9

Creating Flexible Images

No matter how perfectly you build your liquid or elastic layout, it's not going to work if you don't make the content within it flexible too. Text is easy—it wraps by default. Images are where it gets tricky. Luckily, as you saw in Chapter 2, there are lots of creative ways to make your images—content images as well as decorative graphic elements—flexible to either the viewport or the text size. In this chapter, you'll learn the CSS behind those flexible image examples.

Dynamically Changing Images' Screen Area

■ **NOTE:** Each of the completed example files is available for download from this book's companion web site at www.flexiblewebbook.com. Download the file ch9_examples.zip to get the complete set. I'll let you know which file goes with which technique as we go along.

Since the area available for an image to display within a flexible layout changes on the fly, your images may need to as well. While fixed-width images *can* work within flexible layouts—as long as they're not too large, or you have matching minimum widths in place—there are lots of ways you can dynamically change the screen area that an image takes up.

Foreground Images that Scale with the Layout

One way to dynamically alter the footprint of an image is to make it literally scale. You saw an example of an image that scales with the text in Figure 2.18, and an example of an image scaling with the changing dimensions of its parent element in Figure 2.19. Both are elegant effects that are deceptively simple to create.

Both liquid and elastic scaling images start out with a regular `img` element in the (X)HTML:

```
<img src="styx.jpg" alt="my cat Styx">
```

Notice that this `img` element has no `width` or `height` attributes, as it normally would. You control the dimensions with CSS instead.

■ **NOTE:** This rule will make all images 50 percent as wide as their parents. In a real page, you would probably add an `id` or `class` to the specific image you wanted to scale and use that `id` or `class` as the selector in the CSS.

For a liquid image, create a CSS rule to set the image's width to a percentage value:

```
img {
   width: 50%;
   }
```

No `height` value is necessary; the browser determines the height that will proportionately constrain the image's dimensions. If you were more concerned about making the height of your image stay proportional to its parent's height than you were with width, you could use the `height` property in the CSS and leave off the `width` property. Just make sure not to use the two together.

As with all percentage dimensions, the percentage `width` value you choose is relative to the width of the parent element. As you change the width of the parent element, the image scales to match (**Figure 9.1**).

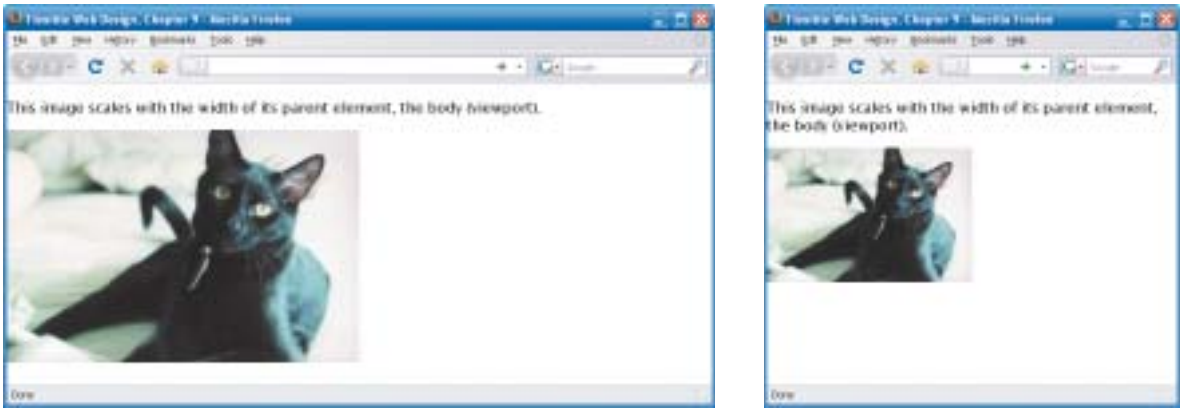

FIGURE 9.1 The image is set to 50 percent of the width of its parent, the body element, so it always takes up half the width of the viewport.

If you want the image to scale with the text size instead of the width of the parent element, simply change the `width` value in the CSS to an em value:

```
img {
   width: 20em;
   }
```

