3
Embracing Fluid Layouts

When I first started making websites at the end of the 1990s, layout structures were table based. More often than not, all the sectioning up of screen real estate was done with percentages. For example, a left navigation column might be 20 percent whilst the main content area would be 80 percent. There weren't the vast differences in browser viewports we see today so these layouts worked and scaled well across the limited range of viewports. Nobody much cared that sentences looked a little different on one screen compared to another. However, as CSS-based designs took over, it enabled web-based designs to more closely mimic print. With that transition, for many (including myself), proportionally based layouts dwindled for many years in favor of their rigid, pixel-based counterparts.

Like all great designs and solutions, they come back around. The mini car, permed hair (I wish!), and flared jeans have all made their comebacks over the years. Now, it's time for proportional layouts to make a re-appearance.

In this chapter, we shall:

- Learn why proportional layouts are necessary for responsive design
- Convert pixel-based element widths to proportional percentages
- Convert pixel-based typography sizes to their em-based equivalent
- Understand how to find the context for any element
- Learn how to make images scale fluidly
- Learn how to serve different images to different screen sizes
- Understand how media queries can work with fluid images and layouts
- Create a responsive layout from scratch using a CSS grid system

Fixed layouts aren't future proof

As I mentioned, since the "table layout" days, I've had little call to use proportional layouts. Typically, I've been asked to code HTML and CSS that best matches a design composite that almost always measures 950-1000 pixels wide. If the layout was ever built with a proportional width (say, 90 percent), the complaints would have arrived quickly, "It looks different on my monitor". Web pages with fixed, pixel-based dimensions were the easiest way to match the fixed, pixel-based dimensions of the composite.

Even in more recent times, when using media queries to produce a tweaked version of a layout, specific to a certain popular device such as an iPad or iPhone (as we did in *Chapter 2*, *Media Queries: Supporting Differing Viewports*), the dimensions could still be pixel-based as the viewport was known. However, whilst many might enjoy the possibility of re-charging a client each time they need a site tweaked for today's newest gizmo, it's not exactly a future proof way of building web pages. As more and more varied viewports are being introduced, we need some way of provisioning for the unknown.

Why proportional layouts are essential for responsive designs

Whist media queries are incredibly powerful we are now aware of some limitations. Any fixed width design, using only media queries to adapt for different viewports will merely "snap" from one set of CSS media query rules to the next with no linear progression between the two. From our own experience in *Chapter 2*, *Media Queries: Supporting Differing Viewports*, where a viewport fell between the fixed-width ranges of our media queries (as may be the case for future unknown devices and their viewports) the design required horizontal scrolling in the browser. Instead, we want to create a design that flexes and looks good on all viewports, not just particular ones specified in a media query. I'll *cut to the chase*. (See what I did there? It's a film-based saying to match our film-based site… No? I'll get my coat…) We need to switch our fixed, pixel-based layout to a fluid proportional one. This will enable elements to scale relative to the viewport until one media query or another modifies the styling.

The symbiosis of proportional layout and media queries

I've already mentioned Ethan Marcotte's article on Responsive Web Design at A List Apart (`http://www.alistapart.com/articles/responsive-web-design/`). Whilst the tools he used (fluid layout and images, and media queries) were not new, the application and embodiment of the ideas into a single coherent methodology were. For many working in web design, his article was the genesis of new possibilities. Indeed, new ways to create web pages that offered the best of both worlds; a way to have a fluid flexible design based on a proportional layout, whilst being able to limit how far elements could flex with media queries. Putting them together forms the core of a responsive design, creating something truly greater than the sum of its parts.

Amending a design from fixed to proportional layout

Typically, for the foreseeable future, any design composite you receive or create will have fixed dimensions. Currently we measure (in pixels) the element sizes, margins, and so on within the graphics files from Photoshop, Fireworks, and so on. We then punch these dimensions directly into our CSS. The same goes for text sizes. We click on a text element in our image editor of choice, note the font size, and then enter it (again, often measured in pixels) into the relevant CSS rule. So how do we convert our fixed dimensions into proportional ones?

A formula to remember

It's possible I'm coming off too much of an Ethan Marcotte fan boy, but at this point it's essential that I provide another large tip of the hat (it should probably be a bow, maybe even a kneel) to him. In Dan Cederholm's excellent book, *Handcrafted CSS*, Mr. Marcotte contributed a chapter covering fluid grids. In it, he provided a simple and consistent formula for converting fixed width pixels into proportional percentages:

target ÷ context = result

Smells a bit like an equation to you? Fear not, when creating a responsive design, this formula soon becomes your new best friend. Rather than talk any more theory, let's put the formula to work converting the current fixed dimension for the *And the winner isn't...* site to a fluid percentage based layout.

If you remember, back in *Chapter 2*, *Media Queries: Supporting Differing Viewports*, we established that the basic markup structure of our site looked like this:

```
<div id="wrapper">
  <!-- the header and navigation -->
  <div id="header">
    <div id="navigation">
      <ul>
        <li><a href="#">navigation1</a></li>
        <li><a href="#">navigation2</a></li>
      </ul>
    </div>
  </div>
  <!-- the sidebar -->
  <div id="sidebar">
    <p>here is the sidebar</p>
  </div>
  <!-- the content -->
  <div id="content">
    <p>here is the content</p>
  </div>
  <!-- the footer -->
  <div id="footer">
    <p>Here is the footer</p>
  </div>
</div>
```

Content was later added but what's important to note here is the CSS we are currently using to set the widths of the main structural (header, navigation, sidebar, content, and footer) elements. Note, I've omitted many of the styling rules so we can concentrate on structure:

```
#wrapper {
  margin-right: auto;
  margin-left: auto;
  width: 960px;
}

#header {
  margin-right: 10px;
  margin-left: 10px;
  width: 940px;
}

#navigation {
```

```
  padding-bottom: 25px;
  margin-top: 26px;
  margin-left: -10px;
  padding-right: 10px;
  padding-left: 10px;
  width: 940px;
}

#navigation ul li {
  display: inline-block;
}

#content {
  margin-top: 58px;
  margin-right: 10px;
  float: right;
  width: 698px;
}

#sidebar {
  border-right-color: #e8e8e8;
  border-right-style: solid;
  border-right-width: 2px;
  margin-top: 58px;
  padding-right: 10px;
  margin-right: 10px;
  margin-left: 10px;
  float: left;
  width: 220px;
}
#footer {
  float: left;
  margin-top: 20px;
  margin-right: 10px;
  margin-left: 10px;
  clear: both;
  width: 940px;
}
```

All the values are currently set using pixels. Let's work from the outermost element and change them to proportional percentages using the *target* ÷ *context* = *result* formula.

All our content currently sits within a div with an ID of `#wrapper`. You can see by the CSS above that it's set with automatic margin and a width of 960 px. As the outermost div, how do we define what percentage of the viewport width it should be?

Setting a context for proportional elements

We need something to "hold" and become the context for all the proportional elements (content, sidebar, footer, and so on) we intend to contain within our design. We therefore need to set a proportional value for the width that the `#wrapper` should be in relation to the viewport size. For now, let's knock off a naught and roll with 96 percent and see what happens. Here's the amended rule for `#wrapper`:

```
#wrapper {
  margin-right: auto;
  margin-left: auto;
  width: 96%; /* Holding outermost DIV */
}
```

And here's how it looks in the browser window:

So far, so good! 96 percent actually works quite well here although we could have opted for 100 or 90 percents—whatever we felt and set the design within the viewport in the most aesthetically pleasing manner.

