

# 5

## CSS3: Selectors, Typography, and Color Modes

In *Chapter 1, Getting Started with HTML5, CSS3, and Responsive Web Design*, we noted that the number of people viewing websites over mobile telecom networks is ever increasing. As current telecom network speeds vary enormously, we need to consider the bandwidth and therefore load time of the websites we build. Back in the day we had to consider how long our pages and the images and media they contained would take to load over a 56K modem. Now, we face similar loading time challenges. Just as the percentage rules of table-based layouts are re-emerging, so is the need to re-examine every piece of media and bandwidth sapping content we add to our pages. Although our devices are now mobile, the speeds they download content and the premium they face for doing so (speed and cost) is comparable to years gone by. Everything old is new again! Thankfully, CSS3 can heavily reduce our reliance on images for visual flair giving us the tools to create beautiful sites that also download in record time. There's lots of CSS3 for us to cover. *Chapter 6, Stunning Aesthetics with CSS3*, will deal with more specific CSS3 techniques including text shadows, box shadows, gradients, and backgrounds whilst *Chapter 7, CSS3 Transitions, Transformations, and Animations*, will look at CSS3 animations, transforms, and transitions.

In this chapter, we will learn the following CSS3 fundamental:

- What CSS3 offers the frontend developer
- Quick and handy CSS3 tricks (multiple columns and word wraps)
- The anatomy of a CSS rule
- What vendor-specific prefixes are and how to use them
- New CSS3 selectors and how they work
- Custom typography with `@font-face`
- How to use RGB and HSL color modes with Alpha transparency

## What CSS3 offers the frontend developer

In the past, we either gambled that users would put up with long load times for the sake of a great design (they wouldn't, by the way!) or we ditched images, often compromising our design ideals, for the sake of usability. CSS3, in many ways negates the need for compromise. With just a few lines of code (and no images!) CSS3 can produce onscreen elements such as rounded corners, background gradients, text shadows, box shadows, custom typography, and multiple background images (alright, granted, that one does require images). If that wasn't enough, much of the basic interaction for which we have previously relied on JavaScript, such as hover state animations, can also be handled with pure CSS3. There are heaps of CSS3 goodies and economies that will elevate our responsive design from merely "a normal website made responsive" to a responsive website built for the future. By utilizing CSS3, we will enable our responsive design to load faster, require less resource and be far easier to maintain and amend in the future. Before we get into that, let's deal with the "Elephant in the room".

## CSS3 support in Internet Explorer versions 6 to 8

With a few exceptions (such as `@font-face`), few features of the new CSS3 modules are supported by Old IE (Internet Explorer versions 6, 7, and 8). Should you use CSS3 in your design? As ever in web development, the answer is "it depends".

Personally, at present, I principally use CSS3 to enhance a site, rather than provide essential functionality. I'm entirely comfortable with elements looking a little different in different browsers. I believe you and your clients should be too. You might find it helpful to refer back to the *Educating our clients that websites shouldn't look the same in all browsers* section in *Chapter 1, Getting Started with HTML5, CSS3, and Responsive Web Design*. Which parts of a design are critical to it "working" or "looking right" is subjective. But it's worth knowing that there are many polyfills available for adding CSS3 functionality to Old IE. Applying such polyfills, should you choose to follow that path, is discussed more in *Chapter 9, Solving Cross-browser Responsive Challenges*.



For a full list of what CSS 2.1 and CSS3 features are supported in the differing versions of Internet Explorer, head over to the following URL:  
<http://msdn.microsoft.com/en-us/library/cc351024%28v=vs.85%29.aspx>

---

## Using CSS3 to design and develop pages in the browser

I can't speak for you but I find re-making images tiresome. You know the kind of comment I'm talking about, "Could we make those corners a little rounder?" or "Can the gradient be a little darker at the top?" Once we've dutifully made the amends, we often hear the inevitable, "Oh, no, it was better the way it was. Can you swap it back?" Now, of course, this to-and-fro process is necessary; after all, we often want to tweak a design just to see how it looks. However, CSS3 lets you do much of this in mere seconds, within the code, rather than minutes within the graphics editor.

### Anatomy of a CSS rule

Before exploring some of what CSS3 has to offer, to prevent confusion, let's establish the terminology we use to describe a CSS rule. Consider the following example:

```
.round {
  border-radius: 10px;
}
```

This rule is made up of the **selector** (`.round`) and then the **declaration** (`border-radius: 10px;`). However, the declaration is further defined by the **property** (`border-radius:`) and the **value** (`10px;`). Happy, we're on the same page? Great, let's press on.