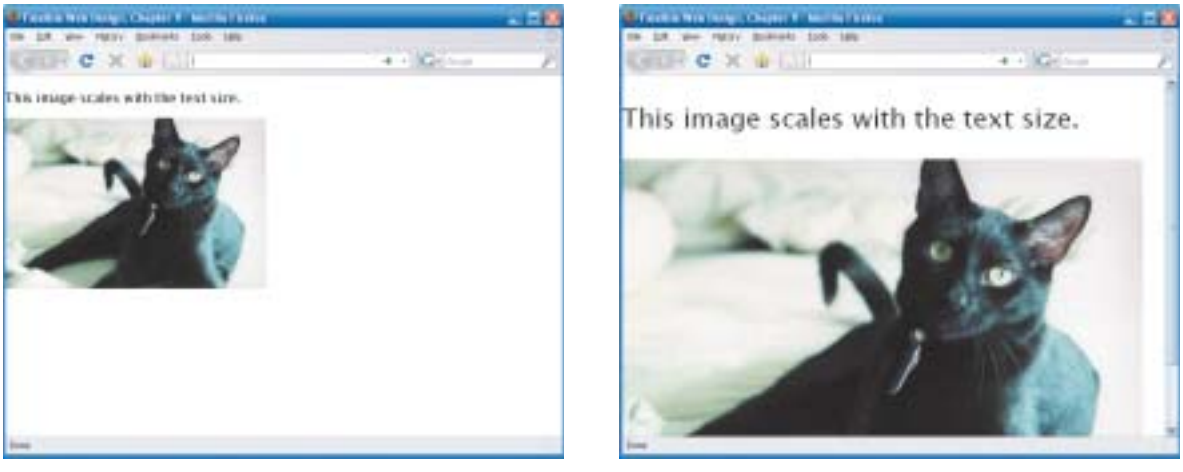

FIGURE 9.2 The image is set to 20 ems, so it will always be roughly 40 text characters wide.

As we discussed in Chapter 2, any time a browser scales an image, there's going to be some distortion, but you can keep it minimal by starting out with a very large image so the browser will usually be scaling it down.

■ **TIP:** If the image must stay above a certain size to remain "readable," add a pixel `min-width` value too.

■ **NOTE:** The page showing this completed technique is scale.html in the ch9_examples.zip file.

To assure that the browser *always* scales the image down, not up, you can set a maximum width on the image that matches its set pixel width:

```
img {
   width: 20em;
   max-width: 500px;
   }
```

Now the image will scale only until it grows to 500 pixels wide; thereafter it will act as any other fixed-width image (**Figure 9.3**).

FIGURE 9.3 The image does not stay proportional to the text size once it reaches its maximum width of 500 pixels.

SIMULATE IMAGE SCALING WITH JAVASCRIPT

If you don't want the browser to scale your images at all, yet you want them to change in size based on the amount of space available, you can use JavaScript to swap in differently sized versions of the same image. The JavaScript detects the user's viewport size and chooses the appropriate version of the image to show. This works in the same way that resolution-dependent layouts (discussed in Chapter 1) swap in different CSS files based on viewport size.

A live site that uses image-swapping to simulate scaling is Art & Logic (www.artlogic.com). There are five illustrations under the banner on the home page. Try narrowing or widening your browser window; the images don't scale in real time in most browsers, but as soon as you stop moving the window, they jump to a different size to match the available new space.

Hiding and Revealing Portions of Images

Another way to change the amount of screen area an image takes up is to dynamically change how much of the image is shown at any given time. The image itself doesn't change in size—the amount of space in which it's allowed to show does, and the rest of the image just remains hidden outside of that space. I call this "variable cropping," and you saw an example of it in Figure 9.2.

You can create a variable cropping effect with either background or foreground images. Both look the same, but each is specially suited to different situations.

VARIABLE CROPPING WITH BACKGROUND IMAGES

Putting the image that you want to dynamically crop in the background is ideal when the image is purely decorative. This technique lets you keep the image in the CSS with the other decorative images, so if you later change the look of the site, all the decorative images can be changed in a single style sheet instead of having to replace multiple `img` elements across multiple pages of the site. By keeping the decorative image as a CSS background, you're also making it likely that the image won't print when the user prints the page—background printing is turned off by default in all major browsers—so the user can save ink by printing only content.

To use a CSS background image, you'll first need an element on which to place the background. This example will use a `div`:

```
<div id="background"></div>
```

The `div` is completely empty; it contains no content, but exists simply to hold a background image. If you have a more semantic element you can hang the background on instead, use it. For instance, perhaps the image you want to dynamically crop sits above an `h3` element and matches it in width. You could add the image as a background to the `h3` element and give the `h3` enough top padding to make sure its text sits below the image, not on top of it.

Next, create a rule for this `div` that sets the image as its non-tiling background:

```
div#background {
   background: url(styx.jpg) no-repeat;
   border: 2px solid #000;
}
```

I've added a border on to this `div` as well so you can easily see where its edges lie. Right now, with no content within the `div`, it will collapse to zero height. Add dimensions to the `div` to prop it open:

```
div#background {
   width: 50%;
   height: 330px;
   background: url(styx.jpg) no-repeat;
   border: 2px solid #000;
}
```

■ **TIP:** To dynamically change the height of the image as well as or instead of the width, use a flexible value for the `height` property .

The width is set to some flexible dimension—either a percentage, as I've done here, or an em value to make it elastic—so that the `div` can change in width to show more or less of the image. The height is set to the pixel height of the image so that the entire height of the image will show at all times.

The `div` will now always be 50 percent as wide as the viewport; its background image doesn't change in size, but gets cropped to a varying degree from the right side (**Figure 9.4**).