Now changing from fixed to proportional gets a little more complicated as we move inwards. Let's look at the header section first. Consider the formula again, *target ÷ context = result*. Our `#header` div (the target) sits within the `#wrapper` div (the context). Therefore, we take our `#header` (the target) width of 940 pixels, divide it by the width of the context (the `#wrapper`), which was 960 px and our result is .979166667. We can turn this into a percentage by moving the decimal place two digits to the right and we now have a percentage width for the header of 97.9166667. Let's add that to our CSS:

```
#header {
  margin-right: 10px;
  margin-left: 10px;
  width: 97.9166667%; /* 940 ÷ 960 */
}
```

And as both the `#navigation` and the `#footer` divs also have the same declared width, we can swap both of those pixel values to the same percentage-based rule.

Finally, before we take a peek in the browser, let's turn to the `#content` and `#sidebar` div's. As the context is still the same (960 px) we just need to divide our target size by that figure. Our `#content` is currently 698 px so divide that value by 960 and our answer is .727083333. Move the decimal place and we have a result of 72.7083333 percent—that's the width of the `#content` div in percentage terms. Our sidebar is currently 220 px but there's also a 2 px border to consider. I don't want the thickness of the right-hand border to expand or contract proportionately so that will stay at 2 px. Because of that I need to subtract some size from the width of the sidebar. So in the case of this sidebar, I have subtracted 2 px from the sidebar width and then performed the same calculation. I've divided the target (now, 218 px) by the context (960 px) and the answer is .227083333. Shift the decimal and we have a result of 22.7083333 percent for the sidebar. After amending all the pixel widths to percentages, the following is what the relevant CSS looks like:

```
#wrapper {
  margin-right: auto;
  margin-left: auto;
  width: 96%; /* Holding outermost DIV */
}

#header {
  margin-right: 10px;
  margin-left: 10px;
```

```
  width: 97.9166667%; /* 940 ÷ 960 */
}

#navigation {
  padding-bottom: 25px;
  margin-top: 26px;
  margin-left: -10px;
  padding-right: 10px;
  padding-left: 10px;
  width: 72.7083333%; /* 698 ÷ 960 */
}

#navigation ul li {
  display: inline-block;
}

#content {
  margin-top: 58px;
  margin-right: 10px;
  float: right;
  width: 72.7083333%; /* 698 ÷ 960 */
}

#sidebar {
  border-right-color: #e8e8e8;
  border-right-style: solid;
  border-right-width: 2px;
  margin-top: 58px;
  margin-right: 10px;
  margin-left: 10px;
  float: left;
  width: 22.7083333%; /* 218 ÷ 960 */
}
#footer {
  float: left;
  margin-top: 20px;
  margin-right: 10px;
  margin-left: 10px;
  clear: both;
  width: 97.9166667%; /* 940 ÷ 960 */
}
```

The following screenshot shows what it looks like in Firefox with the viewport around 1000 px wide:

All good so far. Now, let's go ahead and replace all the 10 px instances used for padding and margin throughout with their proportional equivalent using the same *target* ÷ *context* = *result* formula. As all the 10 px widths have the same 960 px context, the width in percentage terms is 1.0416667 percent (10 ÷ 960).

Can't we just round the numbers?

Some critics of responsive design techniques (for example, see `http://tripleodeon.com/2010/10/not-a-mobile-web-merely-a-320px-wide-one/`) argue that entering numbers such as .550724638 em into stylesheets is daft. You may wonder yourself, why aren't these simply rounded to something more sensible? The counter argument is that it's a more accurate answer to the question being asked. Providing a browser with the most accurate answer should make it more able to display that answer in the most accurate manner. As a related aside, if you stayed awake through more than a couple math classes I'm sure you've heard of the Golden Ratio (`http://en.wikipedia.org/wiki/Golden_ratio`)? The mathematical ratio, found and used throughout almost every discipline we know, is expressed as approximately 1:1.61803398874989 (if you want it to 10,000 decimal places, knock yourself out here `http://www.maths.surrey.ac.uk/hosted-sites/R.Knott/Fibonacci/phi10000dps.txt`). Not a neat number by any means but quite an important one. If the Golden Ratio can suffer such precise measurements, I'm inclined to believe our web designs can too.

Everything still looks fine at the same viewport size. However, the navigation area isn't behaving. If I bring the viewport size in, just a little, the links start to span two lines:

Furthermore, if I expand my viewport, the margin between the links doesn't increase proportionally. Let's take a look at the CSS associated with the navigation and try and figure out why:

```
#navigation {
  padding-bottom: 25px;
  margin-top: 26px;
  margin-left: -1.0416667%; /* 10 ÷ 960 */
  padding-right: 1.0416667%; /* 10 ÷ 960 */
  padding-left: 1.0416667%; /* 10 ÷ 960 */
  width: 97.9166667%; /* 940 ÷ 960 */
  background-repeat: repeat-x;
  background-image: url(../img/atwiNavBg.png);
  border-bottom-color: #bfbfbf;
  border-bottom-style: double; border-bottom-width: 4px;
}

#navigation ul li {
   display: inline-block;
}

#navigation ul li a {
  height: 42px;
  line-height: 42px;
  margin-right: 25px;
  text-decoration: none;
  text-transform: uppercase;
  font-family: Arial, "Lucida Grande", Verdana, sans-serif;
  font-size: 27px;
   color: black;
}
```

Well, on first glance, looks like our third rule there, the `#navigation ul li a`, still has a pixel-based margin of 25 px. Let's go ahead and fix that with our trusty formula. As the `#navigation` div is based on 940 px our result should be 2.6595745 percent. So we'll change that rule to be as follows:

```
#navigation ul li a {
  height: 42px;
  line-height: 42px;
  margin-right: 2.6595745%; /* 25 ÷ 940 */
  text-decoration: none;
  text-transform: uppercase;
  font-family: Arial, "Lucida Grande", Verdana, sans-serif;
  font-size: 27px;
  color: black;
}
```

That was easy enough! Let's just check all is OK in the browser…

Oh, wait, that isn't exactly what we were gunning for. OK, the links aren't spanning two lines but we don't have the correct proportional margin value, clearly. The navigation links look like one big word, and not one I can find in my dictionary…

It's always important to remember the context

Considering our formula again (*target* ÷ *context* = *result*), it's possible to understand why this issue is occurring. Our problem here is the context. Here's the relevant markup:

```
<div id="navigation">
  <ul>
    <li><a href="#">Why?</a></li>
```

```
      <li><a href="#">Synopsis</a></li>
      <li><a href="#">Stills/Photos</a></li>
      <li><a href="#">Videos/clips</a></li>
      <li><a href="#">Quotes</a></li>
      <li><a href="#">Quiz</a></li>
    </ul>
  </div>
```

As you can see our `<a href="#">` links sit within the `<li>` tags. They are the context for our proportional margin. Looking at the CSS for the `<li>` tags, we can see there are no width values set:

```
#navigation ul li { display: inline-block; }
```

As if often the case, it turns out that there are various ways of solving this problem. We could add an explicit width to the `<li>` tags but that would either have to be fixed-width pixels or a percentage of the containing element (the `navigation` div), neither of which allows any flexibility for the text that ultimately sits within them.

