (100 + 10 + 10 + 5 + 5 + 10 + 10).

By now, you can begin to see how the box model affects your design. Let's say that you have only 250 pixels of horizontal space, but you would like 10 pixels of margin, 10 pixels of padding, and 5 pixels of border on all sides. To accommodate what you would like with what you have room to display, you must specify the `width` of your `<div>` as only 200 pixels so that 200 + 10 + 10 + 5 + 5 + 10 + 10 adds up to that 250 pixels of available horizontal space.

The mathematics of the model are important as well. In dynamically driven sites or sites in which user interactions drive the client-side display (such as through JavaScript events), your server-side or client-side code could draw and redraw container elements on the fly. In other words, your code will produce the numbers, but you have to provide the boundaries.

Now that you've been introduced to the way of the box model, keep it in mind throughout the rest of the work you do in this book and in your web design. Among other things, it will affect element positioning and content flow, which are the two topics we tackle next.

The Whole Scoop on Positioning

Relative positioning is the default type of positioning HTML uses. You can think of relative positioning as being akin to laying out checkers on a checkerboard: The checkers are arranged from left to right, and when you get to the edge of the board, you move on to the next row. Elements that are styled with the `block` value for the `display` style property are automatically placed on a new row, whereas `inline` elements are placed on the same row immediately next to the element preceding them. As an example, `<p>` and `<div>` tags are considered block elements, whereas the `<span>` tag is considered an inline element.

The other type of positioning CSS supports is known as *absolute positioning* because it enables you to set the exact position of HTML content on a page. Although absolute positioning gives you the freedom to spell out exactly where an element is to appear, the position is still relative to any parent elements that appear on the page. In other words, absolute positioning enables you to specify the exact location of an element's rectangular area with respect to its parent's area, which is very different from relative positioning.

With the freedom of placing elements anywhere you want on a page, you can run into the problem of overlap, when an element takes up space another element is using. Nothing is stopping you from specifying the absolute locations of elements so that they overlap. In this case, CSS relies on the z-index of each element to determine which element is on the top and which is on the bottom. You'll learn more about the z-index of elements later in this lesson. For now, let's look at exactly how you control whether a style rule uses relative or absolute positioning.

The type of positioning (relative or absolute) a particular style rule uses is determined by the `position` property, which is capable of having one of the following four values:

- **`static`**—The default positioning according to the normal flow of the content
- **`relative`**—The element is positioned relative to its normal position, using offset properties discussed below

- **absolute**—The element is positioned relative to its nearest ancestor element, or according to the normal flow of the page if no ancestor is present
- **fixed**—The element is fixed relative to the viewport—this type of positioning is used for images that scroll along with the page (as an example)

After specifying the type of positioning, you provide the specific position using the following properties:

- **left**—The left position offset
- **right**—The right position offset
- **top**—The top position offset
- **bottom**—The bottom position offset

You might think that these position properties make sense only for absolute positioning, but they actually apply to relative and fixed positioning as well. For example, under relative positioning, the position of an element is specified as an offset relative to the original position of the element. So if you set the `left` property of an element to `25px`, the left side of the element shifts over 25 pixels from its original (relative) position. An absolute position, on the other hand, is specified relative to the ancestor element to which the style is applied. So if you set the `left` property of an element to `25px` under absolute positioning, the left side of the element appears 25 pixels to the right of the ancestor element's left edge. On the other hand, using the `right` property with the same value positions the element so that its *right* side is 25 pixels to the right of the ancestor's *right* edge.

Let's return to the color-blocks example to see how positioning works. In Listing 3.1, the four color blocks have relative positioning specified. As you can see in Figure 3.4, the blocks are positioned vertically.

LISTING 3.1 Showing Relative Positioning with Four Color Blocks

Click here to view code image

```
<!DOCTYPE html>

<html lang="en">
<head>
  <title>Positioning the Color Blocks</title>
   <style type="text/css">
   div {
     position: relative;
     width: 250px;
     height: 100px;
     border: 5px solid #000;
     color: black;
     font-weight: bold;
     text-align: center;
    }
    div#d1 {
     background-color: #ff0000;
    }
    div#d2 {
     background-color: #00ff00;
    }
    div#d3 {
     background-color: #0000ff;
    }
    div#d4 {
     background-color: #ffff00;
    }
   </style>
</head>
<body>
  <div id="d1">DIV #1</div>
  <div id="d2">DIV #2</div>
  <div id="d3">DIV #3</div>
  <div id="d4">DIV #4</div>
</body>
</html>
```

The style sheet entry for the `<div>` element itself sets the `position` style property for the `<div>` element to `relative`. Because the remaining style rules are inherited from the `<div>` style rule, they inherit its relative positioning. In fact, the only difference between the other style rules is that they have different background colors.

Notice in Figure 3.4 that the `<div>` elements are displayed one after the next, which is what you would expect with relative positioning. But to

make things more interesting, which is what we're here to do, you can change the positioning to absolute and explicitly specify the placement of the colors. In Listing 3.2, the style sheet entries are changed to use absolute positioning to arrange the color blocks.

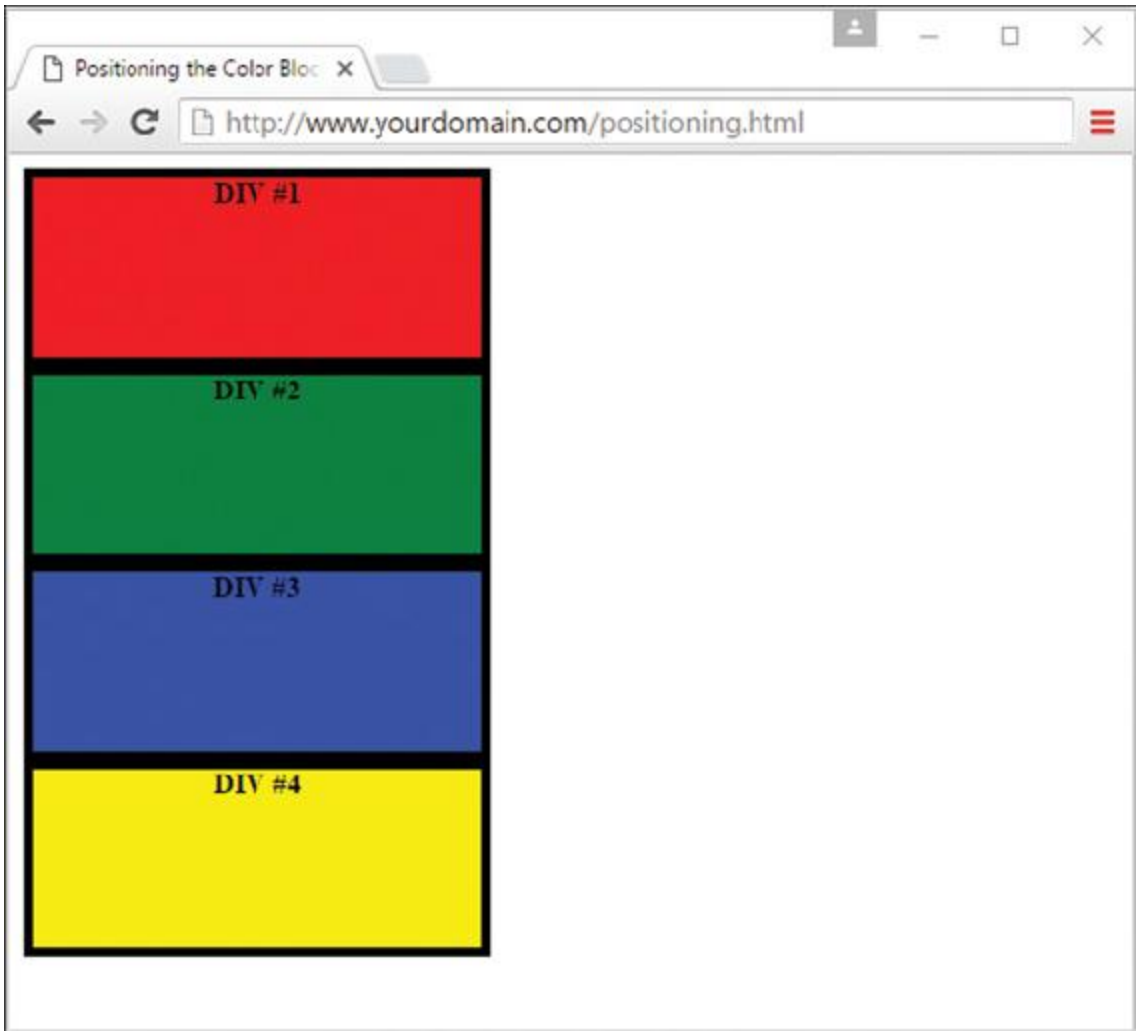

FIGURE 3.4
The color blocks are positioned vertically, with one on top of the other.