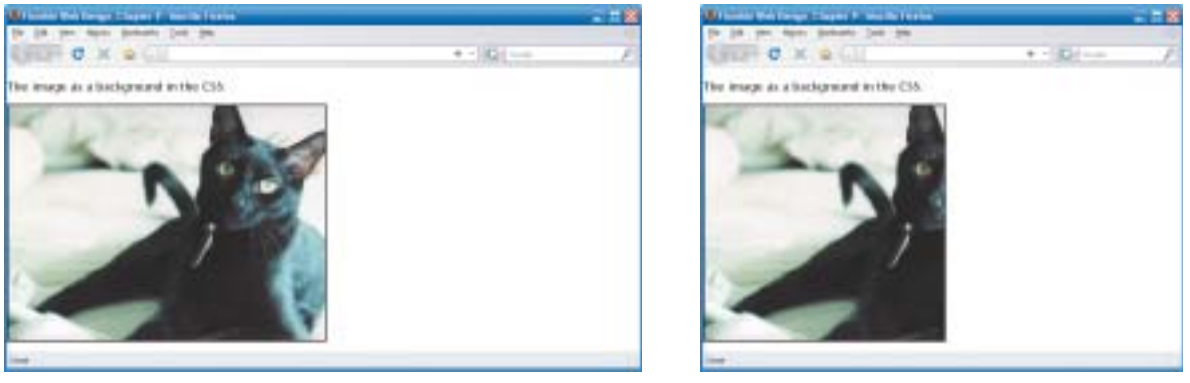

FIGURE 9.4 As the width of the browser window decreases, the black-bordered `div` narrows and cuts off more and more of the right side of its background image.

However, this particular image would look better cropped from the left side, as the cat's face is on the right side of the photo. To specify from where the image gets cropped, use the `background-position` property, or its shorthand in the `background` property, to change the alignment of the image within the `div`:

```
div#background {
   width: 50%;
   height: 330px;
   background: url(styx.jpg) no-repeat right;
   border: 2px solid #000;
}
```

The image is now anchored to the right side of the `div`, so more or less of its left side shows as the `div` changes in size (**Figure 9.5**).

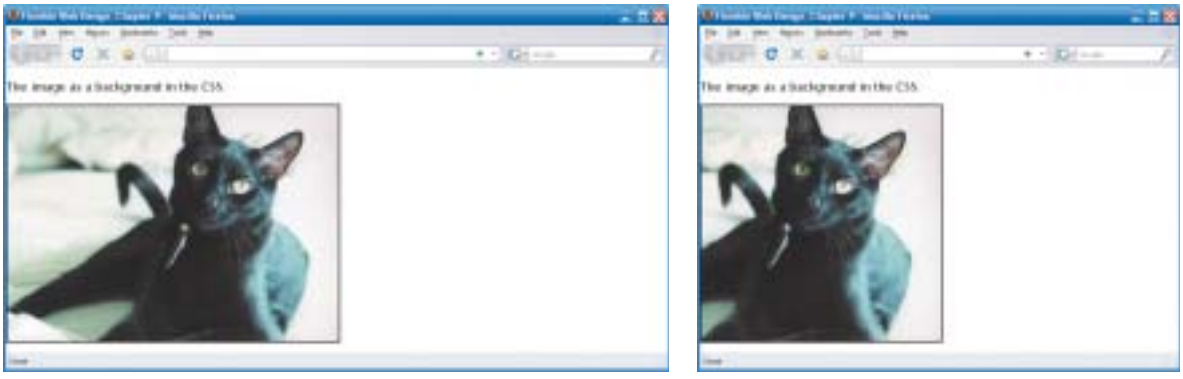

FIGURE 9.5 With the background anchored to the right side of the `div`, more of the left side of the image is cut off when the browser window is narrowed.

This is all the CSS necessary to get the basic variable cropping technique working, but you can add a few other enhancements if you like. For instance, right now, once the `div` exceeds the width of the image, empty white space shows within the `div`. There are a few ways you could handle this. You could add a background color to the `div` as well that would fill up whatever space the image cannot; if you blend the edge of the image into this background color, the effect can look seamless, as in Figures 2.16 and 2.17. Or, you could add a maximum width to the `div` so it can never grow larger than the image. You could also add minimum widths, as well as maximum and minimum heights, to ensure that the `div` can never grow or shrink past particular points in the image.

■ **NOTE:** The page showing this completed technique is crop_background.html in the ch9_examples.zip file.

VARIABLE CROPPING WITH FOREGROUND IMAGES

If the image that you want to dynamically crop is functional content, you'll want to keep it as a foreground image by placing it in the (X)HTML using the `img` element. You can ask yourself these questions to determine if the image is content, not decoration:

- Does the image convey information that I ought to put as text in an `alt` attribute?
- Do I want to make sure the image always prints because without it the printout wouldn't make sense or be complete?
- Do I want to link the image?

If the answer to any of these questions is yes, the image is content and should be kept in the (X)HTML. CSS background images can't achieve any of these goals—at least not without some complicated workarounds and hacks, all of which are quite silly, considering how easily a simple `img` element can achieve all this.