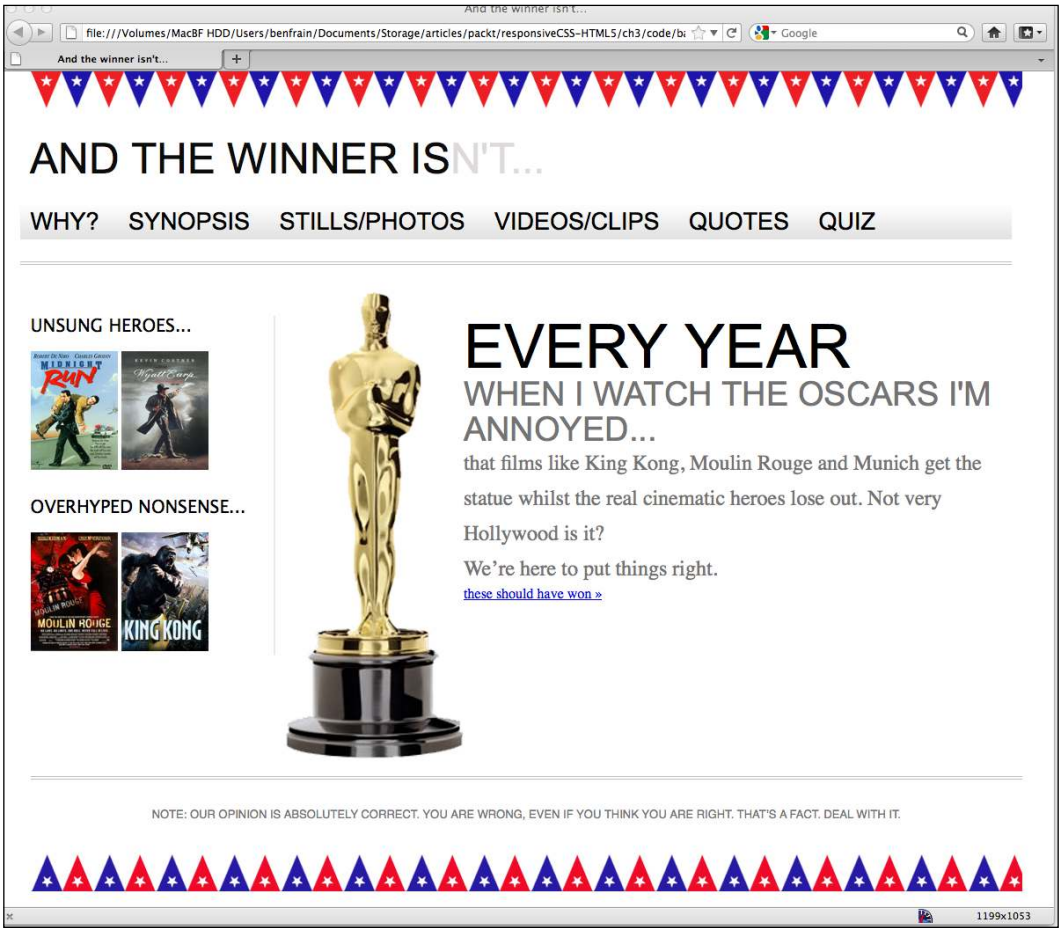
We could instead amend the CSS for the `<li>` tags, changing `inline-block` to be simply `inline`:

```
#navigation ul li {
  display: inline;
}
```

Opting for `display: inline;` (which stops the `<li>` elements behaving like block level elements), also makes the navigation render horizontally in earlier versions of Internet Explorer (versions 6 and 7) that have problems with `inline-block`. However, I'm a fan of `inline-block` as it gives greater control over the margins and padding for modern browsers so instead I'm going to leave the `<li>` tags as inline-blocks (and perhaps add an override style for IE 6 and IE 7, later) and instead move my percentage based margin rule from the `<a>` tag (which has no explicit context) to the containing `<li>` block instead. Here's what the amended rules now look like:

```
#navigation ul li {
  display: inline-block;
  margin-right: 2.6595745%; /* 25 ÷ 940 */
}

#navigation ul li a {
  height: 42px;
  line-height: 42px;
  text-decoration: none;
  text-transform: uppercase;
  font-family: Arial, "Lucida Grande", Verdana, sans-serif;
  font-size: 27px;
  color: black;
}
```

And the following screenshot shows how it looks in the browser with a 1200 px wide viewport:



So the navigation is getting there now, but I still have the problem of the navigation links spanning two lines as the viewport gets smaller, right until I get below 768 px wide when the media query we wrote in *Chapter 2, Media Queries: Supporting Differing Viewports*, then overrides the current navigation styles. Before we start fixing the navigation I'm going to switch all my typography sizes from fixed size pixels to the proportional unit, "ems". Once that's done we'll look at the other elephant in the room, getting our images to scale with the design.

Using ems rather than pixels for typography

In years gone by, web designers primarily used ems for sizing typography, rather than pixels, because earlier versions of Internet Explorer were unable to zoom text set in pixels. For some time, modern browsers have been able to zoom text on screen, even if the size values of the text were declared in pixels. So, why is using ems instead of pixels required or preferable? Here are two obvious reasons: firstly anyone still using Internet Explorer 6 (yes, those two) automatically gets the ability to zoom the text and secondly it makes life for you, the designer/developer, much easier. The size of an em is in relation to the size of its context. If we set a font size of 100 percent to our `<body>` tag and style all further typography using ems, they will all be affected by that initial declaration. The upshot of this being that if, having completed all the necessary typesetting, a client asks for all our fonts to be a little bigger we can merely change the body font size and all other typography changes in proportion.

Using our same *target* ÷ *context* = *result* formula, I'm going to convert every pixel based font size to ems. It's worth knowing that all modern desktop browsers use 16 px as the default font size (unless explicitly stated otherwise). Therefore, from the outset, applying any of the following rules to the body tag will provide the same result:

```
font-size: 100%;
font-size: 16px;
font-size: 1em;
```

As an example, the first pixel-based font size in our stylesheet controls the site's title, **AND THE WINNER ISN'T…** at top-left:

```
#logo {
  display: block;
  padding-top: 75px;
  color: #0d0c0c;
  text-transform: uppercase;
  font-family: Arial, "Lucida Grande", Verdana, sans-serif;
  font-size: 48px;
}

#logo span { color: #dfdada; }
```

Therefore, 48 ÷ 16 = 3. So our style changes to the following:

```
#logo {
  display: block;
  padding-top: 75px;
  color: #0d0c0c;
  text-transform: uppercase;
  font-family: Arial, "Lucida Grande", Verdana, sans-serif;
  font-size: 3em; /* 48 ÷ 16 = 3*/
}
```

You can apply this same logic throughout. If at any point things go haywire, it's probable the context for your target has changed. For example, consider the `<h1>` within the markup of our page:

```
<h1>Every year <span>when I watch the Oscars I'm annoyed...</span></
h1>
```

Our new em-based CSS looks like this:

```
#content h1 {
  font-family: Arial, Helvetica, Verdana, sans-serif;
  text-transform: uppercase;
  font-size: 4.3125em; } /* 69 ÷ 16 */

#content h1 span {
  display: block;
  line-height: 1.052631579em; /* 40 ÷ 38 */
  color: #757474;
  font-size: .550724638em; /* 38 ÷ 69 */
}
```

You can see here that the font size (which was 38 px) of the `<span>` element is in relation to the parent element (which was 69 px). Furthermore, the line-height (which was 40 px) is set in relation to the font itself (which was 38 px).

What on earth is an em?

The term **em** is simply a way of expressing the letter "M" in written form and is pronounced as such. Historically, the letter "M" was used to establish the size of a given font due to the letter "M" being the largest (widest) of the letters. Nowadays, em as a measurement defines the proportion of a given letter's width and height with respect to the point size of a given font.

So our structure is now resizing and we've switched our pixel-based type to ems. However, we still have to figure out how to scale images as the viewport resizes so let's look at that now.