### Vendor prefixes and how to use them

As the CSS3 Modules specifications have yet to be either ratified by the W3C or have all their proposed features fully implemented into browsers, browser vendors use what's known as **vendor prefixes** to test new "experimental" CSS features. Whilst this helps browser makers implement the new CSS3 modules, it makes our lives, as writers of CSS3, just a little more tedious. Consider the following code for a rounded corner in CSS3:

```
.round{
  -khtml-border-radius: 10px; /* Konqueror */
  -rim-border-radius: 10px; /* RIM */
  -ms-border-radius: 10px; /* Microsoft */
  -o-border-radius: 10px; /* Opera */
  -moz-border-radius: 10px; /* Mozilla (e.g Firefox) */
  -webkit-border-radius: 10px; /* Webkit (e.g. Safari and Chrome) */
  border-radius: 10px; /* W3C */
}
```

You can see a number of vendor prefixed properties (and that is by no means an exhaustive list), each with their own unique prefix, for example, `-webkit-` for Webkit based browsers, `-ms-` is the Microsoft prefix, so covers the Internet Explorer, and so on. Due to the way CSS works, a browser will go line by line down the stylesheet, applying properties that apply to it and ignoring ones that don't.

Furthermore, applicable properties later in the stylesheet take precedence over earlier ones. Thanks to this cascade, we can list our vendor-prefixed properties first and then the correct (but perhaps yet to be implemented) non-prefix version last, safe in the knowledge that when the feature is fully implemented, the correct version will be implemented by the browser, rather than the experimental, browser-specific one listed before it.



#### Clippings and JavaScript for quick CSS3 prefixes

You may find it handy to keep clippings of common CSS3 rules containing all the necessary vendor prefixed properties. That way you can just paste them in without needing rewrite them all each time. Many code-editing programs (or **Integrated Development Environments (IDEs)** as they are often labeled) have code clip features and when using CSS3 they can save a lot of time. There's also JavaScript solutions that automatically add prefixes to CSS files, check out "-prefix-free", a great solution, at <http://leaverou.github.com/prefixfree/>.

It's acceptable to list every vendor prefix version of a property. However, in reality, few people do. Instead they either target the browsers they expect to see most often or check what browsers support the feature before writing the rule. For example, you might just opt to go with:

```
.round{
  -moz-border-radius: 10px; /* Mozilla (e.g Firefox) */
  -webkit-border-radius: 10px; /* Webkit (e.g. Safari and Chrome) */
  border-radius: 10px; /* W3C */
}
```

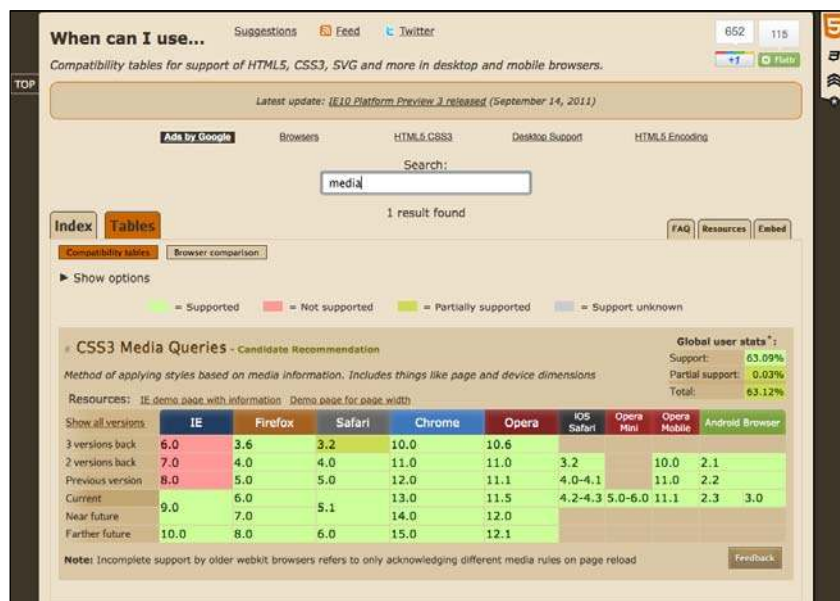
That would cover Firefox, Chrome, and Safari, along with any browser that has fully implemented the rule.

I know what you're thinking, isn't listing multiple vendor prefixed versions of the same property going to lead to code bloat? Well, a little yes. But no matter how many prefixed properties we add, it's still a faster, more elegant and robust solution than using images.

Before working on a site, it's wise to look at the current browser usage statistics. In doing so, you'll have a better idea of what browsers you need to build specific support for. For example, if time and budget are tight, you might decide to omit vendor specific prefixes for any browser with less than 3 percent usage rate for your site. As ever, you need to make a judgment based on a number of variables.

Now, we understand what the prefixes are and how to apply them in our rules. Let's look at some quick and useful little CSS3 tricks.

### When can I use specific CSS3 and HTML5 features?



The screenshot shows the 'When can I use...' page for 'CSS3 Media Queries'. It includes a search bar with 'media' entered, a table of browser support, and global usage statistics.