■ **NOTE:** The `img` element has an `alt` attribute providing the text equivalent of the image. You can't do this with a CSS background image.

As with the background-image version of the variable cropping technique, you'll need some block element in the (X)HTML to hold the image. We'll use a `div` again; this time it won't be empty, but will instead contain the `img` element:

```
<div id="foreground">
   <img src="styx.jpg" alt="my cat Styx" width="500"
height="330">
</div>
```

Just as before, the `div` needs to have a flexible width and a height set to the pixel height of the image:

```
div#foreground {
   width: 50%;
   height: 330px;
   border: 2px solid #000;
}
```

So far, all we have is a regular `div` holding a regular image—there's nothing yet that makes this a variable cropping technique. If the image is bigger than the `div`, it doesn't get cropped, but simply overflows (**Figure 9.6**).

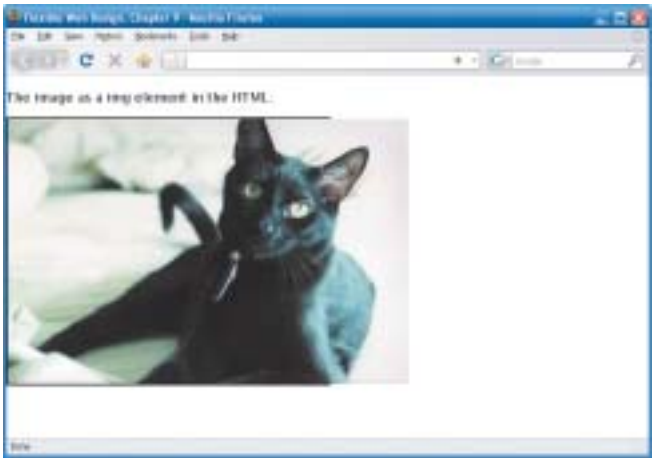

FIGURE 9.6 The image inside the `div` hangs out the right side of the `div`, overlapping its black borders, when the `div` becomes narrower than the image.

To get the cropping effect, add `overflow: hidden;` to the CSS rule:

```
div#foreground {
   overflow: hidden;
   width: 50%;
   height: 330px;
   border: 2px solid #000;
}
```

Now whatever portion of the image would overflow out of the `div` is hidden from view (**Figure 9.7**).

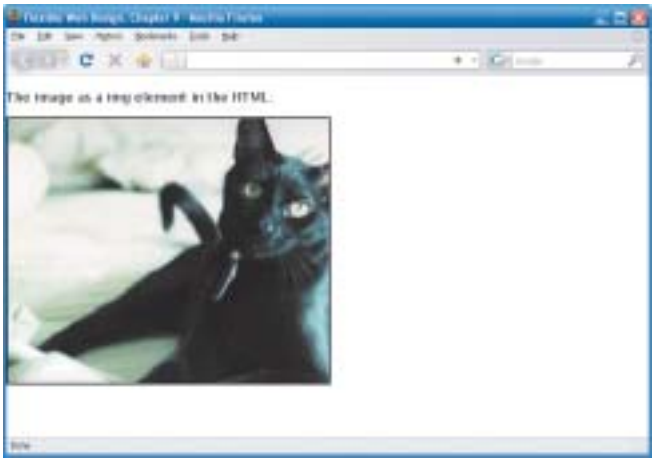

FIGURE 9.7
With `overflow` set to `hidden`, the extra portion of the image is now hidden from view.

Once again, though, it would be better for this image to be cropped from the left side, not the right. We can't use the `background-position` property this time because it's not a background image. To change how a foreground image is anchored within its parent, you can float the image:

```
div#foreground img {
   float: right;
}
```

This anchors the image to the right side of the `div`, so more or less of its left side shows as the `div` changes in size. Using a foreground image results in an effect that looks exactly like using a background image (seen in Figure 9.5), but the foreground image has alternative text, and you could also easily add a link to it.

■ **NOTE:** The page showing this completed technique is crop_foreground.html in the ch9_examples.zip file. You can also view both background and foreground techniques together on the page crop.html.

Creating Sliding Composite Images

Perhaps you don't want either end of your image dynamically cropped off—there may be important content on each side that you want to always keep in view. You could scale the entire image instead, which would keep the entire width of the image always visible, but it would also change the vertical space the image takes up, and perhaps you don't want this either. This is the perfect time to try using what I call a composite image.

Creating what appears to be a single image out of multiple pieces that slide over and away from each other takes a little more work on the graphics side

than the variable width image techniques we've gone over so far. The real-web-site example of this composite image technique shown in Figures 2.23 and 2.24 used two images to create the effect; you can use an unlimited number of images, but we'll keep it simple and use two for our own alien-invasion example as well.