LISTING 3.2 Using Absolute Positioning of the Color Blocks

Click here to view code image

```
<!DOCTYPE html>

<html lang="en">
<head>
  <title>Positioning the Color Blocks</title>
```

```
    <style type="text/css">
    div {
     position: absolute;
     width: 250px;
     height: 100px;
     border: 5px solid #000;
     color: black;
     font-weight: bold;
     text-align: center;
    }
    div#d1 {
     background-color: #ff0000;
     left: 0px;
     top: 0px;
    }
    div#d2 {
     background-color: #00ff00;
     left: 75px;
     top: 25px;
    }
    div#d3 {
     background-color: #0000ff;
     left: 150px;
     top: 50px;
    }
    div#d4 {
     background-color: #ffff00;
     left: 225px;
     top: 75px;
    }
    </style>
</head>
<body>
     <div id="d1">DIV #1</div>
     <div id="d2">DIV #2</div>
     <div id="d3">DIV #3</div>
     <div id="d4">DIV #4</div>
</body>
</html>
```

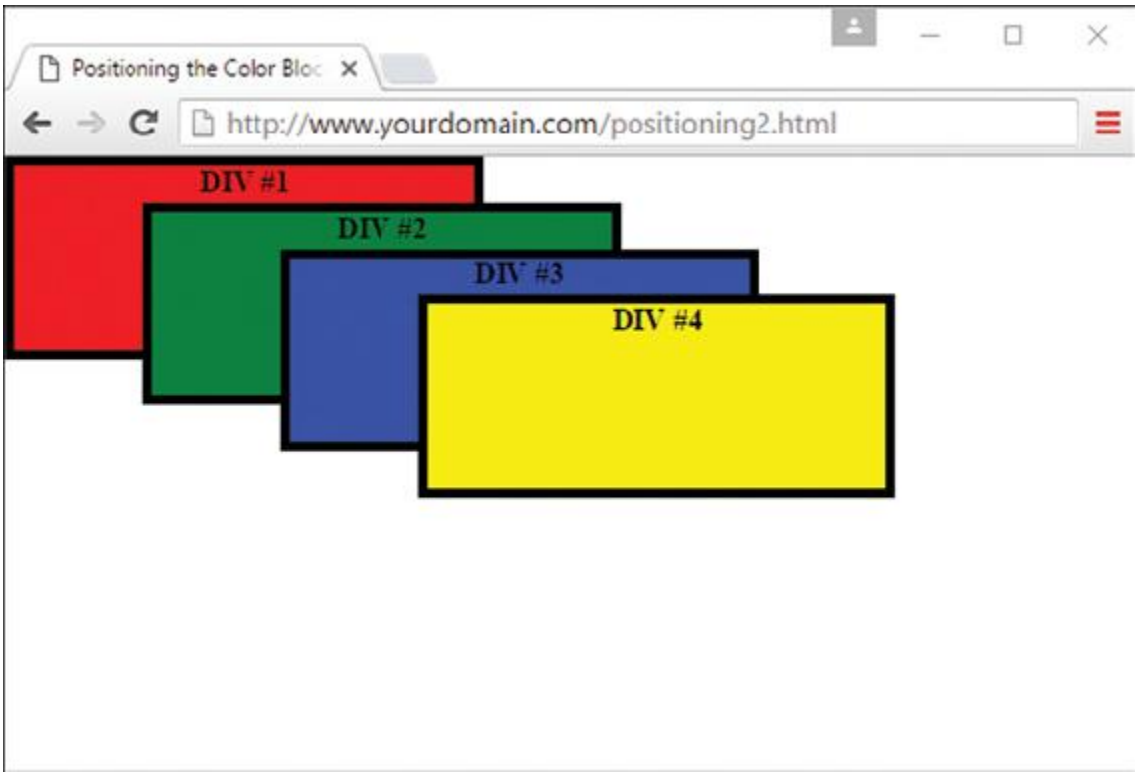

FIGURE 3.5
The color blocks are displayed using absolute positioning.

This style sheet sets the `position` property to `absolute`, which is necessary for the style sheet to use absolute positioning. Additionally, the `left` and `top` properties are set for each of the inherited `<div>` style rules. However, the position of each of these rules is set so that the elements are displayed overlapping each other, as Figure 3.5 shows.

Now we're talking layout! Figure 3.5 shows how absolute positioning enables you to place elements exactly where you want them. It also reveals how easy it is to arrange elements so that they overlap. You might be curious about how a web browser knows which elements to draw on top when they overlap. The next section covers how you can control stacking order.

Controlling the Way Things Stack Up

In certain situations, you want to carefully control the manner in which elements overlap each other on a web page. The `z-index` style property

enables you to set the order of elements with respect to how they stack on top of each other. The name *z-index* might sound a little strange, but it refers to the notion of a third dimension (Z) that points into the computer screen, in addition to the two dimensions that go across (X) and down (Y) the screen. Another way to think of the z-index is to consider the relative position of a single magazine within a stack of magazines. A magazine nearer the top of the stack has a higher z-index than a magazine lower in the stack. Similarly, an overlapped element with a higher value for its `z-index` is displayed on top of an element with a lower value for its `z-index`.

The `z-index` property is used to set a numeric value that indicates the relative `z-index` of a style rule. The number assigned to `z-index` has meaning only with respect to other style rules in a style sheet, which means that setting the `z-index` property for a single rule doesn't mean much. On the other hand, if you set `z-index` for several style rules that apply to overlapped elements, the elements with higher `z-index` values appear on top of elements with lower `z-index` values.

NOTE

Regardless of the `z-index` value you set for a style rule, an element displayed with the rule will always appear on top of its parent.

Listing 3.3 contains another version of the color-blocks style sheet and HTML that uses `z-index` settings to alter the natural overlap of elements.

LISTING 3.3 Using `z-index` to Alter the Display of Elements in the Color-Blocks Example

Click here to view code image

```
<!DOCTYPE html>

<html lang="en">
<head>
  <title>Positioning the Color Blocks</title>
   <style type="text/css">
   div {
```

```
     position: absolute;
     width: 250px;
     height: 100px;
     border: 5px solid #000;
     color: black;
     font-weight: bold;
     text-align: center;
    }
    div#d1 {
     background-color: #ff0000;
     left: 0px;
     top: 0px;
     z-index: 0;
    }
    div#d2 {
     background-color: #00ff00;
     left: 75px;
     top: 25px;
     z-index: 3;
    }
    div#d3 {
     background-color: #0000ff;
     left: 150px;
     top: 50px;
     z-index: 2;
    }
   div#d4 {
     background-color: #ffff00;
     left: 225px;
     top: 75px;
     z-index: 1;
    }
   </style>
</head>
<body>
  <div id="d1">DIV #1</div>
  <div id="d2">DIV #2</div>
  <div id="d3">DIV #3</div>
  <div id="d4">DIV #4</div>
</body>
</html>
```

The only change in this code from what you saw in Listing 3.2 is the addition of the `z-index` property in each of the numbered `div` style classes. Notice that the first numbered `div` has a `z-index` setting of `0`, which should make it the lowest element in terms of the z-index, whereas the second `div` has the highest `z-index`. Figure 3.6 shows the color-

blocks page as displayed with this style sheet, which clearly shows how the z-index affects the displayed content and makes it possible to carefully control the overlap of elements.

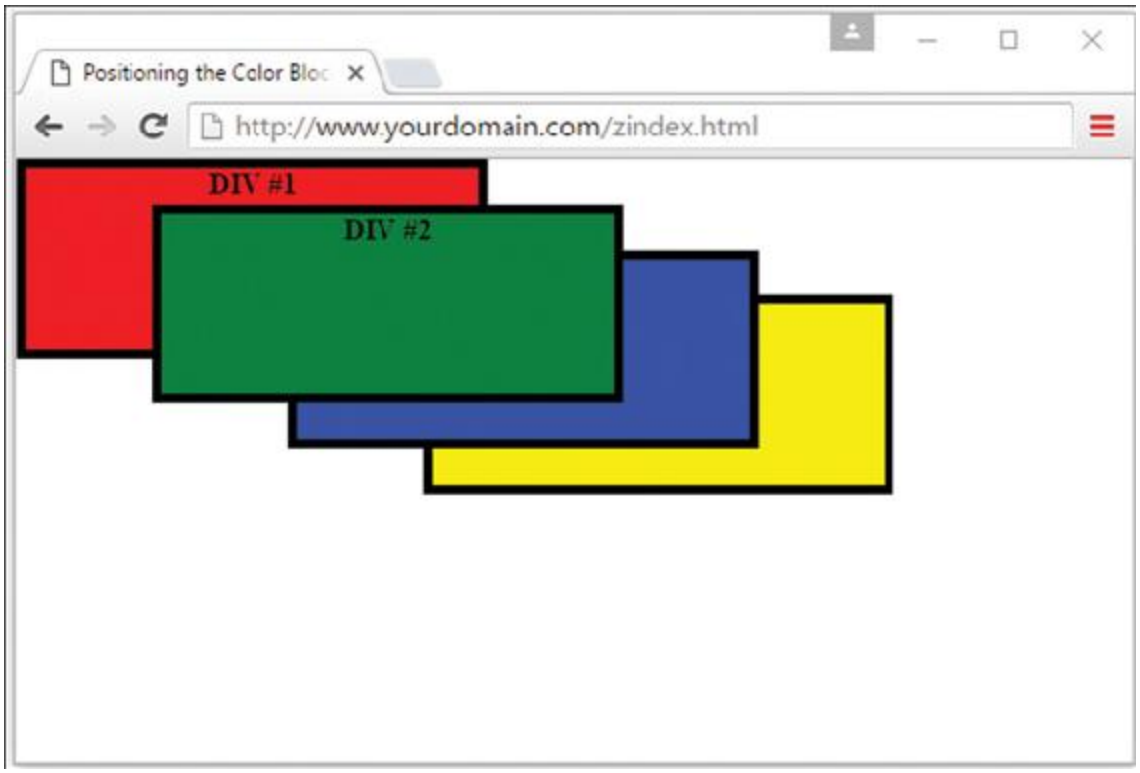

FIGURE 3.6
Using `z-index` to alter the display of the color blocks.

Although the examples show color blocks that are simple `<div>` elements, the `z-index` style property can affect any HTML content, including images.

Managing the Flow of Text

Now that you've seen some examples of placing elements relative to other elements or placing them absolutely, it's time to revisit the flow of content around elements. The conceptual *current line* is an invisible line used to place elements on a page. This line has to do with the flow of elements on a page; it comes into play as elements are arranged next to each other across and down the page. Part of the flow of elements is the flow of text on a

page. When you mix text with other elements (such as images), it's important to control how the text flows around those other elements.

Following are some style properties that give you control over text flow:

- **`float`**—Determines how text flows around an element
- **`clear`**—Stops the flow of text around an element
- **`overflow`**—Controls the overflow of text when an element is too small to contain all the text

The `float` property controls how text flows around an element. It can be set to either `left` or `right`. These values determine where to position an element with respect to flowing text. So setting an image's `float` property to `left` positions the image to the left of flowing text.