Fluid images

Making images scale with a fluid layout can be achieved simply for modern browsers (including IE 7+). It's as simple as declaring the following in the CSS:

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

This makes any images automatically scale to up to 100 percent of their containing element. Furthermore, the same attribute and property can be applied to other media. For example:

```
img,object,video,embed {
  max-width: 100%;
}
```

And they will scale too, apart from a few notable exceptions such as `<iframe>` videos from YouTube but we'll wrestle those into submission in *Chapter 4*, *HTML5 for Responsive Designs*. For now though, we'll concentrate on images as the principles are the same, regardless of the media.

There are some important considerations in using this approach. Firstly, it requires some forward planning—the images inserted must be large enough to scale to larger viewport sizes. This leads to a further, perhaps more important consideration. No matter the viewport size or device viewing the site, they will still have to download the large images, even though on certain devices the viewport may only need to see an image 25 percent of its actual size. This is an important bandwidth consideration in some instances so we'll revisit this second problem shortly. For now, let's just get our images scaling.

Making images scale with the viewport

Consider our sidebar with the posters of two cracking movies and two absolute stinkers (this isn't up for discussion). The markup is currently as follows:

```
<!-- the sidebar -->
  <div id="sidebar">
    <div class="sideBlock unSung">
```

```
      <h4>Unsung heroes...</h4>
      <a href="#"><img src="img/midnightRun.jpg" alt="Midnight Run"
width="99" height="135" /></a>
      <a href="#"><img src="img/wyattEarp.jpg" alt="Wyatt Earp"
width="99" height="135" /></a>
    </div>
    <div class="sideBlock overHyped">
      <h4>Overhyped nonsense...</h4>
      <a href="#"><img src="img/moulinRouge.jpg" alt="Moulin Rouge"
width="99" height="135" /></a>
      <a href="#"><img src="img/kingKong.jpg" alt="King Kong"
width="99" height="135" /></a>
    </div>
  </div>
```

Although I've added the `max-width: 100%` declaration to the `img` element in my CSS, nothing has changed and the images aren't scaling as I expand the viewport:

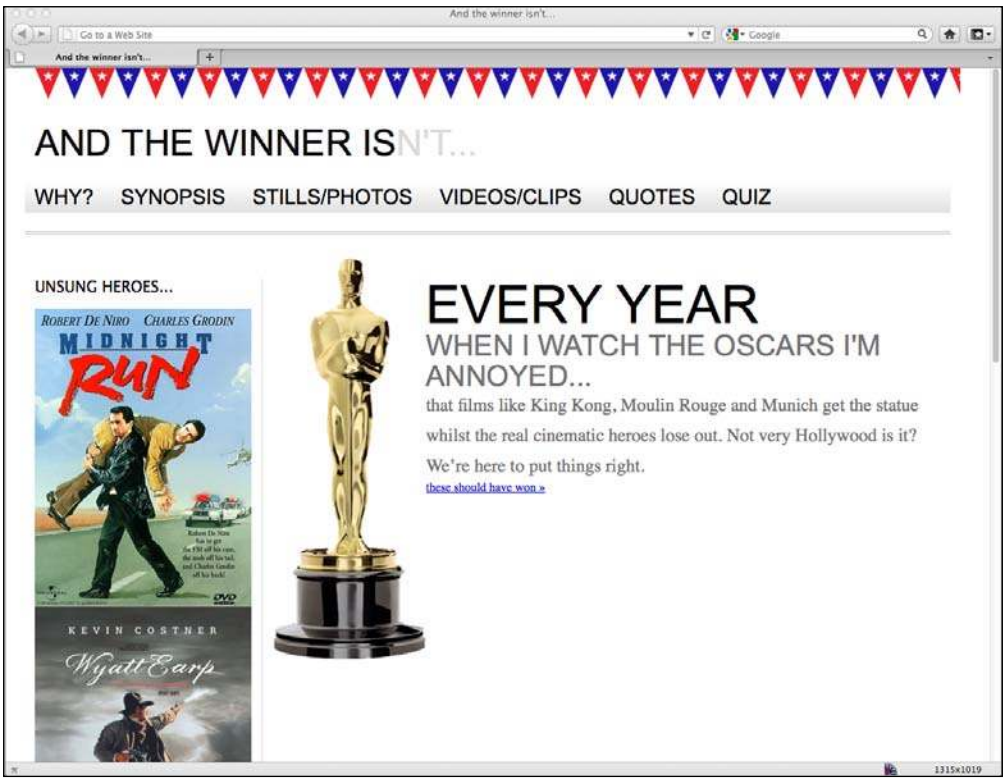
The reason here is that I've explicitly stated both the width and height of my images in the markup:

```
<img src="img/wyattEarp.jpg" alt="Wyatt Earp" width="99" height="135"
/>
```

Another schoolboy error! So I'll amend the markup associated with the images, removing the height and width attributes:

```
<img src="img/wyattEarp.jpg" alt="Wyatt Earp" />
```

Let's see what that does for us by refreshing the browser window:



Well, that's certainly working! But that's introduced a further problem. Because the images are scaling to fill up to 100 percent of the width of their containing element, they're each filling the sidebar. As ever, there are various ways to fix this…

Specific rules for specific images

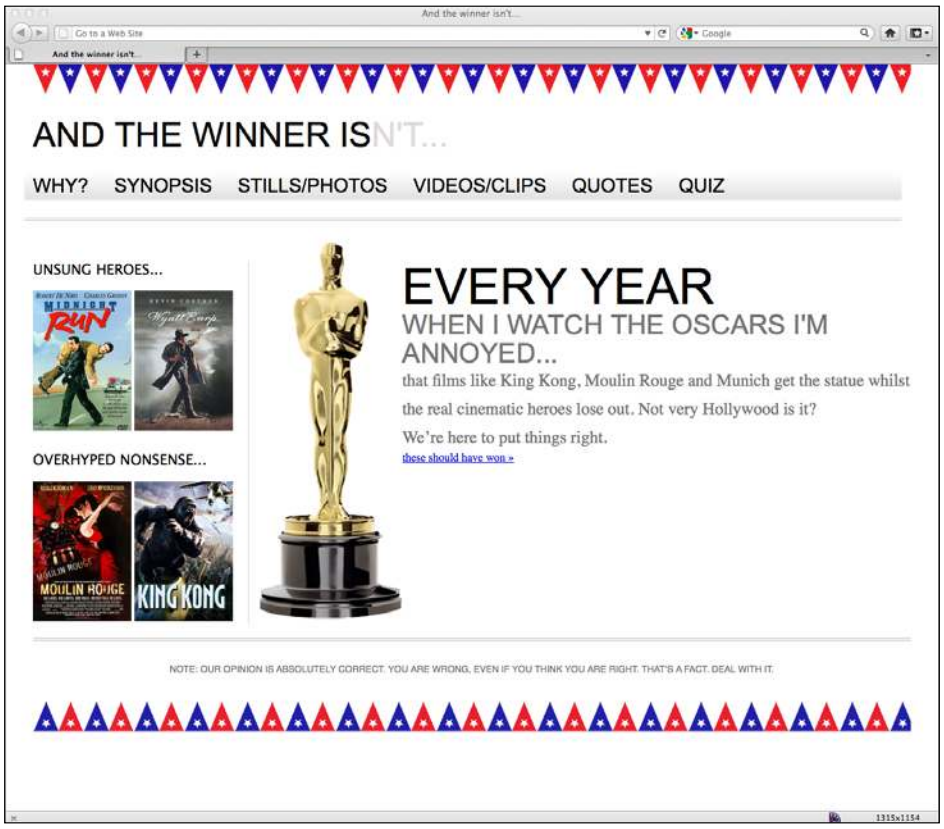
I could add an additional class to each image as done in the following code snippet:

```
<img class="sideImage" src="img/wyattEarp.jpg" alt="Wyatt Earp" />
```