One image is going to be at least partially overlapping the other, so at least the topmost image needs to have a transparent background. (You may choose to make the lower image transparent too, to allow parts of the main page background to show through, for instance.) You can use either a GIF with index transparency or a PNG with alpha transparency. PNGs are more versatile, since they can lay over any other color or pattern without the skinny colored edge that shows around GIF images when they're placed over something that's a different color than they were optimized for. PNGs can also have variable degrees of transparency, instead of each pixel being either 100 percent transparent or 100 percent opaque.

We'll use an alpha-transparent PNG for our top image in this example. **Figure 9.8** shows our flying saucer image in Adobe Fireworks; the checkerboard background indicates the transparent areas of the image. **Figure 9.9** shows the image that the flying saucer will be laid on top of—a photo of the Chicago skyline—which can be saved as an ordinary JPG.

FIGURE 9.8 The flying saucer image has large areas of partial or total transparency through which the skyline image will be visible.

FIGURE 9.9 The skyline image is completely separate from the flying saucer image.

Once you have your images made, you need two block elements to place each on as a background image. One block element needs to be nested inside the other. In a real page, you'd want to make use of block elements that were already in place as much as possible, such as existing wrapper and header `div`s. For this simple example, we'll use two empty `div`s:

```
<div id="outer"><div id="inner"></div></div>
```

Next, you need to create rules placing each image as a non-repeating background on each `div`, with the image you want on the bottom used for the background of the outer `div`:

```
#outer {
   background: url(skyline.jpg) no-repeat;
}
#inner {
   background: url(ufo.png) no-repeat;
}
```

Since the `divs` are empty, they also need dimensions added to them to stop them from collapsing entirely, as well as to create the flexible behavior that we want:

```
#outer {
   width: 100%;
   max-width: 1000px;
   height: 300px;
   background: url(skyline.jpg) no-repeat;
}
#inner {
   width: 100px;
   height: 250px;
   background: url(ufo.png) no-repeat;
}
```

Both of the images now show on the page, with the flying saucer layered over the skyline (**Figure 9.10**). However, the flying saucer never moves when the window changes in size—it's always pinned to the top left corner of the skyline photo. That's because the `div` for which it's a background begins in that corner, and non-repeating, non-positioned background images display in the top left corner by default.

■ **NOTE:** Remember that IE 6 and earlier do not support alpha-transparent PNGs, so the flying saucer image will have a solid gray background in those browsers. Use the `alphaimageloader` hack described in the exercise section of Chapter 7 to fix this.

FIGURE 9.10 The flying saucer image is layered over the skyline image to create the appearance of a single image, but the flying saucer isn't yet in the place we want it.

There are a couple ways to fix this: use the `background-position` property to change where the flying saucer displays within the `div`, or move the entire `div`. Either option is fine, but the latter seems a little easier to understand and implement—at least to me—so that's what we'll use here.

We'll move the `div` using absolute positioning; floating would work as well. First, add `position: relative;` to the `#outer` rule to make that `div` act as the containing element for the absolutely positioned inner `div`. Then, add `position: absolute;` as well as `top` and `right` values to the `#inner` rule:

```
#inner {
   position: absolute;
   top: 50px;
   right: 50px;
   width: 100px;
   height: 250px;
   background: url(ufo.png) no-repeat;
}
```

■ **NOTE:** The page showing this completed technique is composite.html in the ch9_examples.zip file.

Now the flying saucer image will always be 50 pixels away from both the top and right edges of the skyline photo. Because the outer `div` has a flexible width, its right edge moves as the window is resized, which in turn makes the flying saucer image move as well (**Figure 9.11**).

FIGURE 9.11 The flying saucer image now appears to move as the browser window changes in size.

Creating Flexible Collections of Images

You now know several ways to make individual images flexible to either their parent's dimensions or text size, but what about when you need a whole group of images to be flexible as a whole? Let's go over how to make

two of the most common types of image collections—teaser thumbnail lists and image galleries—flexible too.

Teaser Thumbnail Lists

A teaser thumbnail list is my own personal name for the design convention of a list where each item is made up of a title, short description, and thumbnail image. Figure 2.22 is one example of a teaser thumbnail list, as is the list of featured pets on the home page of our fictional Beechwood Animal Shelter site (Figure 2.43). These types of lists can be built in many different ways, but many techniques result in lists that are not flexible or not as accessible to the end user as they could be.

FIGURE 9.12 Each teaser thumbnail list item is made up of a title, short description, and thumbnail image.

Figure 9.12 shows the teaser thumbnail list I'll be using as an example throughout this section. I've chosen the following HTML as the most semantic way of marking up each of the elements of this design component:

```
<h1>Seafood of the Month Club 2008</h1>
<ul>
   <li>
      <h2>January</h2>
      <img src="january.jpg" alt="" width="100" height="100">
      <p>Seared sea scallops, served with mushy peas.</p>
   </li>
   <li>
      <h2>February</h2>
      <img src="february.jpg" alt="" width="100" height="100">
```

```
      <p>Soy-glazed salmon, served with coconut and bell pepper
broccoli slaw.</p>
   </li>
   <li>
      <h2>March</h2>
      <img src="march.jpg" alt="" width="100" height="100">
      <p>Tuna steak with ginger-shitake cream sauce, served with
sesame broccoli and brown rice.</p>
   </li>
</ul>
```

You'll note that the `img` elements follow the `h2` heading elements, even though Figure 9.12 shows the images appearing on the same line as the headings. You'll need to find a way to get the images to move up to sit beside the headings, even though they come later in the source. Luckily, you're already an expert at doing just that—you have several negative margin layouts you can use to achieve such an effect. A teaser thumbnail list is essentially nothing more than a two-column layout. This particular one has a fixed-width left "sidebar" and a liquid right "main content area," so any negative margin technique that works for two-column, hybrid liquid-fixed layouts will work here.