**Global user stats:**  
 Support: 63.09%  
 Partial support: 0.03%  
 Total: 63.12%

Show all versions	IE	Firefox	Safari	Chrome	Opera	iOS Safari	Opera Mini	Opera Mobile	Android Browser	
3 versions back	6.0	3.6	3.2	10.0	10.6					
2 versions back	7.0	4.0	4.0	11.0	11.0	3.2		10.0	2.1	
Previous version	8.0	5.0	5.0	12.0	11.1	4.0-4.1		11.0	2.2	
Current	9.0	6.0	5.1	13.0	11.5	4.2-4.3	5.0-6.0	11.1	2.3	3.0
Near future		7.0		14.0	12.0					
Farther future	10.0	8.0	6.0	15.0	12.1					

**Note:** Incomplete support by older webkit browsers refers to only acknowledging different media rules on page reload

As we delve into CSS3 more and more, I can heartily recommend visiting <http://caniuse.com>, if you ever want to know what the current level of browser support is available for a particular CSS3 or HTML5 feature. Alongside showing browser version support (searchable by feature) it also provides the most recent set of global usage statistics from <http://gs.statcounter.com>.

## Quick and useful CSS3 tricks

In my day-to-day work, some of the new CSS3 features I use constantly and others I've never needed. Before getting into the heavier stuff, I thought it might be useful to share a couple of CSS3 goodies that make life easier, especially in responsive designs, by accomplishing simple tasks that used to be minor headaches.

### CSS3 multiple columns for responsive designs

Ever needed to make a single piece of text appear in multiple columns? Until CSS3, you'd need to separate the content into different markup elements and then style accordingly. Altering markup for stylistic purposes is never a good practice. CSS3 allows us to span one or more pieces of content across multiple columns. Consider the following markup:

```
<div id="main" role="main">
  <p>lloremipsimLorem ipsum dolor sit amet, consectetur
  // LOTS MORE TEXT //
</p>
  <p>lloremipsimLorem ipsum dolor sit amet, consectetur
  // LOTS MORE TEXT //
</p>
</div>
```

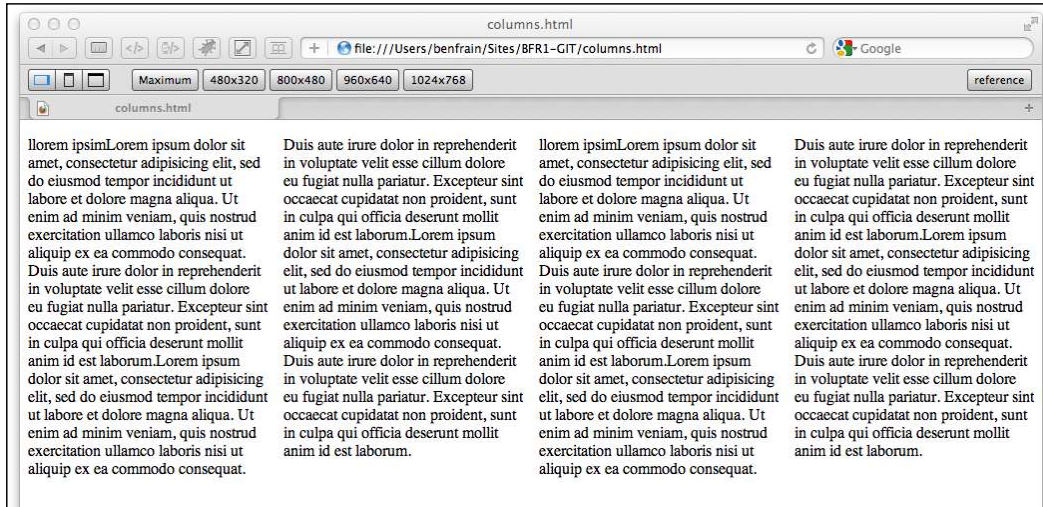
You can make all that content flow across multiple columns that are either: a certain column width (for example, 12em) or certain number of columns (for example, 3). Here's how:

For a certain width of column, use the following syntax (note that vendor prefixes have been omitted for brevity):

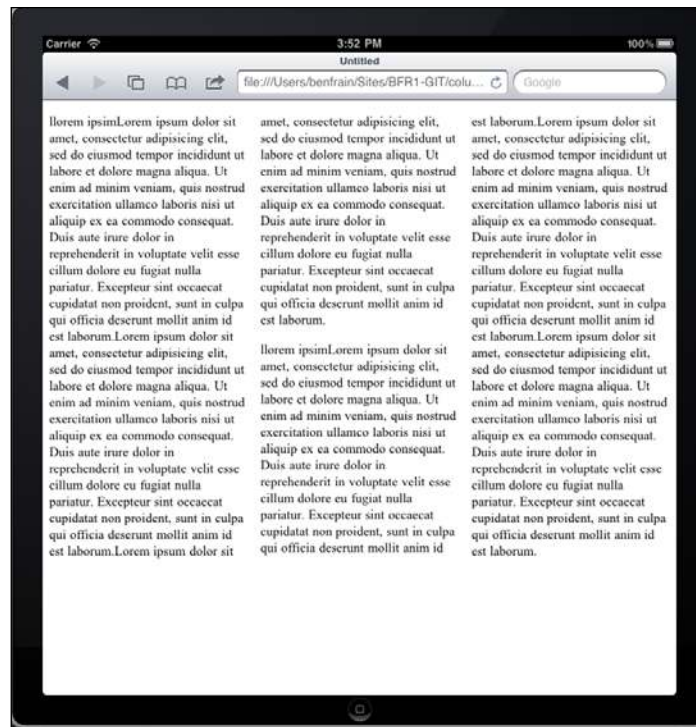
```
#main {
  column-width: 12em;
}
```

This will mean, no matter the viewport size, the content will span across columns that are 12 em in width. Altering the viewport will adjust the number of columns displayed dynamically.