As you learned in the preceding chapter, you can prevent text from flowing next to an element by using the `clear` property, which you can set to `none`, `left`, `right`, or `both`. The default value for the `clear` property is `none`, indicating that text is to flow with no special considerations for the element. The `left` value causes text to stop flowing around an element until the left side of the page is free of the element. Likewise, the `right` value means that text is not to flow around the right side of the element. The `both` value indicates that text isn't to flow around either side of the element.

The `overflow` property handles overflow text, which is text that doesn't fit within its rectangular area; this can happen if you set the `width` and `height` of an element too small. The `overflow` property can be set to `visible`, `hidden`, or `scroll`. The `visible` setting automatically enlarges the element so that the overflow text fits within it; this is the default setting for the property. The `hidden` value leaves the element the same size, allowing the overflow text to remain hidden from view. Perhaps the most interesting value is `scroll`, which adds scrollbars to the element so that you can move around and see the text.

Understanding Fixed Layouts

A fixed layout, or fixed-width layout, is just that: a layout in which the body of the page is set to a specific width. That width is typically controlled by a master "wrapper" element that contains all the content. The `width` property of a wrapper element, such as a `<div>`, is set in the style sheet entry if the `<div>` was given an ID value such as `main` or `wrapper` (although the name is up to you).

When you're creating a fixed-width layout, the most important decision is determining the minimum screen resolution you want to accommodate. For many years, 800×600 was the "lowest common denominator" for web designers, resulting in a typical fixed width of approximately 760 pixels. However, the number of people using 800×600 screen resolution for non-mobile browsers is now less than 4%. Given that, many web designers consider 1,024×768 the current minimum screen resolution, so if they create fixed-width layouts, the fixed width typically is somewhere between 800 and 1,000 pixels wide.

CAUTION

Remember, the web browser window contains nonviewable areas, including the scrollbar. So if you are targeting a 1,024-pixel-wide screen resolution, you really can't use all 1,024 of those pixels.

A main reason for creating a fixed-width layout is so that you can have precise control over the appearance of the content area. However, if users visit your fixed-width site with smaller or much larger screen sizes or resolutions than the size or resolution you had in mind while you designed it, they will encounter scrollbars (if their size or resolution is smaller) or a large amount of empty space (if their size or resolution is greater). Finding fixed-width layouts is difficult among the most popular websites these days because site designers know they need to cater to the largest possible audience (and therefore make no assumptions about browser size). However, fixed-width layouts still have wide adoption, especially by site administrators using a content management system with a strict template.

The following figures show one such site, for San Jose State University (university websites commonly use a strict template and content management system, so this was an easy example to find); it has a wrapper element fixed at 960 pixels wide. In Figure 3.7, the browser window is a shade under 900 pixels wide. On the right side of the image, important content is cut off (and at the bottom of the figure, a horizontal scrollbar displays in the browser).

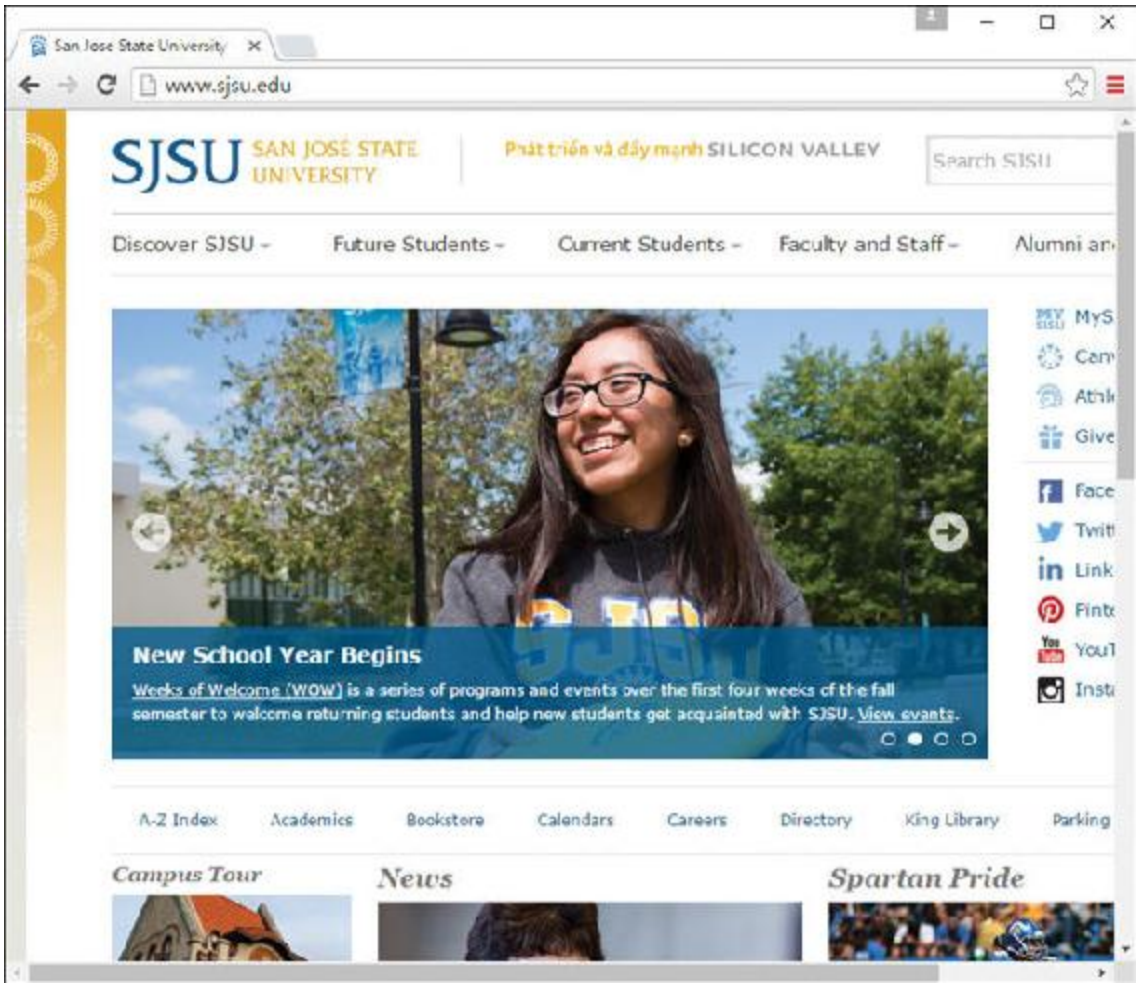

FIGURE 3.7
A fixed-width example with a smaller screen size.

However, Figure 3.8 shows how this site looks when the browser window is more than 1,300 pixels wide: You see a lot of empty space (or “real estate”) on both sides of the main body content, which some consider aesthetically displeasing.

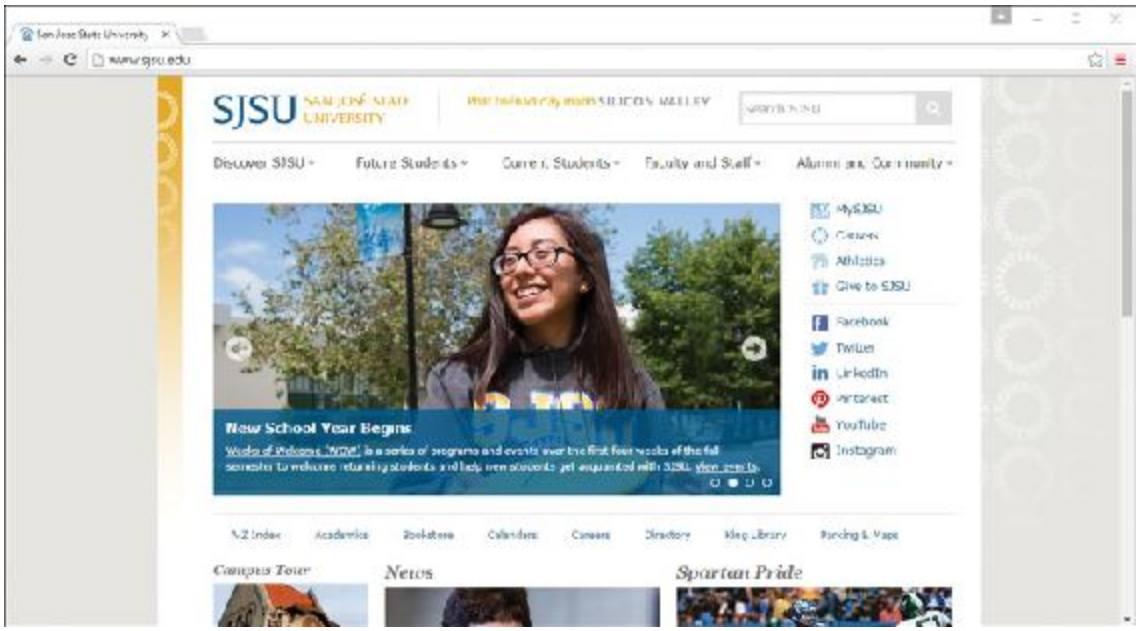

FIGURE 3.8
A fixed-width example with a larger screen size.

Besides the decision to create a fixed-width layout in the first place is the task of determining whether to place the fixed-width content flush left or center it. Placing the content flush left produces extra space on the right side only; centering the content area creates extra space on both sides. However, centering at least provides balance, whereas a flush-left design could end up looking like a small rectangle shoved in the corner of the browser, depending on the size and resolution of a user's monitor.

Understanding Fluid Layouts

A fluid layout—also called a *liquid* layout—is one in which the body of the page does not use a specified width in pixels, although it might be enclosed in a master "wrapper" element that uses a percentage width. The idea behind the fluid layout is that it can be perfectly usable and still retain the overall design aesthetic even if the user has a very small or very wide screen.