To turn this into a negative margin "layout," the basic steps are:

1. Create an empty space on the left side of the list.
2. Use negative margins to pull each image into that space.
3. Float all the elements within each list item so they can sit side by side.

We'll create the empty space on the left side of the list using a left margin on the `ul` element that is equal to the width of the images (100 pixels) plus the width of the gap we want between each image and its accompanying text (15 pixels):

```
ul {
   margin: 0 0 0 115px;
   padding: 0;
   list-style: none;
}
```

This rule also gets rid of some of the default list styling, including the bullets. The rest of the default list styling that needs to be overridden is on the list items:

```
li {
```

```
    margin: 0 0 20px 0;
    padding: 0;
}
```

This removes the default left margin and padding that some browsers add to `li` elements, as well as adds 20 pixels of space below each list item to space them out from each other.

EASIER TEASER THUMBNAIL LIST CREATION OPTIONS

If you're still feeling a little uneasy about creating negative margin layouts, there are a few easier ways to create teaser thumbnail lists that achieve the same visual effect:

- Remove the images from the (X)HTML altogether and simply use CSS background images to place the thumbnails next to each heading-paragraph pair. This is a perfectly acceptable option—*if the thumbnails are purely decoration*. If you want the images to show even when the user has CSS off or unavailable, as well as when the user prints the page, you should keep them in the (X)HTML. Doing so also allows you to add alternative text and links to the images—not important if they're just decoration, but essential if they're content.
- You can keep the images in the (X)HTML and still avoid a negative margin technique by placing the image before the heading within each list item. This markup isn't quite as ideal as placing the headings first—after all, the headings should head or precede the images and text they describe. But the markup is still quite clean and fairly semantic, and it does allow you to use a simple floats-with-matching-side-margins technique: just float the images to the left, and give the headings and paragraphs left margin values that exceed the images' width.
- Another way to keep the images in the (X)HTML but use the most semantic markup of headings-first is to use the same unit of measurement for both the thumbnails and the blocks of text beside them. The example we're going over here is essentially a hybrid layout: the thumbnails are fixed-width and the text beside them is liquid. This is the most common type of teaser thumbnail list. But you could make the thumbnails scalable instead. This would allow you to use a simple float-all-the-columns layout method: float the images to the left and the headings and paragraphs to the right, all with matching units of measurement. You'll get to try this method in the exercise section at the end of the chapter.

FIGURE 9.13 A large empty space on the left side of the list stands ready to receive the thumbnails.

You can now pull the image into the empty space on the left:

```
img {
   float: left;
   margin-left: -115px;
}
```

This positions the images correctly horizontally, but not vertically (**Figure 9.14**). To get them to move up and sit beside the headings, the headings have to be floated, as do the paragraphs:

```
h2 {
   float: right;
   width: 100%;
   margin: 0;
}
p {
   float: right;
```

```
    width: 100%;
    margin: 0;
}
```

FIGURE 9.14 Negative left margins pull the images to the left, but don't pull them up to sit beside the headings.

The `width: 100%;` declarations ensure that each piece of text fills up the entire width to the right of the images, instead of each element shrinkwrapping to its content, as floats without declared widths do naturally.

The images have now moved up to sit beside the headings, but they overlap each other (**Figure 9.15**). This is because the list items contain only floated content now, which is out of the flow, and have thus collapsed down to zero height.

FIGURE 9.15 Floating the text elements to the right allows the images to sit beside the headings, but the list items will not expand to hold the full height of the thumbnails when everything inside the list items is floated.

■ **NOTE:** The page showing this completed technique is teaser.html in the ch9_examples.zip file.

To address this, we need to use a float containment method to get each list item to encompass all of the floated elements within it. Floating the `li` elements themselves is one easy way to do this:

```
li {
   float: left;
   width: 100%;
   margin: 0 0 20px 0;
   padding: 0;
}
```

FIGURE 9.16 No matter which is longer—thumbnail or accompanying text—the list items remain spaced out from each other.

The list items are now properly spaced out from each other, whether the text within them is shorter or longer than the thumbnail images (**Figure 9.16**).