And then set a specific rule for the width. However, instead I'm going to leave the markup as is and use CSS specificity to overrule the existing max-width rule with a further, more specific rule for my sidebar images:

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

.sideBlock img {
  max-width: 45%;
}
```

The following screenshot shows how things look in the browser now:



Using CSS specificity in this way allows us to add additional control to the width of any other images or media, too. Also, in *Chapter 5*, *CSS3: Selectors, Typography, and Color Modes* we'll look at how CSS3's powerful new selectors let us target almost any element without the need for extra markup or introducing JavaScript frameworks such as jQuery to do our dirty work.

For the sidebar images I decided on a width of 45 percent simply because I know that I need to add a little margin between the images later, and so having two images totaling 90 percent of the width gives me a little room (10 percent) to play with.

Now that the sidebar images are working, I'll also remove the width and height attributes on the Oscar statue image in the markup. However, unless I set a proportional width value for it, it's not going to scale so I've tweaked the associated CSS to set a proportional width using good ol' trusty *target* ÷ *context* = *result*.

```
.oscarMain {
  float: left;
  margin-top: -28px;
  width: 28.9398281%; /* 698 ÷ 202 */
}
```

Putting the brakes on fluid images

So now the images are scaling nicely as the viewport expands and contracts. However, if by expanding the viewport the image scales beyond its native size, things get very ugly. Take a look at Oscar in the following screenshot, with the viewport up to 1900 px:

The `oscar.png` image is actually 202 px wide. However, with the viewport over 1900 px wide and the image scaling to fit, it's actually displaying over 300 px wide. We can easily "put the brakes on" this image by setting another more specific rule:

```
.oscarMain {
  float: left;
  margin-top: -28px;
  width: 28.9398281%; /* 698 ÷ 202 */
  max-width: 202px;
}
```

That would let the `oscar.png` image scale because of the more general image rule but never go beyond the more specific max-width property set above. Here's how the page looks with this rule set:

The incredibly versatile max-width property

Another tack to limit things expanding limitlessly would be to set a `max-width` property on our entire `#wrapper` div like this:

```
#wrapper {
  margin-right: auto;
  margin-left: auto;
  width: 96%; /* Holding outermost DIV */
  max-width: 1414px;
}
```

This means the design will scale to 96 percent of the viewport but will never expand beyond 1414 px wide (I settled on 1414 px as on most modern browsers it cuts the bunting flags off at the end of a flag rather than halfway through one). The following screenshot shows how it looks like with a viewport of around 1900 px:

Obviously these are merely options. It, however, proves the versatility of a fluid grid and how we can control the flow with just a few specific declarations.

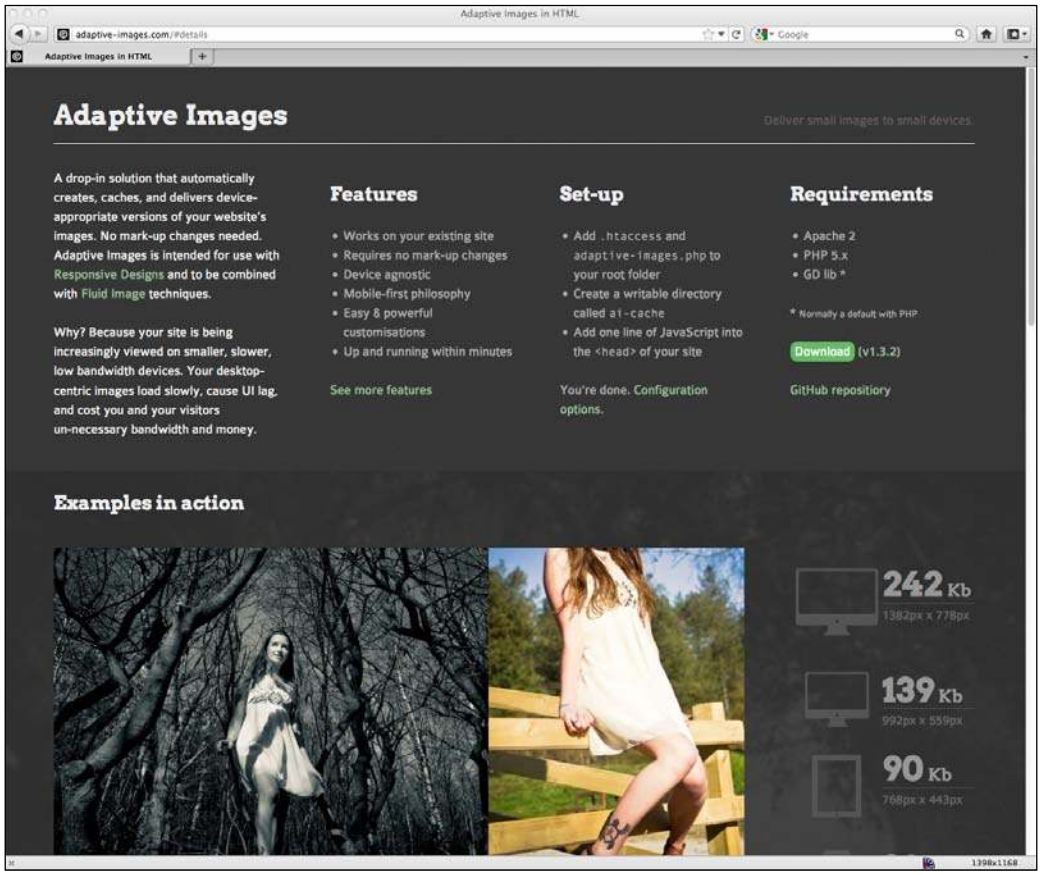
Serving different images for different screen sizes

We have our images resizing nicely and we now understand how we can limit the display size of specific images should we choose to. However, earlier in the chapter we noted the inherent problem with scaling images. They must be physically larger than they are displayed in order to render well. If they aren't, they start to look a mess. Because of this, images, in terms of file size, are almost always bigger than they need to be given the likely display size.

Various people have tackled the problem, attempting to provide smaller images to smaller screens. The first notable example was the Filament Group's "Responsive Images" (`http://filamentgroup.com/lab/responsive_images_experimenting_with_context_aware_image_sizing/`). However, recently, I've switched to Matt Wilcox's "Adaptive Images" (`http://adaptive-images.com`). The Filament Group's solution required the image related markup to be altered. Matt's solution doesn't and automatically creates the (smaller) resized images based on the full size image already specified in the markup. This solution therefore allows images to be resized and served to the user as needed based upon a number of screen size break points. Let's jump in and get Adaptive Images up and running.

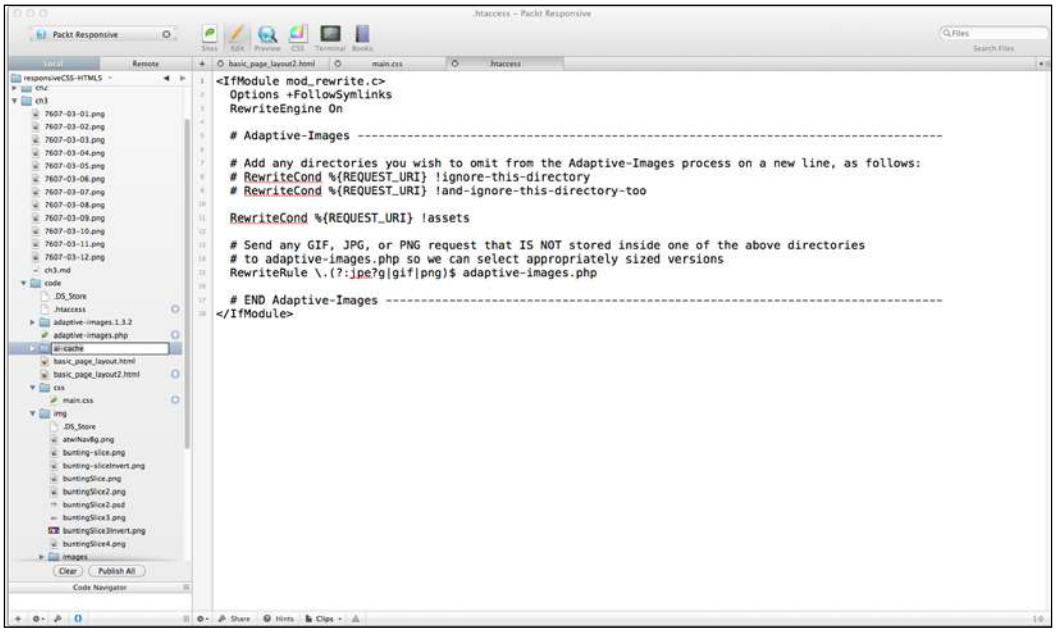
Setting up Adaptive Images

The Adaptive Images solution requires Apache 2, PHP 5.x, and GD Lib. So you'll need to be developing on an appropriate server to see the benefits. So, go ahead, download the `.zip` file and let's get started:



Extract the content of the ZIP file and copy the `adaptive-images.php` and `.htaccess` files into the root directory of your site. If you are already using an `.htaccess` file in your site's root directory, do not overwrite it. Instead read the additional information in the `instructions.htm` file included in the download.

Now create a folder in the root of your site called **ai-cache**.



Use your favourite FTP client to set write permissions of `777`.

Now copy the following JavaScript into the `<head>` tag of each page that needs adaptive images:

```
<script>document.cookie='resolution='+Math.max(screen.width,screen.
height)+'; path=/';</script>
```

Note that if you're not using HTML5 (we'll be changing to HTML5 in the next chapter), if you want the page to validate, you'll need to add the `type` attribute. So the script should be as follows:

```
<script type="text/javascript">document.cookie='resolution='+Math.
max(screen.width,screen.height)+'; path=/';</script>
```

It's important that the JavaScript is in the head (preferably the first piece of script) because it needs to work before the page has finished loading, and before any images have been requested. Here it is added to the `<head>` section of our site in progress:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<meta name="viewport"  content="width=device-width,initial-scale=1.0"
/>
<title>And the winner isn't…</title>
<script type="text/javascript">document.cookie='resolution='+Math.
max(screen.width,screen.height)+'; path=/';</script>
<link href="css/main.css" rel="stylesheet" type="text/css" />
</head>
```

Put background images somewhere else

In the past, I've typically placed all my images (both those used for background CSS elements and inline images inserted in the markup) in a single folder such as `images` or `img`. However, if using Adaptive Images, it's advisable that images to be used with CSS as background images (or any other images you don't want to be re-sized) be placed in a different directory. Adaptive Images by default defines a folder called `assets` to keep images you don't want resizing within. Therefore, if you want any images left alone, keep them there. If you'd like to use a different folder (or more than one) you can amend the `.htaccess` file as follows:

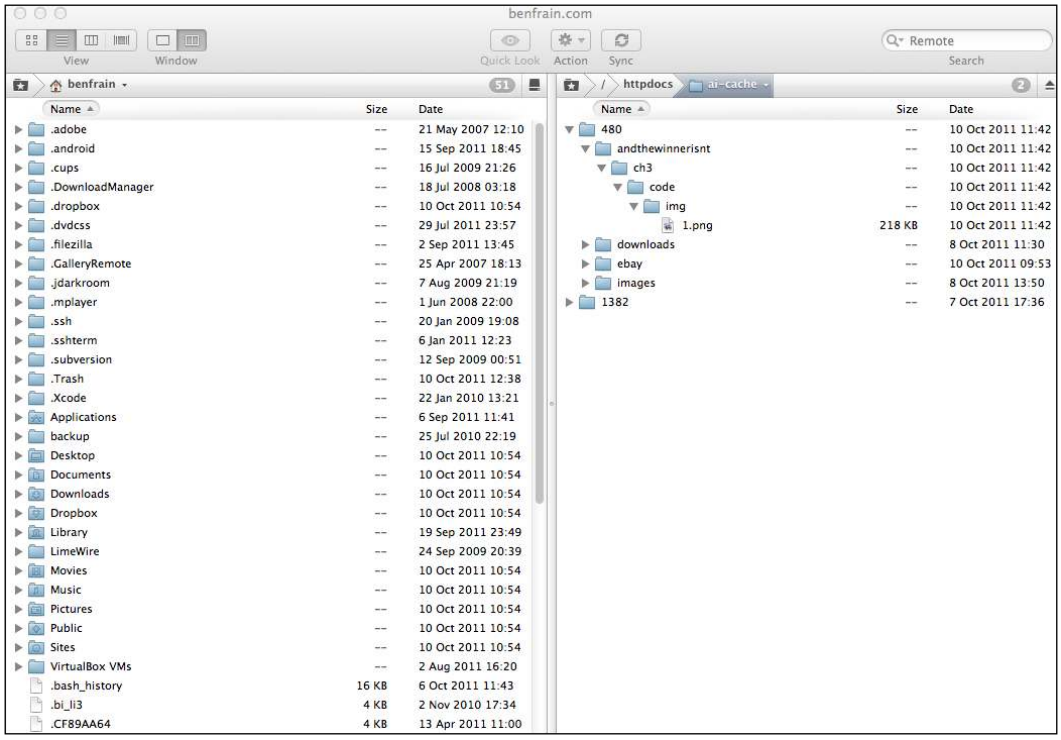
```
<IfModule mod_rewrite.c>
  Options +FollowSymlinks
  RewriteEngine On
```

```
  # Adaptive-Images ----------------------------------------------
--------------------------------

  RewriteCond %{REQUEST_URI} !assets
  RewriteCond %{REQUEST_URI} !bkg

  # Send any GIF, JPG, or PNG request that IS NOT stored inside one of
the above directories
  # to adaptive-images.php so we can select appropriately sized
versions
  RewriteRule \.(?:jpe?g|gif|png)$ adaptive-images.php

  # END Adaptive-Images ------------------------------------------
--------------------------------
</IfModule>
```

In this example, we have specified that we don't want images within `assets` or `bkg` adapting. Conversely, should you wish to explicitly state that you only want images within certain folders to be adapted, you can omit the exclamation mark from the rule. For example, if I only wanted images in a subfolder of my site, called `andthewinnerisnt`, I would edit the `.htaccess` file as follows:

```
<IfModule mod_rewrite.c>
  Options +FollowSymlinks
  RewriteEngine On

  # Adaptive-Images ----------------------------------------------
--------------------------------

  RewriteCond %{REQUEST_URI} andthewinnerisnt

  # Send any GIF, JPG, or PNG request that IS NOT stored inside one of
the above directories
  # to adaptive-images.php so we can select appropriately sized
versions
  RewriteRule \.(?:jpe?g|gif|png)$ adaptive-images.php

  # END Adaptive-Images ------------------------------------------
--------------------------------
</IfModule>
```

That is all there is to it. The easiest way to check that it's up and running is to insert a large image into a page, and then visit the page with a smart phone. If you check the contents of your `ai-cache` folder with an FTP program you should see files and folders within named breakpoint folders, for example, `480` (see the following screenshot):



Adaptive Images isn't restricted to static sites. It can also be used alongside Content Management Systems and there are also workarounds for when JavaScript is unavailable. With Adaptive Images, there is a way to serve entirely different images based upon screen size, saving bandwidth overheads for devices that wouldn't see the benefit of the default full size images.