Figures 3.9, 3.10, and 3.11 show three examples of a fluid layout in action.

In Figure 3.9, the browser window is approximately 745 pixels wide. This example shows a reasonable minimum screen width before a horizontal scrollbar appears. In fact, the scrollbar does not appear until the browser is 735 pixels wide. On the other hand, Figure 3.10 shows a very small browser window (less than 600 pixels wide).

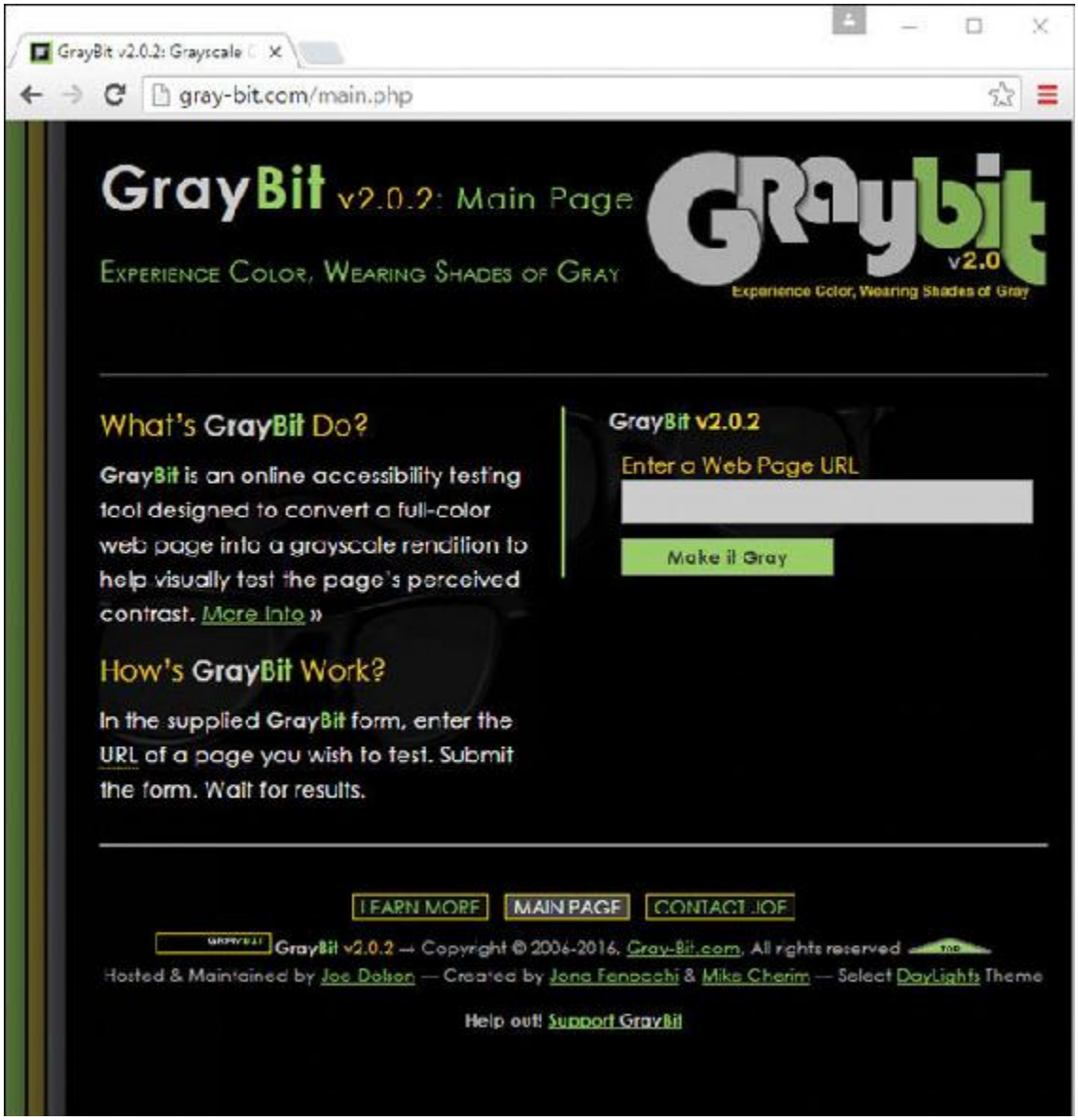

FIGURE 3.9
A fluid layout as viewed in a relatively small screen.

In Figure 3.10, you can see a horizontal scrollbar; in the header area of the page content, the logo graphic is beginning to take over the text and appear

on top of it. But the bulk of the page is still quite usable. The informational content on the left side of the page is still legible and is sharing the available space with the input form on the right side.

Figure 3.11 shows how this same page looks in a very wide screen. In Figure 3.11, the browser window is approximately 1,200 pixels wide. There is plenty of room for all the content on the page to spread out. This fluid layout is achieved because all the design elements have a percentage width specified (instead of a fixed width). Thus, the layout makes use of all the available browser real estate.

The fluid layout approach might seem like the best approach at first glance —after all, who wouldn't want to take advantage of all the screen real estate available? But there's a fine line between taking advantage of space and not allowing the content to breathe. Too much content is overwhelming; not enough content in an open space is underwhelming.

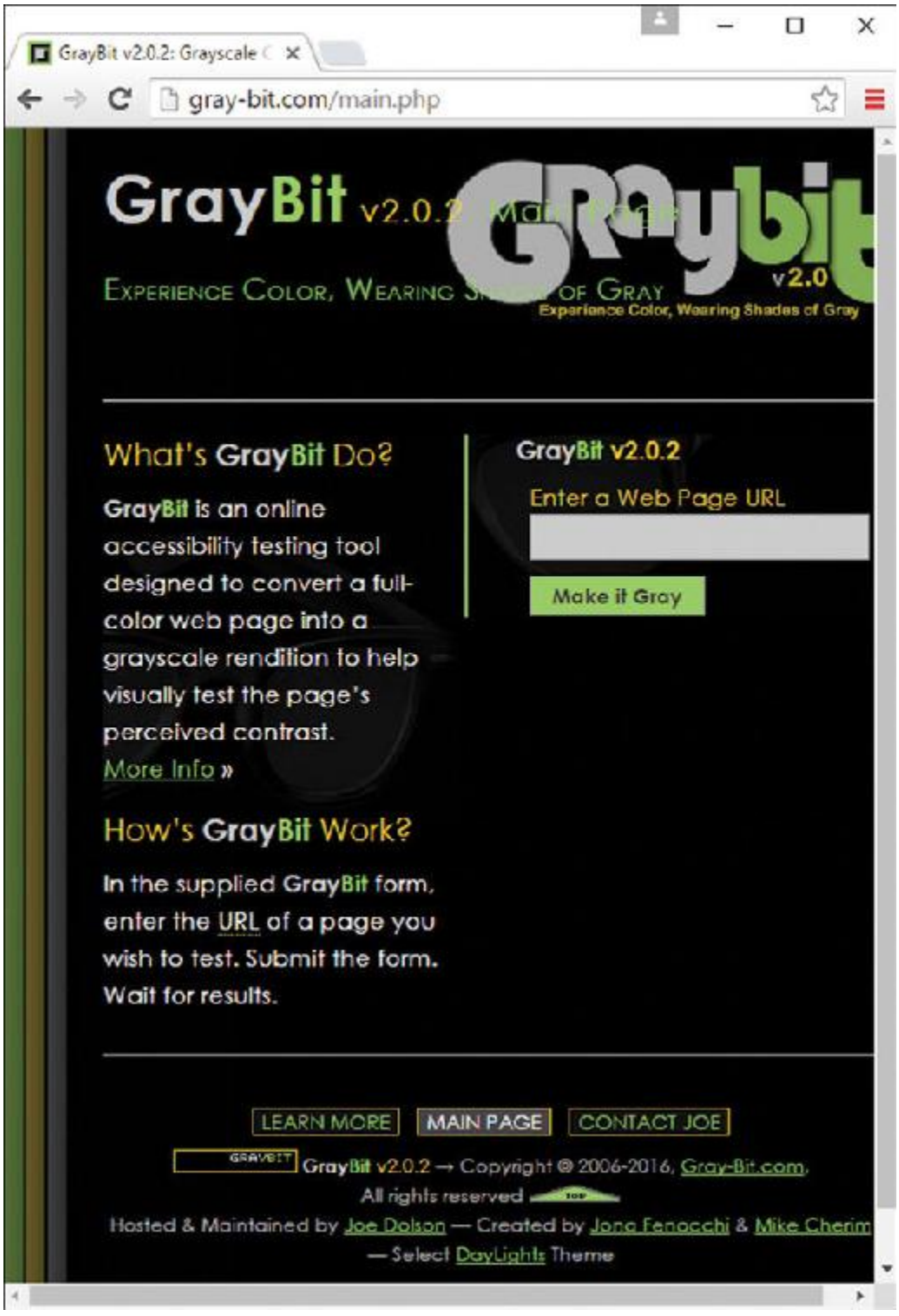

FIGURE 3.10
A fluid layout as viewed in a very small screen.