For example, here it is in Safari with a 1024 px wide viewport:



And the following screenshot shows how the same page renders on an iPad with a 768 px wide viewport:



A beautifully responsive layout requiring the minimum of work—I like it!

If you'd rather keep a fixed number of columns and vary the width, you can write a rule like the following:

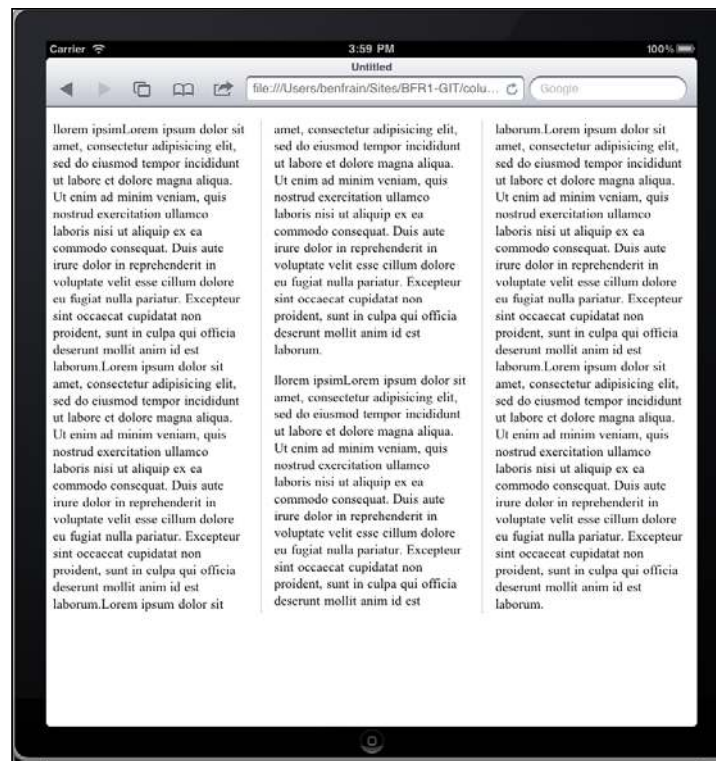
```
#main {
  column-count: 4;
}
```

## Adding a gap and column divider

We can take things even further by adding a specified gap for the columns and a divider:

```
#main {
  column-gap: 2em;
  column-rule: thin dotted #999;
  column-width: 12em;
}
```

This gives us a result like the following:



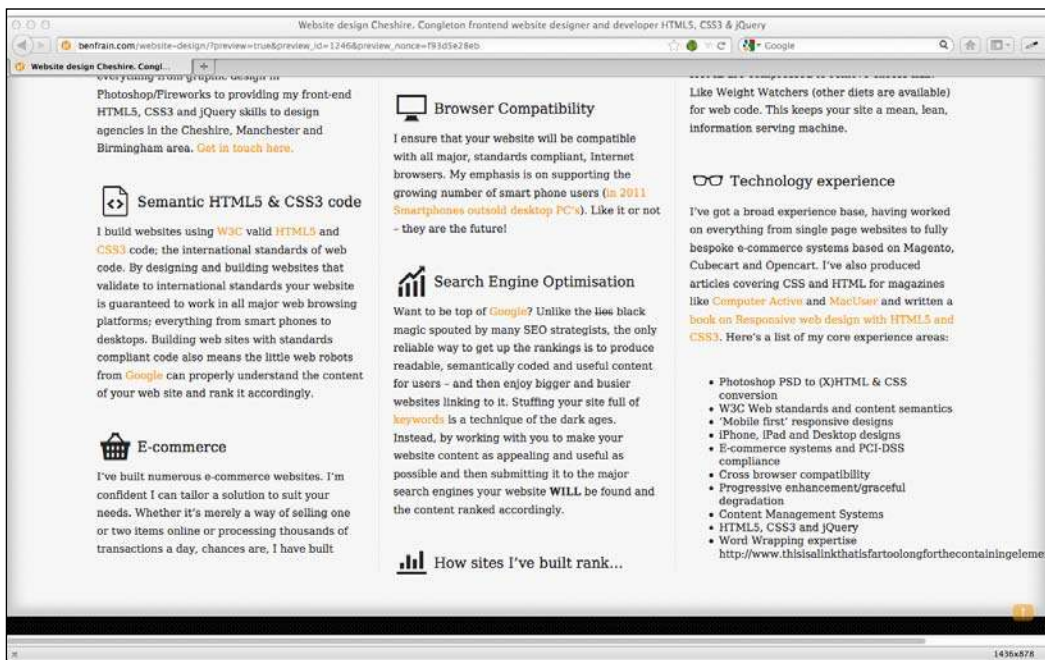


To read the specification on the CSS3 Multi-column Layout Module, visit <http://www.w3.org/TR/css3-multicol/>.