The only problem is that floating the list items made the images disappear in IE 6 and earlier. To fix this, add `position: relative;` to both the `li` and `img` rules:

```
li {
   float: left;
   width: 100%;
   margin: 0 0 20px 0;
   padding: 0;
   position: relative;
}
img {
   float: left;
   margin-left: -115px;
   position: relative;
}
```

Thumbnail Image Galleries

Although images are usually fixed in width, you can line them up side by side and still create a block of image thumbnails that can change in total width. You saw an example of this in Figure 2.41, where the thumbnails wrapped onto a variable number of lines to accommodate the liquid width of the content area. Another way to create a flexible image gallery is to make all of the thumbnails scale, using one of the scalable image techniques you learned at the start of the chapter. Let's go over both options.

WRAPPING THE THUMBNAILS

The two behaviors you want thumbnails in a flexible image gallery to achieve—sitting side by side and wrapping onto more lines as needed—are both native behaviors of floats. So, the only thing you need to do to make thumbnails wrap is to float each one in the same direction.

You could just place the images straight into the (X)HTML with no container, and float each of the `img` elements. But a more semantic way to mark up a group of images is to use an unordered list, which offers you more styling possibilities as well. Put each `img` into an `li` element:

```
<ul>
   <li><img src="january.jpg" alt="January" width="100" height="100">
</li>
   <li><img src="february.jpg" alt="February" width="100" height="100">
</li>
   <li><img src="march.jpg" alt="March" width="100" height="100">
</li>
   <li><img src="april.jpg" alt="April" width="100" height="100">
</li>
   <li><img src="may.jpg" alt="May" width="100" height="100">
</li>
   <li><img src="june.jpg" alt="June" width="100" height="100">
</li>
   <li><img src="july.jpg" alt="July" width="100" height="100">
</li>
</ul>
```

Next, remove the default list styling:

```
ul {
   margin: 0;
   padding: 0;
   list-style: none;
}
li {
   margin: 0;
   padding: 0;
}
```

Now, simply float the `li` elements all to the left, and give them some margin on their right and bottom sides to space them out from each other:

```
li {
   float: left;
   margin: 0 10px 10px 0;
   padding: 0;
}
```

■ **NOTE:** The page showing this completed technique is gallery_wrap.html in the ch9_examples.zip file.

That's all you need to do to create a basic, wrapping thumbnail image gallery (**Figure 9.17**). The perfect number of thumbnails always sits on each line, no matter the viewport width, so you don't get a horizontal scrollbar or a really large gap on the right. If you didn't want the gallery to take up the entire width of its parent, simply assign a width to the `ul` element; as long as the width is a percentage or em value, the list will still be flexible and the thumbnails will still wrap.

FIGURE 9.17 The number of thumbnails on each line adjusts to the space available in the viewport.

You may have noticed that all of the thumbnails in this example share the same dimensions. Variable widths on thumbnails are not a problem, but variable heights make this wrapping thumbnail technique fail. **Figure 9.18** shows the same page with the height of some of the thumbnails increased. When the thumbnails wrap, they move as far over to the left as they can go. But when one of the thumbnails in the previous row hangs down farther than the rest, it impedes the new row of thumbnails from moving any further to the left, and big gaps can be left in the rows.

FIGURE 9.18 The extra height on the second thumbnail blocks the fifth thumbnail from moving all the way to the left, leaving a gap in the second row. The same problem happens in the third row.

There are a couple ways you can modify the basic technique to work with variable height thumbnails. The simplest is to assign a fixed height to the `li` elements that matches the height of the tallest thumbnail. This makes all the `li` elements match in height, instead of depending on the size of the images inside them to dictate their heights, so there are no taller list items sticking down any more that might block the wrapping thumbnails.

If you can't assign a fixed height to the `li` elements, though, perhaps because your thumbnails are pulled into the page dynamically and you don't know what the largest height will be, there's still hope. You'll need to use something other than floats to get the thumbnails sitting side by side—and that something is `display: inline-block`.

An inline block mixes the attributes of block and inline elements. It's placed on the same line as adjacent content, like inline elements are, but you can assign it width, height, margin, and padding, just like a block element.

Since inline block elements sit side by side by default, when you apply a `display` value of `inline-block` to the `li` elements, you can get rid of the `float` declaration:

```
li {
   display: inline-block;
   margin: 0 10px 10px 0;
   padding: 0;
}
```

In browsers that support `inline-block`, that's all you need to do to keep the thumbnails from hanging up on each other when they wrap (**Figure 9.19**). If you want the thumbnails aligned along their top edges, as they were when we used floats, simply add `vertical-align: top;` to the `li` rule.

FIGURE 9.19 When the thumbnails are turned into inline blocks, instead of floats, they no longer hang up on one another.

In browsers that don't support `inline-block`, the thumbnails will just display straight down, each on its own line. These browsers include versions of IE earlier than 8 and versions of Firefox earlier than 3. Let's take care of the IE problem first.

IE 7 and 6 support `inline-block` only on elements that are inline by default, so you can trick these browsers into making `inline-block` work by setting the list items to `display: inline`. Hide this rule inside a conditional comment that only IE 7 and earlier can read:

```
<!--[if lte IE 7]>
<style type="text/css">
li {
   display: inline;
}
</style>
<![endif]-->
```

This fixes the problem in IE; now onto Firefox.