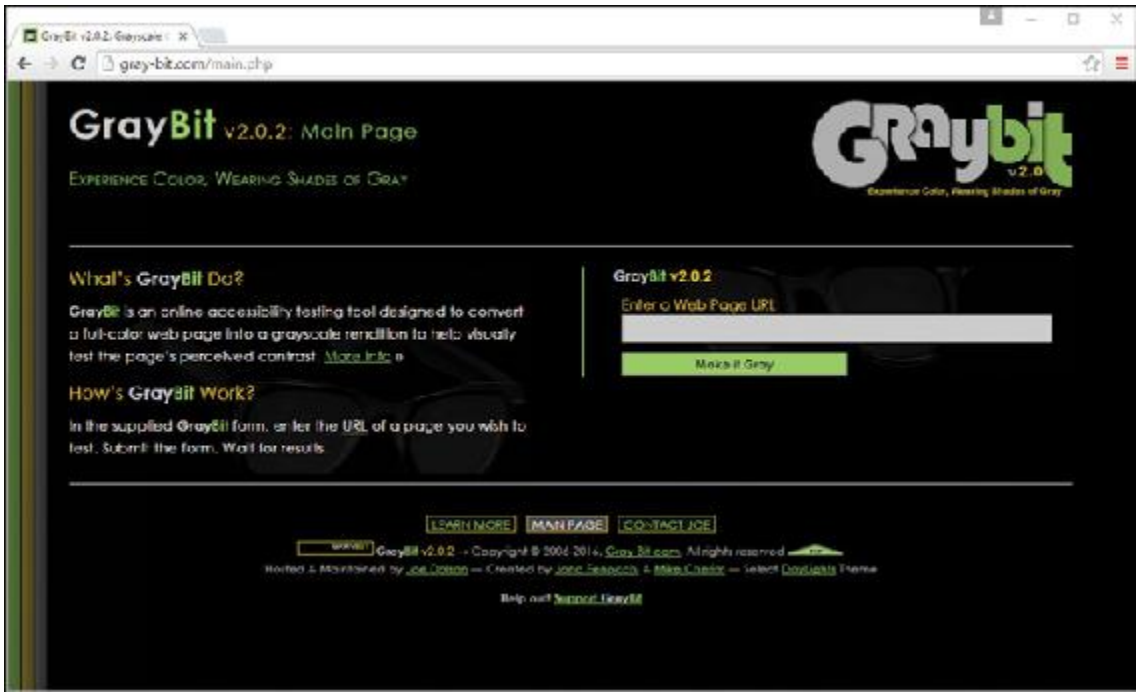

FIGURE 3.11
A fluid layout as viewed in a wide screen.

The pure fluid layout can be impressive, but it requires a significant amount of testing to ensure that it is usable in a wide range of browsers at varying screen resolutions. You might not have the time and effort to produce such a design; in that case, a reasonable compromise is the fixed/fluid hybrid layout, or a fully responsive design, as you'll learn about later in this chapter.

Creating a Fixed/Fluid Hybrid Layout

A fixed/fluid hybrid layout is one that contains elements of both types of layouts. For example, you could have a fluid layout that includes fixed-width content areas either within the body area or as anchor elements (such as a left-side column or as a top navigation strip). You can even create a fixed content area that acts like a frame, in which a content area remains fixed even as users scroll through the content.

Starting with a Basic Layout Structure

In this example, you'll learn to create a template that is fluid but with two fixed-width columns on either side of the main body area (which is a third column, if you think about it, only much wider than the others). The template also has a delineated header and footer area. Listing 3.4 shows the basic HTML structure for this layout.

LISTING 3.4 Basic Fixed/Fluid Hybrid Layout Structure

Click here to view code image

```
<!DOCTYPE html>

<html lang="en">
  <head>
     <title>Sample Layout</title>
     <link href="layout.css" rel="stylesheet" type="text/css">
  </head>

  <body>
    <header>HEADER</header>
    <div id="wrapper">
        <div id="content_area">CONTENT</div>
        <div id="left_side">LEFT SIDE</div>
        <div id="right_side">RIGHT SIDE</div>
    </div>
    <footer>FOOTER</footer>
  </body>
</html>
```

First, note that the style sheet for this layout is linked to with the `<link>` tag instead of included in the template. Because a template is used for more than one page, you want to be able to control the display elements of the template in the most organized way possible. This means you need to change the definitions of those elements in only one place—the style sheet.

Next, notice that the basic HTML is just that: extremely basic. Truth be told, this basic HTML structure can be used for a fixed layout, a fluid layout, or the fixed/fluid hybrid you see here because all the actual styling that makes a layout fixed, fluid, or hybrid happens in the style sheet.

With the HTML structure in Listing 3.4, you actually have an identification of the content areas you want to include in your site. This planning is crucial to any development; you have to know what you want to include

before you even think about the type of layout you are going to use, let alone the specific styles that will be applied to that layout.

NOTE

I am using elements with named identifiers in this example instead of the semantic elements such as `<section>` or `<nav>` because I'm illustrating the point in the simplest way possible without being prescriptive to the content itself. However, if you know that the `<div>` on the left side is going to hold navigation, you should use the `<nav>` tag instead of a `<div>` element with an `id` something like `left_side`—but this type of naming can become problematic as depending on repositioning that content might not end up displaying on the left side of anything, so best to name based on purpose rather than appearance.

At this stage, the `layout.css` file includes only this entry:

```
body {
   margin: 0;
   padding: 0;
}
```

Using a `0` value for `margin` and `padding` allows the entire page to be usable for element placement.

If you look at the HTML in Listing 3.4 and say to yourself, "But those `<div>` elements will just stack on top of each other without any styles," you are correct. As shown in Figure 3.12, there is no layout to speak of.

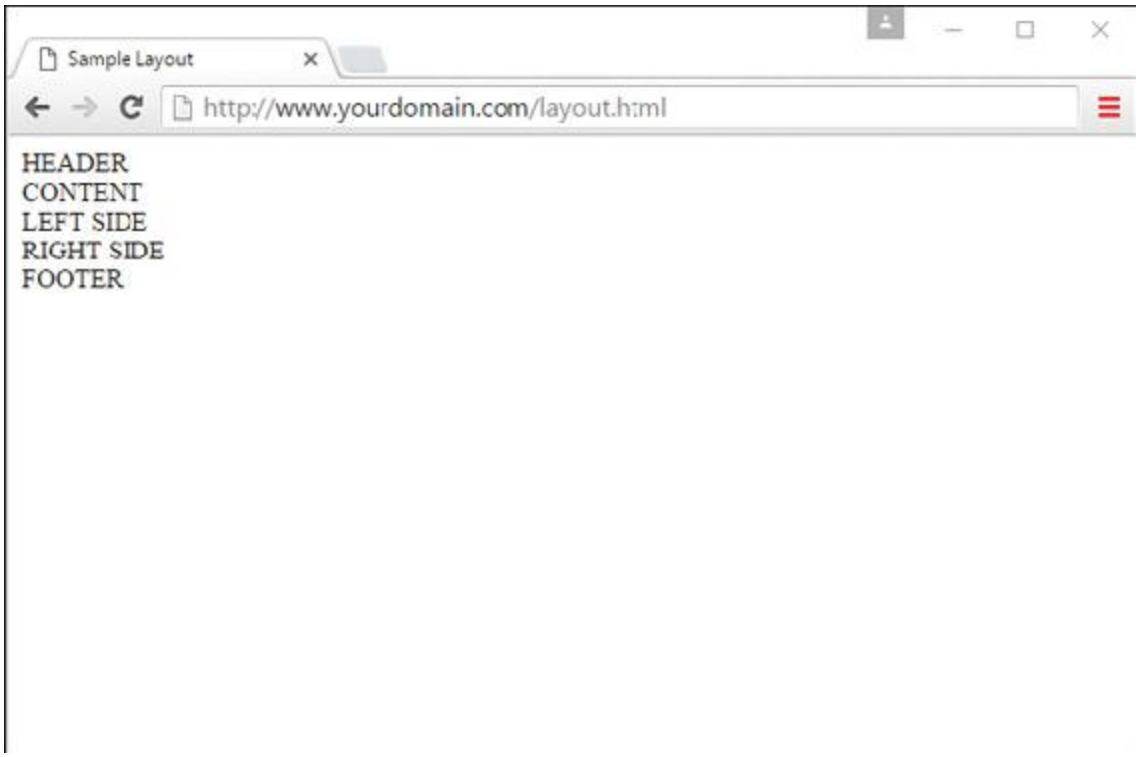

FIGURE 3.12
A basic HTML template with no styles applied to the container elements.

Defining Two Columns in a Fixed/Fluid Hybrid Layout

We can start with the easy things to get some styles and actual layout in there. Because this layout is supposed to be fluid, we know that whatever we put in the header and footer areas will extend the width of the browser window, regardless of how narrow or wide the window might be.

Adding the following code to the style sheet gives the header and footer area each a width of 100% as well as the same background color and text color:

```
header, footer {
  float: left;
  width: 100%;
  background-color: #7152f4;
  color: #ffffff;
}
```

Now things get a little trickier. We have to define the two fixed columns on either side of the page, plus the column in the middle. In the HTML we're using here, note that a `<div>` element, called `wrapper`, surrounds both. This element is defined in the style sheet as follows:

```
#wrapper {
  float: left;
  padding-left: 200px;
  padding-right: 125px;
}
```

The two padding definitions essentially reserve space for the two fixed-width columns on the left and right of the page. The column on the left will be 200 pixels wide, the column on the right will be 125 pixels wide, and each will have a different background color. But we also have to position the items relative to where they would be placed if the HTML remained unstyled (see Figure 3.12). This means adding `position: relative` to the style sheet entries for each of these columns. Additionally, we indicate that the `<div>` elements should float to the left.

But in the case of the `<div>` element `left_side`, we also indicate that we want the rightmost margin edge to be 200 pixels in from the edge (this is in addition to the column being defined as 200 pixels wide). We also want the margin on the left side to be a full negative margin; this will pull it into place (as you will soon see). The `<div>` element `right_side` does not include a value for `right`, but it does include a negative margin on the right side:

```
#left_side {
  position: relative;
  float: left;
  width: 200px;
  background-color: #52f471;
  right: 200px;
  margin-left: -100%;
}

#right_side {
  position: relative;
  float: left;
  width: 125px;
  background-color: #f452d5;
```

```
  margin-right: -125px;
}
```

At this point, let's also define the content area so that it has a white background, takes up 100% of the available area, and floats to the left relative to its position:

```
#content_area {
  position: relative;
  float: left;
  background-color: #ffffff;
  width: 100%;
}
```

At this point, the basic layout should look something like Figure 3.13, with the areas clearly delineated.