For the time being, remember you'll need to use vendor prefixes on the column declarations for maximum compatibility.

## Word wrapping

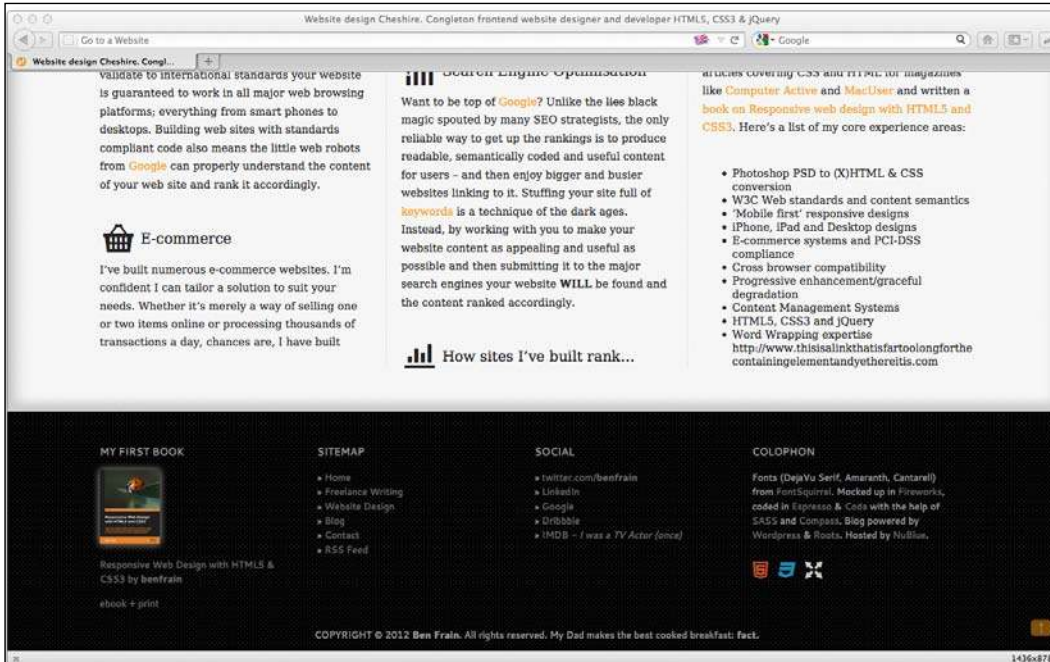
How many times have you had to add a big URL into a tiny space and, well, despaired? Take a look at the problem in the following screenshot; notice the URL at the bottom right breaking out of its allocated space:



CSS3 fixes this problem with a simple declaration, which as chance would have it, also works in older versions of Internet Explorer as far back as 5.5!

```
word-wrap: break-word;
```

Adding this to the containing element gives an effect as shown in the following screenshot. Hey presto, the long URL now wraps perfectly!



## New CSS3 selectors and how to use them

CSS3 gives incredible power for selecting elements within a page. You may not think this sounds very glitzy but trust me, it will make your life easier and you'll love CSS3 for it! I'd better qualify that bold claim...

### CSS3 attribute selectors

You've perhaps used existing CSS attribute selectors to target rules. For example, consider the following rule:

```
img[alt] {  
  border: 3px dashed #e15f5f;  
}
```

This would target any image tags in the markup which have an alt attribute:

```

```

You can also narrow things down by specifying what the attribute value is. For example, consider the following rule:

```
img[alt="atwi_oscar"] {  
  border: 3px dashed #e15f5f;  
}
```

This would only target images which have an `alt` attribute of `atwi_oscar`. So far, so big deal we could do that in CSS2. What is CSS3 bringing to the party? Principally, three new "substring matching" attribute selectors...

## CSS3 substring matching attribute selectors

CSS3 lets us select elements based upon the substring of their attribute selector. That sounds complicated. It isn't! We can now select an element, based on the contents of the attribute. The three options are whether the attribute is:

- Beginning with the prefix
- Contains an instance of
- Ends with the suffix

Let's see what they look like.

### The "beginning with" substring matching attribute selector

The "beginning with" substring matching attribute selector has the following syntax:

```
Element [attribute^="value"]
```

In practical use, if I want to select all images on the site that had an `alt` attribute that *began with* `film`, I would write the following rule:

```
img[alt^="film"] {  
  border: 3px dashed #e15f5f;  
}
```

The key character in all this is the `^` symbol which means "begins with".

### The "contains an instance of" substring matching attribute selector

The "contains an instance of" substring matching attribute selector has the following syntax:

```
Element [attribute*="value"]
```

In practical use, if I want to select all images on the site that had an alt attribute that *contained* film I would write the following rule:

```
img[alt*="film"] {  
  border: 3px dashed #e15f5f;  
}
```

The key character in all this is the \* symbol which means "contains".