Versions of Firefox prior to 3 lacked support for `inline-block` but had their own proprietary values, `-moz-inline-box` and `-moz-inline-stack`, for the

■ **NOTE:** Browser-proprietary values will make your CSS fail validation checks. But they don't hurt any browsers that can't understand them, so don't worry about the lack of validation—validation is just a means to an end, not necessarily an end unto itself.

display property that worked almost identically. Add either of these values to the li rule:

```
li {
   display: -moz-inline-box;
   display: inline-block;
   margin: 0 10px 10px 0;
   padding: 0;
   vertical-align: top;
}
```

■ **TIP:** Add the proprietary value before the proper value, as done here, so that the browser will use the latter one when it's able to support it.

This fixes the problem in Firefox 2 without hurting any other browsers, including Firefox 3—they all just ignore the -moz-inline-box value. If you have links wrapped around the images, however, you'll have just a bit more work to do. Firefox 2 will position the images incorrectly and not make the entire image clickable when you nest *a* elements inside the li elements. To fix this, turn the *a* elements into blocks:

■ **NOTE:** The page showing this completed technique is gallery_wrap_irregular.html in the ch9_examples.zip file.

```
li a {
   display: block;
}
```

Again, this doesn't hurt other browsers.

SCALING THE THUMBNAILS

If you want the thumbnails in your image gallery to scale instead of—or in addition to—wrapping, you need to add the scalable foreground image technique (that we went over earlier in the chapter) to the basic wrapping thumbnail gallery CSS.

The first step in making scalable foreground images, you may remember, is to remove the width and height dimensions from the img elements in the (X)HTML:

```
<ul>
   <li><img src="january.jpg" alt="January"></li>
   <li><img src="february.jpg" alt="February"></li>
   <li><img src="march.jpg" alt="March"></li>
   <li><img src="april.jpg" alt="April"></li>
   <li><img src="may.jpg" alt="May"></li>
   <li><img src="june.jpg" alt="June"></li>
   <li><img src="july.jpg" alt="July"></li>
</ul>
```

Next, add a percentage or em width onto the `li` elements:

```
li {
   float: left;
   width: 18%;
   margin: 0 10px 10px 0;
   padding: 0;
}
```

■ **NOTE:** The page showing this completed technique is gallery_scale.html in the ch9_examples.zip file.

Finally, add a rule for the `img` elements that sets their widths to 100 percent so they always fill up the variable size of their parent list items:

```
img {
   width: 100%;
}
```

The thumbnails now scale with the browser window (**Figure 9.20**).

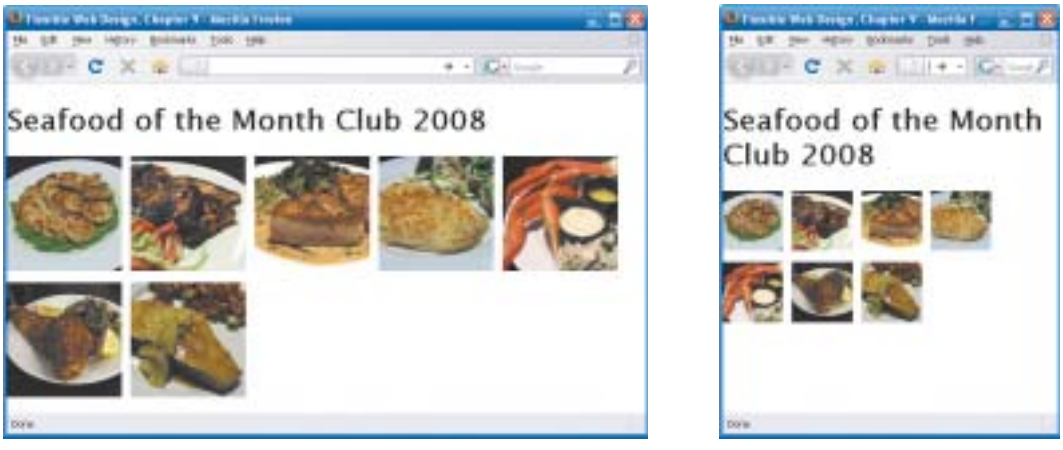

FIGURE 9.20 The thumbnails still wrap to a small degree, but now their primary method of adjusting to the viewport is to scale with it.

If you want to avoid the blurriness or pixelation that happens when browsers scale images past their native dimensions, you can add a maximum width onto the images that matches their pixel widths:

```
img {
   width: 100%;
   max-width: 100px;
}
```

When the thumbnails reach this `max-width` value, they will stop scaling. The list items, however, will not, so the images will appear to move farther apart from each other, still filling up the available space (**Figure 9.21**).

FIGURE 9.21 Once the thumbnails reach their maximum widths, they will stop scaling, but will still adjust to the viewport size by moving farther apart to fill the space.