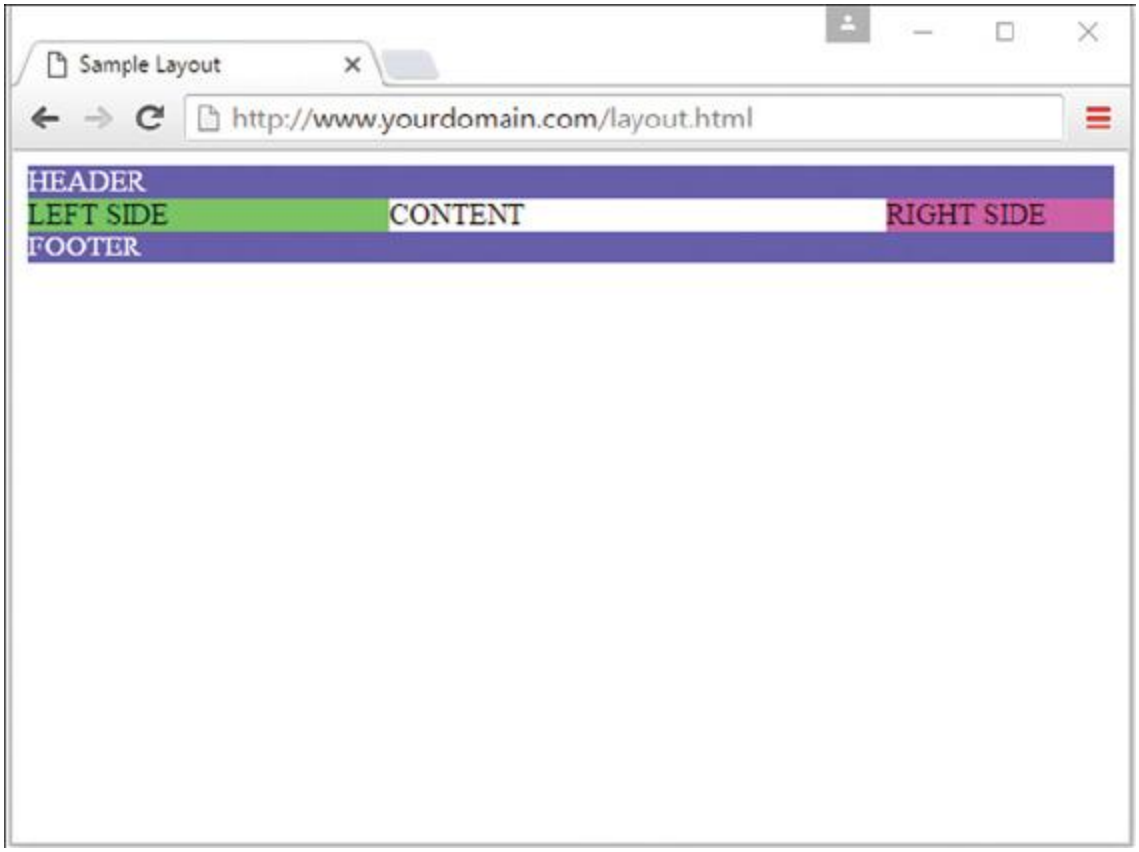

FIGURE 3.13
A basic HTML template after some styles have been put in place.

However, there's a problem with this template if the window is resized below a certain width. Because the left column is 200 pixels wide and the right column is 125 pixels wide, and we want at least *some* text in the content area, you can imagine that this page will break if the window is only 350 to 400 pixels wide. We address this issue in the next section.

Setting the Minimum Width of a Layout

Although users won't likely visit your site with a desktop browser that displays less than 400 pixels wide, the example serves its purpose within the confines of this lesson. You can extrapolate and apply this information broadly: Even in fixed/fluid hybrid sites, at some point, your layout will break down unless you do something about it.

One of those "somethings" is to use the `min-width` CSS property. The `min-width` property sets the minimum width of an element, not including padding, borders, and margins. Figure 3.14 shows what happens when `min-width` is applied to the `<body>` element.

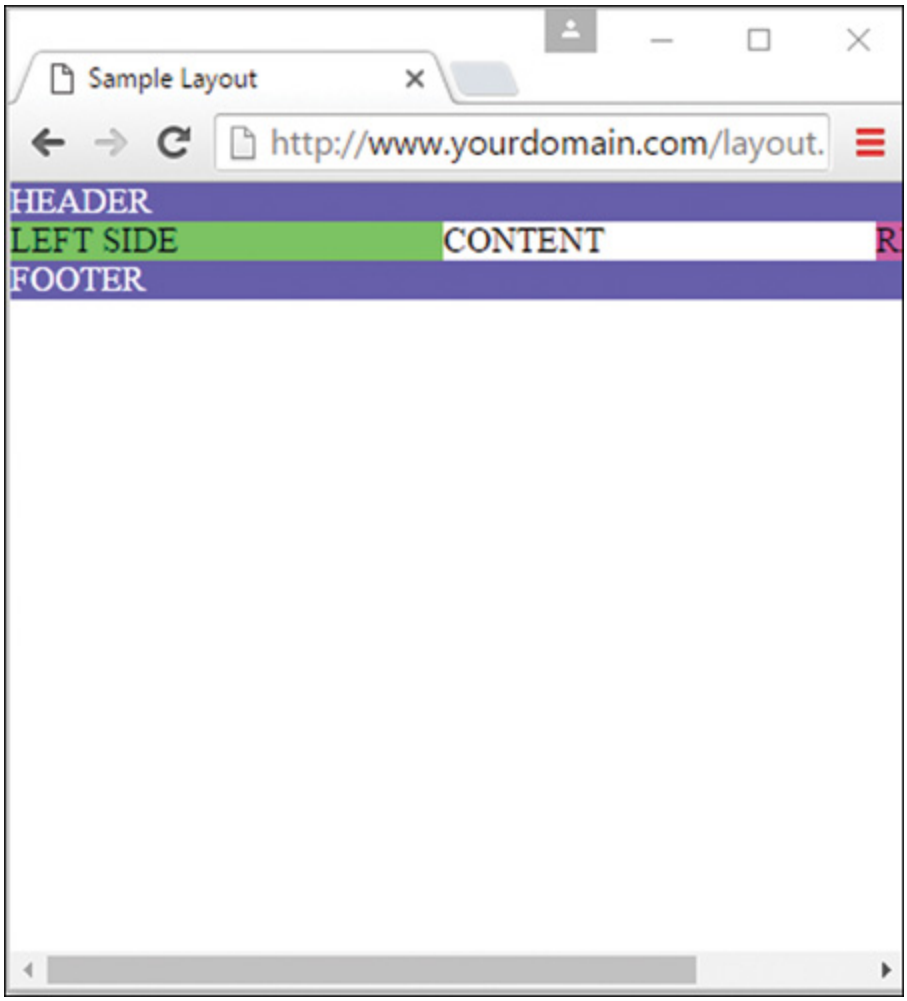

FIGURE 3.14
A basic HTML template resized to around 400 pixels, with a minimum width applied.

Figure 3.14 shows a small portion of the right column after the screen has been scrolled to the right, but the point is that the layout does not break apart when resized below a minimum width. In this case, the minimum width is 525 pixels:

```
body {
  margin: 0;
  padding: 0;
  min-width: 525px;
}
```

The horizontal scrollbar appears in this example because the browser window itself is less than 500 pixels wide. The scrollbar disappears when

the window is slightly larger than 525 pixels wide.

Handling Column Height in a Fixed/Fluid Hybrid Layout

This example is all well and good except for one problem: It has no content. When content is added to the various elements, more problems arise. As Figure 3.15 shows, the columns become as tall as necessary for the content they contain.

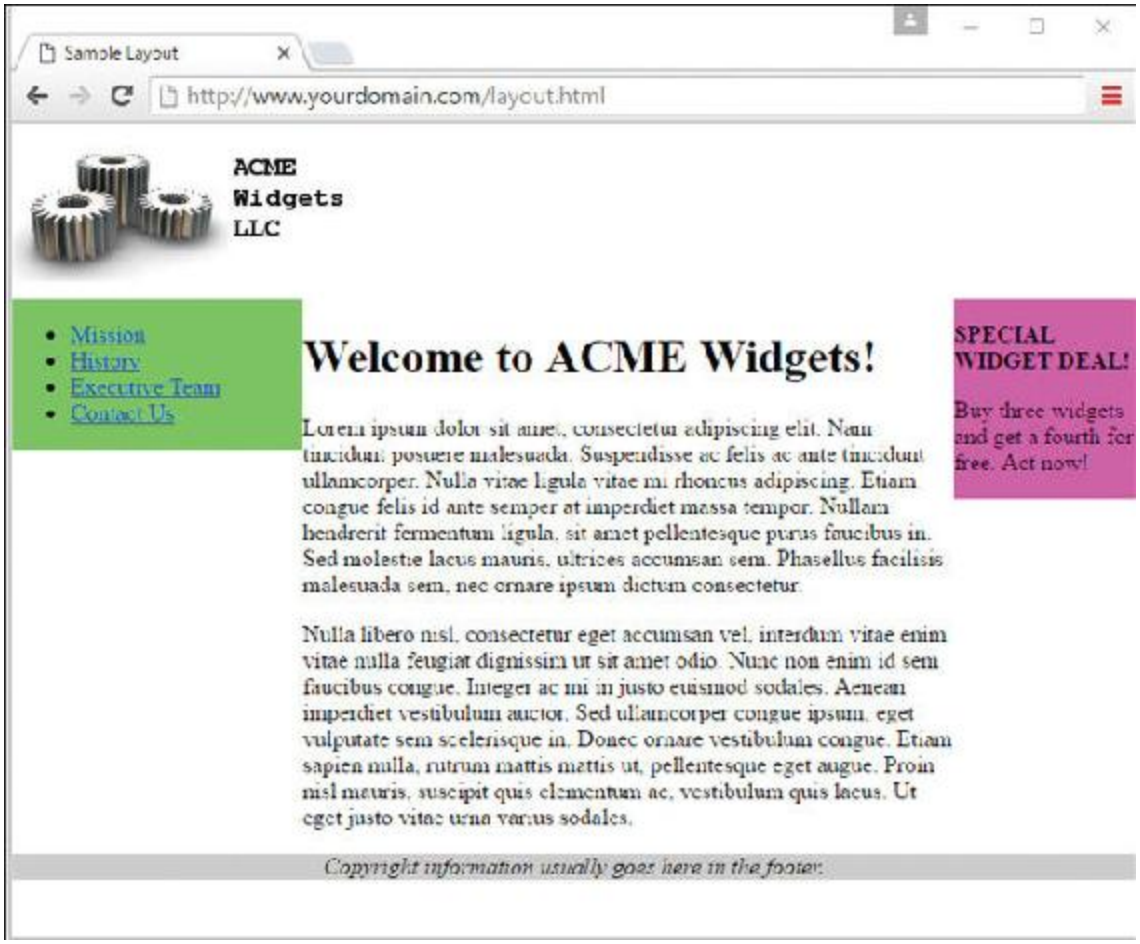

FIGURE 3.15
Columns are only as tall as their content.