Where fluid grids and media queries come together

If you remember, earlier in the chapter, our navigation links were still spanning multiple lines at certain viewport widths. We can fix this problem with media queries. If our links fall apart at 1060 px and work again at 768 px (where our earlier media query takes over), let's set some additional font styles for the ranges in-between:

```
@media screen and (min-width: 1001px) and (max-width: 1080px) {
#navigation ul li a { font-size: 1.4em; }
}
@media screen and (min-width: 805px) and (max-width: 1000px) {
  #navigation ul li a { font-size: 1.25em; }
}
@media screen and (min-width: 769px) and (max-width: 804px) {
  #navigation ul li a { font-size: 1.1em; }
}
```

As you can see, we're changing the font size based upon the viewport width and the result is a set of navigation links that always sit on one line, throughout the range of 769 px to infinity. Evidence again of the symbiosis between media queries and fluid layouts—media queries limit the shortfalls of a fluid layout and a fluid layout eases the change from one set of defined styles within a media query to another.

CSS Grid systems

CSS Grid systems/frameworks are a potentially divisive subject. Some designers swear by them, others swear at them. In a bid to minimize hate mail, I'm going to say I sit entirely on the fence. Whilst I can understand why some developers think they are superfluous and in certain instances create extraneous code, I can also appreciate their value for rapidly prototyping layouts.

Here are a few CSS frameworks that offer varying degrees of "responsive" support:

- Semantic (`http://semantic.gs`)
- Skeleton (`http://getskeleton.com`)
- Less Framework (`http://lessframework.com`)
- 1140 CSS Grid (`http://cssgrid.net`)
- Columnal (`http://www.columnal.com`)

Of these, I personally favor the Columnal grid system as it has a fluid grid built-in alongside media queries and also uses similar CSS classes as `960.gs`, the popular fixed-width grid system that most developers and designers are familiar with.

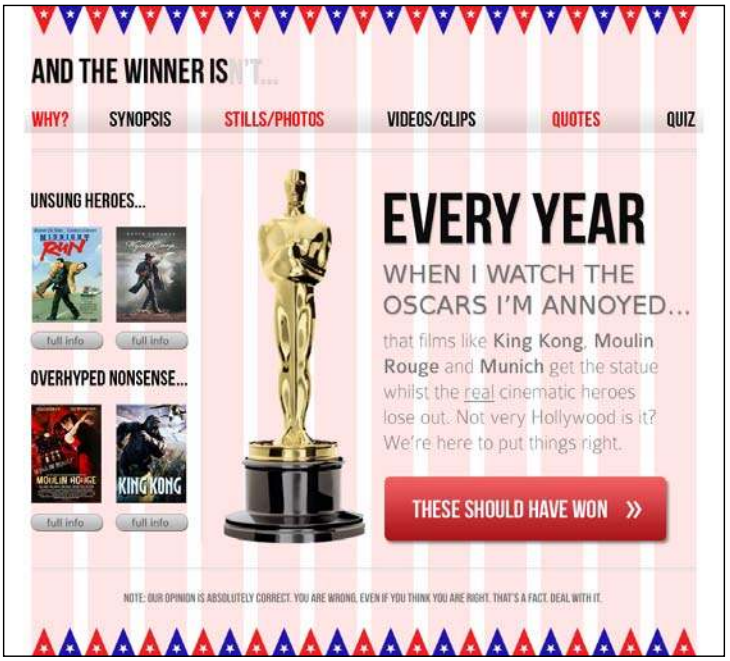
Alpha, Omega, and other common grid classes

Many CSS grid systems use specific CSS classes to perform everyday layout tasks. The `row` and `container` classes are self-explanatory but there are often many more. Therefore, always check any grid system's documentation for any other classes that will make life easier. For example, other typical de facto classes used in CSS Grid systems are `alpha` and `omega`—for the first and last items in a row respectively (the `alpha` and `omega` classes remove padding or margin) and `.col_x` where *x* is the number for the amount of columns the item should span (for example, `col_6` for six columns).

Rapidly building our site with a Grid system

Let's suppose we hadn't already built our fluid grid, nor had we written any media queries. We're handed the original *And the winner isn't...* homepage composite PSD and told to get the basic layout structure up and running in HTML and CSS as quickly as possible. Let's see if the Columnal grid system will help us achieve that goal.

In our original PSD, it was easy to see the layout was based on 16 columns. The Columnal grid system however only supports up to 12 columns so let's overlay 12 columns over the PSD instead of the original 16:



Having downloaded Columnal and extracted the contents of the ZIP file, we'll duplicate the existing page and then link to `columnal.css` rather than `main.css` in the `<head>`. To create visual structure using Columnal, the key is in adding the correct div classes in the markup. Here is the full markup of the page up to this point:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<meta name="viewport"  content="width=device-width,initial-scale=1.0"
/>
<title>And the winner isn't…</title>
<script type="text/javascript">document.cookie='resolution='+Math.
max(screen.width,screen.height)+'; path=/';</script>
<link href="css/columnal.css" rel="stylesheet" type="text/css" />

</head>

<body>

<div id="wrapper">
  <!-- the header and navigation -->
  <div id="header">
    <div id="logo">And the winner is<span>n't...</span></div>
    <div id="navigation">
      <ul>
        <li><a href="#">Why?</a></li>
        <li><a href="#">Synopsis</a></li>
        <li><a href="#">Stills/Photos</a></li>
        <li><a href="#">Videos/clips</a></li>
        <li><a href="#">Quotes</a></li>
        <li><a href="#">Quiz</a></li>
      </ul>
    </div>
  </div>
  <!-- the content -->
  <div id="content">
    <img class="oscarMain" src="img/oscar.png" alt="atwi_oscar" />
    <h1>Every year <span>when I watch the Oscars I'm annoyed...</
span></h1>
    <p>that films like King Kong, Moulin Rouge and Munich get the
statue whilst the real cinematic heroes lose out. Not very Hollywood
is it?</p>
```

```
<p>We're here to put things right. </p>
  <a href="#">these should have won &raquo;</a>
  </div>
  <!-- the sidebar -->
  <div id="sidebar">
    <div class="sideBlock unSung">
      <h4>Unsung heroes...</h4>
      <a href="#"><img src="img/midnightRun.jpg" alt="Midnight Run"
/></a>
      <a href="#"><img class="sideImage" src="img/wyattEarp.jpg"
alt="Wyatt Earp" /></a>
    </div>
    <div class="sideBlock overHyped">
      <h4>Overhyped nonsense...</h4>
      <a href="#"><img src="img/moulinRouge.jpg" alt="Moulin Rouge"
/></a>
      <a href="#"><img src="img/kingKong.jpg" alt="King Kong" /></a>
    </div>
  </div>
  <!-- the footer -->
  <div id="footer">
    <p>Note: our opinion is absolutely correct. You are wrong, even if
you think you are right. That's a fact. Deal with it.</p>
  </div>

</div>
</body>
</html>
```

First of all, we need to specify that our `#wrapper` div is the container for all elements so we'll add the `.container` class to it:

```
<div id="wrapper" class="container">
```

Working down the page we can see that our **AND THE WINNER ISN'T** text is the first row. Therefore, we'll add the `.row` class to that element:

```
<div id="header" class="row">
```

Our logo, although just text, sits within this row and spans the entire 12 columns. Therefore we'll add `.col_12` to it:

```
<div id="logo" class="col_12">And the winner is<span>n't...</span></
div>
```

Then the navigation is the next row so we'll add a `.row` class to that:

```
<div id="navigation" class="row">
```

And on the process goes, adding `.row` and `.col_x` classes as necessary. We'll jump ahead at this point, as I'm concerned the repetition of this process may have you nodding off. Instead, here is the entire amended markup. Note, it was also necessary to move the Oscar image and set it in its own column. Plus add a wrapping `.row` div around our `#content` and `#sidebar`.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<meta name="viewport"  content="width=device-width,initial-scale=1.0"
/>
<title>And the winner isn't…</title>
<script type="text/javascript">document.cookie='resolution='+Math.
max(screen.width,screen.height)+'; path=/';</script>
<link href="css/columnal.css" rel="stylesheet" type="text/css" />
<link href="css/custom.css" rel="stylesheet" type="text/css" />

</head>

<body>

<div id="wrapper" class="container">
  <!-- the header and navigation -->
  <div id="header" class="row">
    <div id="logo" class="col_12">And the winner is<span>n't...</
span></div>
    <div id="navigation" class="row">
      <ul>
        <li><a href="#">Why?</a></li>
        <li><a href="#">Synopsis</a></li>
        <li><a href="#">Stills/Photos</a></li>
        <li><a href="#">Videos/clips</a></li>
        <li><a href="#">Quotes</a></li>
        <li><a href="#">Quiz</a></li>
      </ul>
    </div>
  </div>
  <div class="row">
    <!-- the content -->
    <div id="content" class="col_9 alpha omega">
      <img class="oscarMain col_3" src="img/oscar.png" alt="atwi_
oscar" />
      <div class="col_6 omega">
      <h1>Every year <span>when I watch the Oscars I'm annoyed...</
span></h1>
```

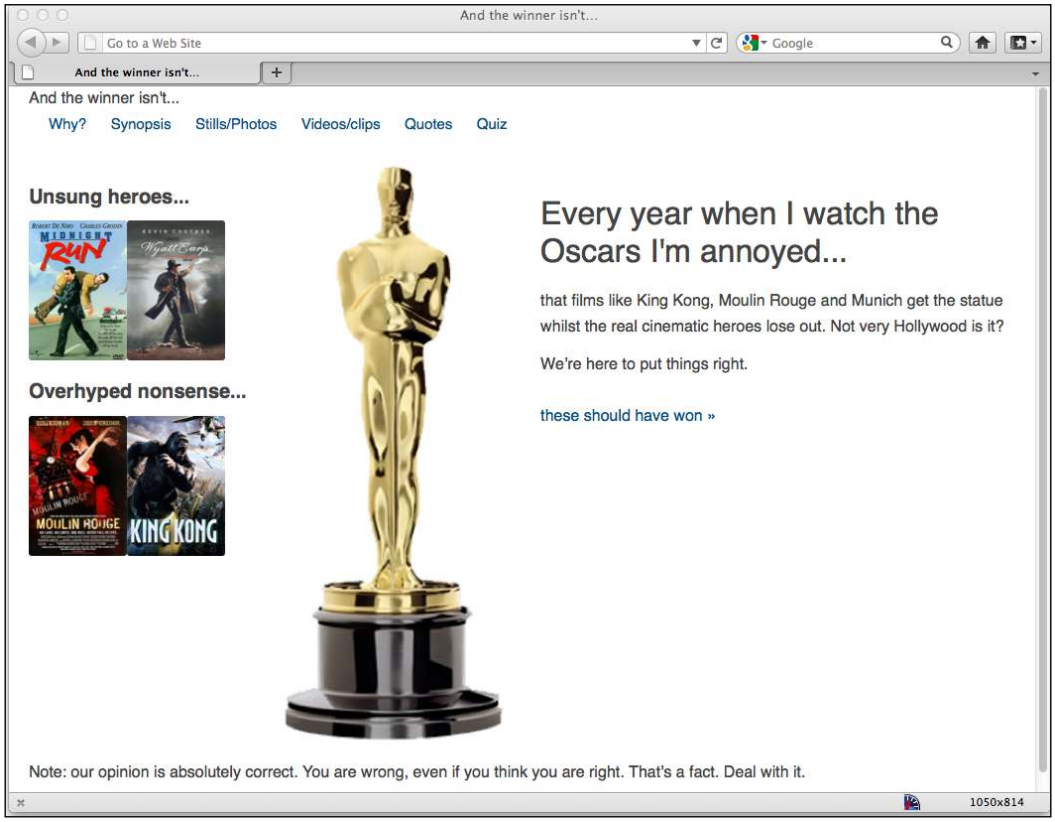
```
      <p>that films like King Kong, Moulin Rouge and Munich get the
statue whilst the real cinematic heroes lose out. Not very Hollywood
is it?</p>
  <p>We're here to put things right. </p>
    <a href="#">these should have won &raquo;</a>
    </div>
    </div>
    <!-- the sidebar -->
    <div id="sidebar" class="col_3">
      <div class="sideBlock unSung">
        <h4>Unsung heroes...</h4>
        <a href="#"><img src="img/midnightRun.jpg" alt="Midnight Run"
/></a>
        <a href="#"><img class="sideImage" src="img/wyattEarp.jpg"
alt="Wyatt Earp" /></a>
      </div>
      <div class="sideBlock overHyped">
        <h4>Overhyped nonsense...</h4>
        <a href="#"><img src="img/moulinRouge.jpg" alt="Moulin Rouge"
/></a>
        <a href="#"><img src="img/kingKong.jpg" alt="King Kong" /></a>
      </div>
    </div>
  </div>
  <!-- the footer -->
  <div id="footer" class="row">
    <p>Note: our opinion is absolutely correct. You are wrong, even if
you think you are right. That's a fact. Deal with it.</p>
  </div>

</div>
</body>
</html>
```

It was also necessary to add some extra CSS styles into a `custom.css` file. The content of this file is as follows:

```
#navigation ul li {
  display: inline-block;
}

#content {
  float: right;
}

#sidebar {
  float: left;
}

.sideBlock {
```

```
    width: 100%;
}

.sideBlock img {
    max-width: 45%;
    float:left;
}

.footer {
    float: left;
}
```

With these basic changes done, a quick look in the browser window shows that our basic structure is in place and scales with the browser viewport:



There's obviously a lot of detail work to still be done (I know, that's more than a slight understatement) but if you need a fast way of creating a basic responsive structure, CSS Grid systems such as Columnal are worthy of consideration.

Summary

In this chapter, we've learned how to change a rigid pixel-based structure to a flexible percentage-based one. We've also learned how to use ems, rather than pixels for more flexible typesetting. We now also understand how we can make images respond and resize fluidly as well as implementing a server-based solution for serving entirely different images based upon device screen size. Finally, we've experimented with a responsive CSS Grid system that allows us to rapidly prototype responsive structures with very minimal effort.

However, until this point we've been pursuing our responsive quest using HTML 4.01 for our markup. In *Chapter 1*, *Getting Started with HTML5, CSS3, and Responsive Web Design*, we touched upon some of the economies that HTML5 offers us. These economies are particularly important and relevant for responsive designs where a "mobile first" mindset lends itself to the leanest, fastest, and most semantic code possible. In the next chapter, we're going to get to grips with HTML5 and modify our markup to take advantage of the latest and greatest iteration of the HTML specification.