## The "ends with" substring matching attribute selector

The "ends with" substring matching attribute selector has the following syntax:

```
Element [attribute$="value"]
```

In practical use, if I want to select all images on the site that had an alt attribute that *ended with* film I would write the following rule:

```
img[alt$="film"] {  
  border: 3px dashed #e15f5f;  
}
```

The key character in all this is the \$ symbol which means "ends with".

## A practical, real world example

How can these substring attribute selectors actually help? Let me give you an example where I often use CSS3 attribute selectors. If I build a website with a Content Management System (for example, Wordpress, Concrete, or Magento), it often gives the client the ability to add new pages. For example, perhaps they are adding a piece of news about their company or a product update. Each time they add a page in the CMS, the generated HTML will include an ID value for the <body> or other relevant tag, which helps distinguish the page, markup wise, from others. For example, one client was involved in Motorsport and had a "Racing History" section with yearly reports. Each <body> tag would have an ID for the year:

```
<body id="2003">
```



### IDs can start with numbers in HTML5

If you're not used to coding in HTML5, you might assume that an ID beginning with a number is invalid, as it was in HTML 4.01. However, HTML5 removes that restriction, the only things to remember with ID names in HTML5 is that there should be no spaces in the ID name and it must be unique on the page. For more information visit <http://dev.w3.org/html5/spec/Overview.html#the-id-attribute>.

I needed the navigation bar link for "Racing History" to be highlighted when any of these yearly pages were viewed, as they related to the "Racing History" section. However, rather than write a style rule covering every future year, I was able to write a defensive (they are sometimes referred to as "defensive" rules as they try and safeguard against future events) CSS3 rule:

```
body[id^="2"] .navHistory { color: #00b4ff; }
```

This means that any element with a class of `.navHistory`, that is a descendant of a body with an ID beginning with 2 (for example, 2002, 2003, 2004, and on) will be colored with the hex value of `#00b4ff`. One simple rule covers all eventualities. Unless of course the website is still in its current form by the year 3000—in which case, chances are, even if I eat and exercise well, I won't be able to continue its upkeep...

## CSS3 structural pseudo-classes

The more often you code websites, the more often it's likely you'll need to solve the same problem again and again. Let's consider a typical example. Horizontal navigation bars are often made up of a number of equally spaced `<li>` links. Suppose we need margin to the left and right side of each list item, except for the first and last list item. Historically, we have been able to solve this problem by adding a semantically superfluous classname to the first and last `<li>` elements in the list, as shown in the highlighted lines in the following code snippet:

```
<ul>
  <li class="first"><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 class="last"><a href="#">Quiz</a></li>
</ul>
```

And then by adding a couple of rules in the CSS, we can amend the margin for those two list items:

```
li {
  margin-left: 5%;
  margin-right: 5%;
}
.first {
  margin-left: 0px;
}
.last {
  margin-right: 0px;
}
```

This works but isn't flexible. For example, when building a website built on a CMS system, list items for linking new content might be added automatically, so it might not be a simple task to add or remove the `last` or `first` class to the correct list item in the markup.

## The `:last-child` selector

CSS2.1 already had a selector applicable for the first item in a list:

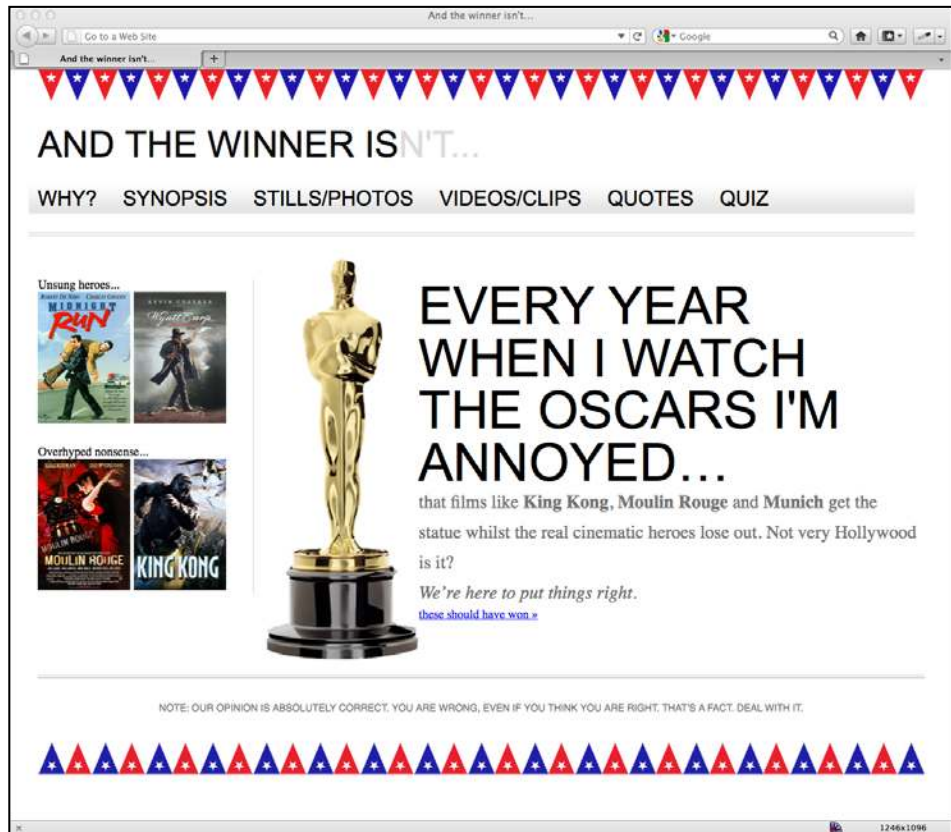
```
li:first-child
```

However, CSS3 adds a selector that can also match the last:

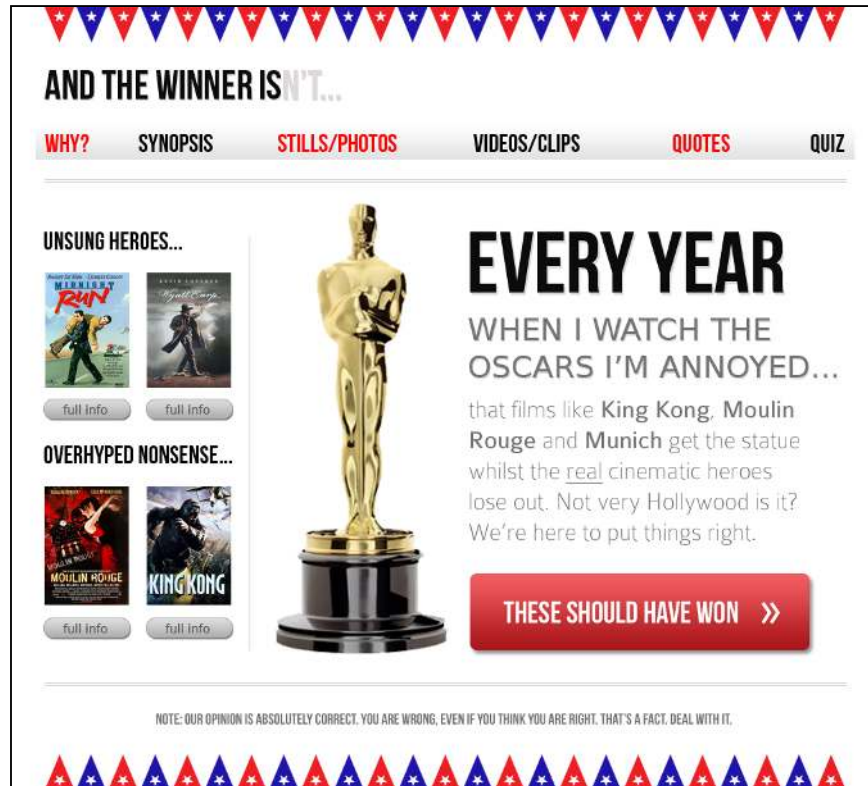
```
li:last-child
```

Using these selectors together, we don't need any additional classes in our markup.

We'll fix up our *And the winner isn't...* site navigation using this and a combination of the `display: table` property. The following screenshot shows how things look currently:



Now, let's take a look at the graphic mockup:



The navigation bar links span the full width of the design, which we need to replicate. Our markup for the navigation looks like this:

```
<nav role="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>
</nav>
```

First, we'll set the `nav` element to be a table:

```
nav {
  display: table;
  /* more code... */
}
```

Then the `<ul>` to be displayed as a `table-row`:

```
nav ul {
  display: table-row;
  /* more code... */
}
```

And finally the list-items to display as `table-cells`:

```
nav ul li {
  display: table-cell;
  /* more code... */
}
```

This means that if extra list items are added, they will automatically space themselves accordingly. Finally, we'll use our CSS selectors to align the text to the right and left of the first and last list items:

```
nav ul li:last-child {
  text-align: right;
}

nav ul li:first-child {
  text-align: left;
}
```



Then in the browser, our navigation is approaching our original composite:



### Don't worry; these tables are only for display!



You may be wondering what on earth I'm thinking of, to suggest that we use a table for the navigational layout. However, don't forget, these tables are only presentational. That means they exist only in the CSS and are nothing to do with the markup. We are merely telling the browser we want those elements to appear and behave as if they were a table, not actually be a table. Displaying the markup in this manner also doesn't preclude us from using a different layout type for a different viewport, for example, `display: inline-block` for viewports below 768 px.