NOTE

Because we have moved beyond the basic layout example, I also took the liberty to remove the background and text color properties for the header and footer, which is why the example no longer shows white text on a very dark background. Additionally, I've centered the text in the `<footer>` element, which now has a light gray background.

Because you cannot count on a user's browser being a specific height, or the content always being the same length, you might think this poses a problem with the fixed/fluid hybrid layout. Not so. If you think a little outside the box, you can apply a few more styles to bring all the pieces together.

First, add the following declarations in the style sheet entries for the `left_side`, `right_side`, and `content_area` IDs:

```
margin-bottom: -2000px;
padding-bottom: 2000px;
```

These declarations add a ridiculous amount of padding and assign a too-large margin to the bottom of all three elements. You must also add `position:relative` to the footer element definitions in the style sheet so that the footer is visible despite this padding.

At this point, the page looks as shown in Figure 3.16—still not what we want, but closer.

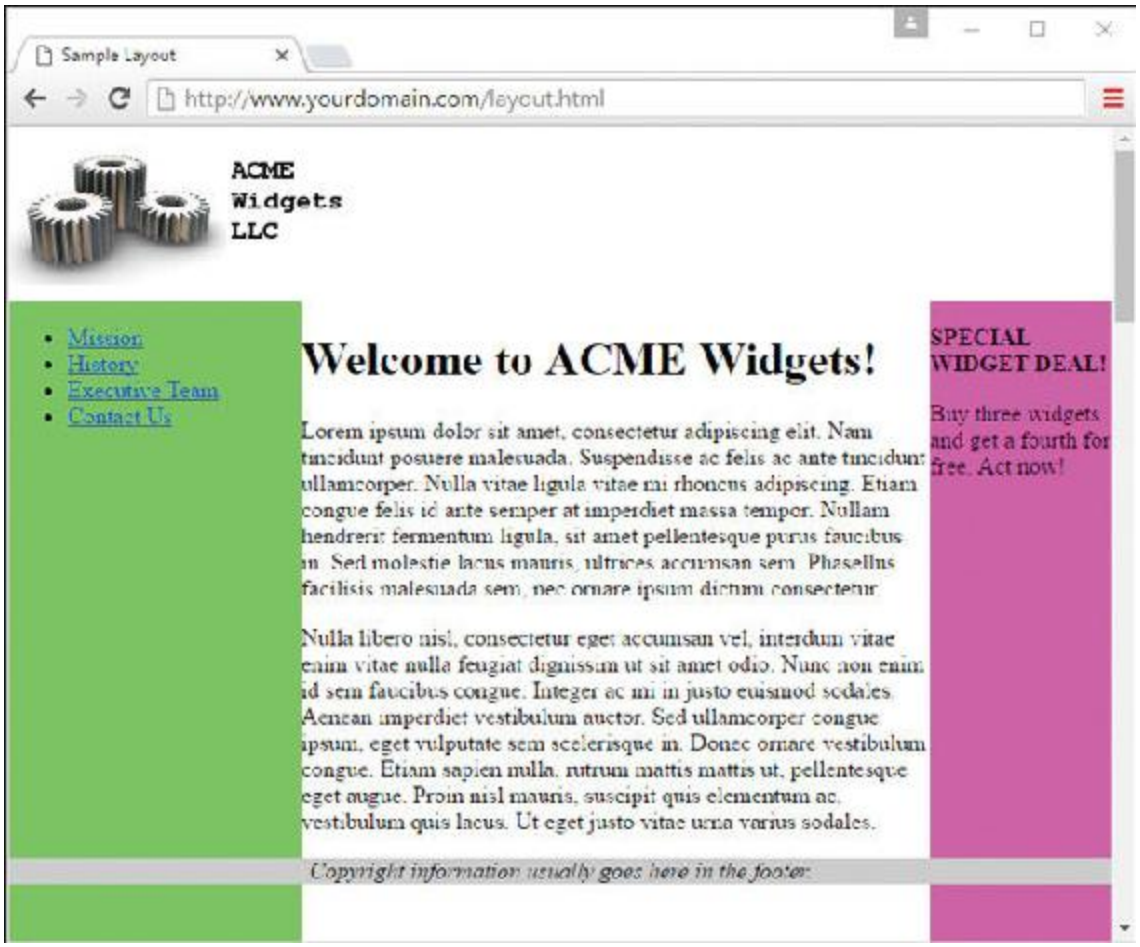

FIGURE 3.16
Color fields are now visible, despite the amount of content in the columns.

To clip off all that extra color, add the following to the style sheet for the `wrapper` ID:

```
overflow: hidden;
```

Figure 3.17 shows the final result: a fixed-width/fluid hybrid layout with the necessary column spacing. I also took the liberty of styling the navigational links and adjusting the margin around the welcome message; you can see the complete style sheet in Listing 3.6.

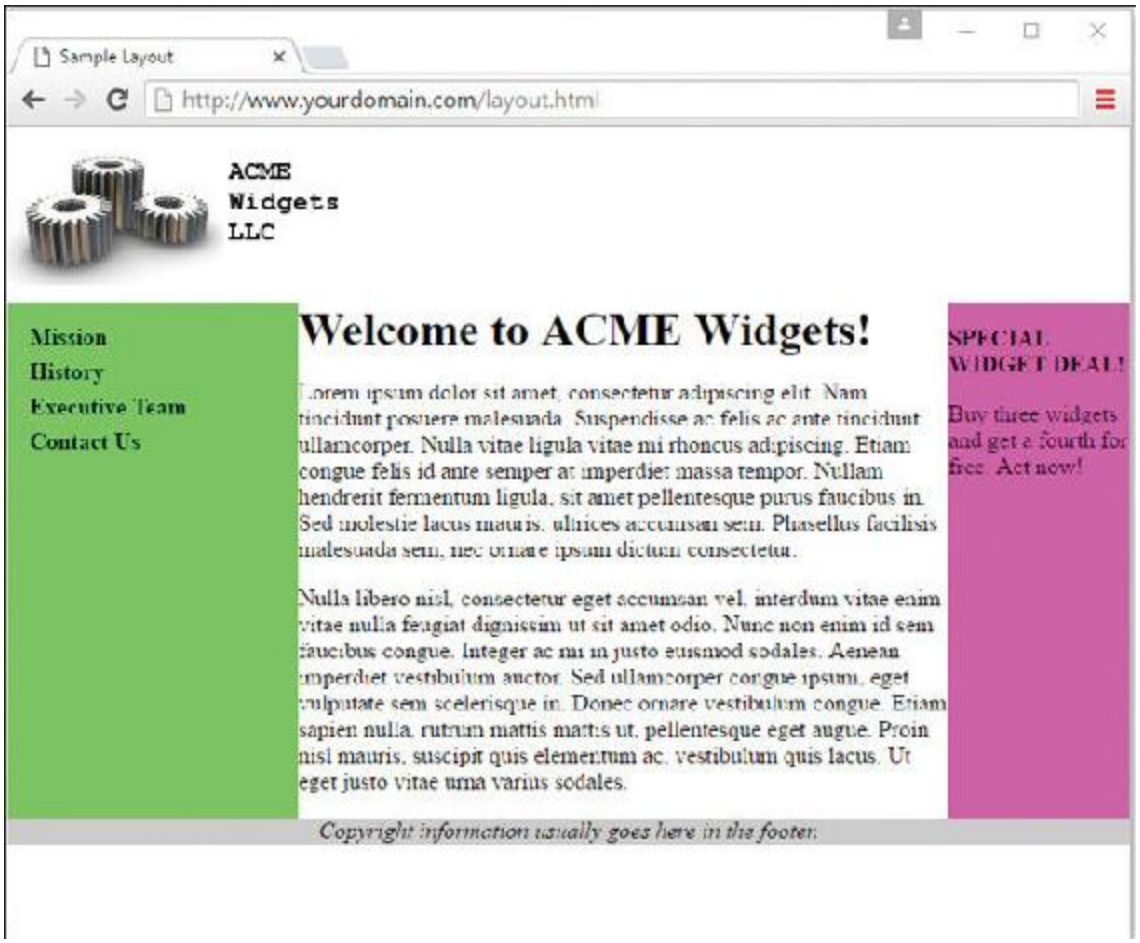

FIGURE 3.17
Congratulations! It's a fixed-width/fluid hybrid layout (although you'll want to do something about those colors!).

The full HTML code appears in Listing 3.5, and Listing 3.6 shows the final style sheet.

LISTING 3.5 Basic Fixed/Fluid Hybrid Layout Structure (with Content)

Click here to view code image

```
<!DOCTYPE html>

<html lang="en">
  <head>
      <title>Sample Layout</title>
      <link href="layout.css" rel="stylesheet" type="text/css">
  </head>
```

```
  <body>
    <header><img src="acmewidgets.jpg" alt="ACME Widgets
       LLC"/></header>
    <div id="wrapper">
       <div id="content_area">
        <h1>Welcome to ACME Widgets!</h1>
        <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit.
        Nam tincidunt posuere malesuada. Suspendisse ac felis ac
ante
        tincidunt ullamcorper. Nulla vitae ligula vitae mi rhoncus
        adipiscing. Etiam congue felis id ante semper at imperdiet
        massa tempor. Nullam hendrerit fermentum ligula, sit amet
        pellentesque purus faucibus in. Sed molestie lacus mauris,
        ultrices accumsan sem. Phasellus facilisis malesuada sem,
nec
        ornare ipsum dictum consectetur.</p>
        <p>Nulla libero nisl, consectetur eget accumsan vel,
interdum
        vitae enim vitae nulla feugiat dignissim ut sit amet odio.
        Nunc non enim id sem faucibus congue. Integer ac mi in
justo
        euismod sodales. Aenean imperdiet vestibulum auctor. Sed
        ullamcorper congue ipsum, eget vulputate sem scelerisque
in.
        Donec ornare vestibulum congue. Etiam sapien nulla, rutrum
        mattis mattis ut, pellentesque eget augue. Proin nisl
mauris,
        suscipit quis elementum ac, vestibulum quis lacus. Ut eget
        justo vitae urna varius sodales. </p>
       </div>
       <div id="left_side">
         <ul>
         <li><a href="#">Mission</a></li>
         <li><a href="#">History</a></li>
         <li><a href="#">Executive Team</a></li>
         <li><a href="#">Contact Us</a></li>
       </ul>
       </div>
       <div id="right_side">
         <p><strong>SPECIAL WIDGET DEAL!</strong></p>
         <p>Buy three widgets and get a fourth for free. Act now!
</p>
       </div>
    </div>
    <footer>Copyright information usually goes here in the
    footer.</footer>
  </body>
</html>
```

LISTING 3.6 Full Style Sheet for Fixed/Fluid Hybrid Layout

Click here to view code image

```
body {
  margin: 0;
  padding: 0;
  min-width: 525px;
}

header {
  float: left;
  width: 100%;
}

footer {
  position: relative;
  float: left;
  width: 100%;
  background-color: #cccccc;
  text-align: center;
  font-style: italic;
}

#wrapper {
  float: left;
  padding-left: 200px;
  padding-right: 125px;
  overflow: hidden;
}

#left_side {
  position: relative;
  float: left;
  width: 200px;
  background-color: #52f471;
  right: 200px;
  margin-left: -100%;
  margin-bottom: -2000px;
  padding-bottom: 2000px;
}

#right_side {
  position: relative;
  float: left;
  width: 125px;
  background-color: #f452d5;
  margin-right: -125px;
  margin-bottom: -2000px;
  padding-bottom: 2000px;
}
```

```
#content_area {
  position: relative;
  float: left;
  background-color: #ffffff;
  width: 100%;
  margin-bottom: -2000px;
  padding-bottom: 2000px;
}

h1 {
  margin: 0;
}

#left_side ul {
  list-style: none;
  margin: 12px 0px 0px 12px;
  padding: 0px;
}

#left_side li a:link, #nav li a:visited {
  font-size: 12pt;
  font-weight: bold;
  padding: 3px 0px 3px 3px;
  color: #000000;
  text-decoration: none;
  display: block;
}

#left_side li a:hover, #nav li a:active {
  font-size: 12pt;
  font-weight: bold;
  padding: 3px 0px 3px 3px;
  color: #ffffff;
  text-decoration: none;
  display: block;
}
```

Considering a Responsive Web Design

In 2010, web designer Ethan Marcotte coined the term *responsive web design* to refer to a web design approach that builds on the basics of fluid design you just learned a bit about. The goal of a responsive web design is that content is easy to view, read, and navigate, regardless of the device type and size on which you are viewing it. In other words, a designer who sets out to create a responsive website is doing to so to ensure that the site is

similarly enjoyable to and usable by audience members viewing on a large desktop display, a small smartphone, or a medium-size tablet.

The underlying structure of a responsive design is based on fluid (liquid) grid layouts, much as you learned about earlier in this chapter, but with a few modifications and additions. First, those grid layouts should always be in relative units rather than absolute ones. In other words, designers should use percentages rather than pixels to define container elements.

Second—and this is something we have not discussed previously—all images should be flexible. By this, I mean that instead of using a specific height and width for each image, we use relative percentages so that the images always display within the (relatively sized) element that contains them.

Finally, until you get a handle on intricate creations of style sheets for multiple uses, spend some time developing specific style sheets for each media type, and use media queries to employ these different rules based on the type. As you advance in your work and understanding of responsive design—well beyond the scope of this book—you will learn to progressively enhance your layouts in more meaningful ways.

Remember, you can specify a link to a style sheet like the following:

Click here to view code image

```
<link rel="stylesheet" type="text/css"
   media="screen and (max-device-width: 480px)"
   href="wee.css">
```

In this example, the `media` attribute contains a type and a query: The type is screen and the query portion is `(max-device-width: 480px)`. This means that if the device attempting to render the display is one with a screen and the horizontal resolution (device width) is less than 480 pixels wide—as with a smartphone—then load the style sheet called `wee.css` and render the display using the rules found within it.

Of course, a few short paragraphs in this book cannot do justice to the entirety of responsive web design. I highly recommend reading Marcotte's book *Responsive Web Design*

(http://www.abookapart.com/products/responsive-web-design) after you have firmly grounded yourself in the basics of HTML5 and CSS3 that are discussed and used throughout this book. Additionally, several of the HTML and CSS frameworks discussed in Chapter 22, "Managing Web Applications," take advantage of principles of responsive design, and that makes a great starting point for building up a responsive site and tinkering with the fluid grid, image resizing, and media queries that make it so.

Summary

This chapter began with an important discussion about the CSS box model and how to calculate the width and height of elements when taking margins, padding, and borders into consideration. The lesson continued by tackling absolute positioning of elements, and you learned about positioning using `z-index`. You then learned about a few nifty style properties that enable you to control the flow of text on a page.

Next, you saw some practical examples of the three main types of layouts: fixed, fluid, and a fixed/fluid hybrid. In the third section of the lesson, you saw an extended example that walked you through the process of creating a fixed/fluid hybrid layout in which the HTML and CSS all validate properly. Remember, the most important part of creating a layout is figuring out the sections of content you think you might need to account for in the design.

Finally, you were introduced to the concept of responsive web design, which itself is a book-length topic. Given the brief information you learned here, such as using a fluid grid layout, responsive images, and media queries, you have some basic concepts to begin testing on your own.

Q&A

Q. How would I determine when to use relative positioning and when to use absolute positioning?

A. Although there are no set guidelines regarding the usage of relative versus absolute positioning, the general idea is that absolute

positioning is required only when you want to exert a finer degree of control over how content is positioned. This has to do with the fact that absolute positioning enables you to position content down to the exact pixel, whereas relative positioning is much less predictable in terms of how it positions content. This isn't to say that relative positioning can't do a good job of positioning elements on a page; it just means that absolute positioning is more exact. Of course, this also makes absolute positioning potentially more susceptible to changes in screen size, which you can't really control.

Q. If I don't specify the z-index of two elements that overlap each other, how do I know which element will appear on top?

A. If the `z-index` property isn't set for overlapping elements, the element that appears later in the web page will appear on top. The easy way to remember this is to think of a web browser drawing each element on a page as it reads it from the HTML document; elements read later in the document are drawn on top of those that were read earlier.

Q. I've heard about something called an *elastic layout*. How does that differ from the fluid layout?

A. An *elastic layout* is a layout whose content areas resize when the user resizes the text. Elastic layouts use ems, which are inherently proportional to text and font size. An em is a typographical unit of measurement equal to the point size of the current font. When ems are used in an elastic layout, if a user forces the text size to increase or decrease in size using Ctrl and the mouse scroll wheel, the areas containing the text increase or decrease proportionally.

Q. You've spent a lot of time talking about fluid layouts and hybrid layouts—are they better than a purely fixed layout?

A. *Better* is a subjective term; in this book, the concern is with standards-compliant code. Most designers will tell you that fluid layouts take longer to create (and perfect), but the usability enhancements are worth it, especially when it leads to a responsive design. When might the

time not be worth it? If your client does not have an opinion and is paying you a flat rate instead of an hourly rate. In that case, you are working only to showcase your own skills (that might be worth it to you, however).

Workshop

The Workshop contains quiz questions and exercises to help you solidify your understanding of the material covered. Try to answer all questions before looking at the "Answers" section that follows.

Quiz

1. What's the difference between relative positioning and absolute positioning?
2. Which CSS style property controls the manner in which elements overlap each other?
3. What does `min-width` do?

Answers

1. In relative positioning, content is displayed according to the flow of a page, with each element physically appearing after the element preceding it in the HTML code. Absolute positioning, on the other hand, enables you to set the exact position of content on a page.
2. The `z-index` style property controls the manner in which elements overlap each other.
3. The `min-width` property sets the minimum width of an element, not including padding, borders, and margins.

Exercises

- Practice working with the intricacies of the CSS box model by creating a series of elements with different margins, padding, and borders, and see how these properties affect their height and width.
- Figure 3.17 shows the finished fixed/fluid hybrid layout, but notice a few areas for improvement: There isn't any space around the text in the right-side column, there aren't any margins between the body text and either column, the footer strip is a little sparse, and so on. Take some time to fix these design elements.
- Using the code you fixed in the preceding exercise, try to make it responsive, using only the brief information you learned in this chapter. Just converting container elements to relative sizes should go a long way toward making the template viewable on your smartphone or other small device, but a media query and alternative style sheet certainly wouldn't hurt, either.