## The nth-child selectors

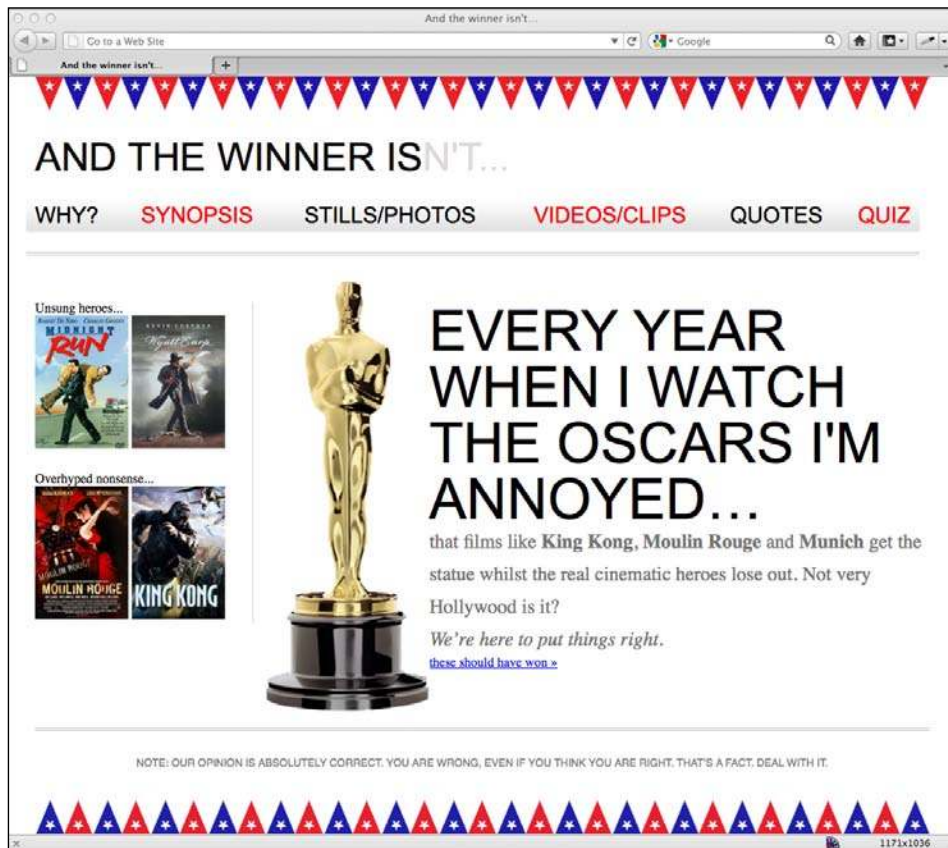
But what about those alternate colors shown in the navigation bar links of the original composite? Again, CSS3 has a selector that can solve this problem for us without the need for additional markup:

```
:nth-child(even)
```

Let's use this selector to fix the problem and then we can look at some of the many ways that nth-child can solve problems that previously required extra markup. I'll add alternate red links in the navigation bar by adding the following style rule:

```
nav ul li:nth-child(even) a {  
  color: #fe0208;  
}
```

And now we have alternate colors in the navigation links:



How about that? Not a line of jQuery in site and no extra markup! What did I tell you? CSS3 selectors are great!

## Understanding what nth rules do

Amongst frontend web developers and designers, nothing makes mathematics weaklings tremble quite like the **nth-based rules** (well, you know, except maybe someone asking you to code a little PHP or give them a hand with some REGEX expressions). Let's see if we can make sense of the beast and gain a little respect from those backend wizards.

When it comes to selecting elements in the tree structure of the DOM (Document Object Model or more simplistically, the elements in a page's markup) CSS3 gives us incredible flexibility with a few nth-based rules — `:nth-child(n)`, `:nth-last-child(n)`, `:nth-of-type(n)`, and `:nth-last-of-type(n)`. We've seen that we can use (odd) or (even) values (as we have to fix our navigation above) but the (n) parameter can be used in another couple of ways:

- Used as an integer; for example, `:nth-child(2)` — would select the second item
- Used as a numeric expression; for example, `:nth-child(3n+1)` — would start at 1 and then select every third element

The integer based property is easy enough to understand, just enter the element number you want to select. The numeric expression version of the selector is the part that can be a little baffling for mere mortals. Let's break it down. For practicality, within the brackets, I start from the right. So, for example, if I want to figure out what `(2n+3)` will select, I start at the right (from the third item) and know it will select every second element from that point on. I've amended our navigation rule to illustrate this:

```
nav ul li:nth-child(2n+3) a {
  color: #fe0208;
}
```

As you can see, the third list item is colored and then every subsequent second one after that (if there were 100 list items, it would continue selecting every second list item):



How about selecting everything from the second item onwards? Well, although you could write `:nth-child(1n+2)`, you don't actually need the first number 1 as unless otherwise stated,  $n$  is equal to 1. We can therefore just write `:nth-child(n+2)`. Likewise, if we wanted to select every third element, rather than write `:nth-child(3n+3)`, we can just write `:nth-child(3n)` as every third item would begin at the third item anyway, without needing to explicitly state it.

The expression can also use negative numbers for example, `:nth-child(3n-2)` starts at minus 2 and then selects every third item. Here's our navigation amended with the following rule:

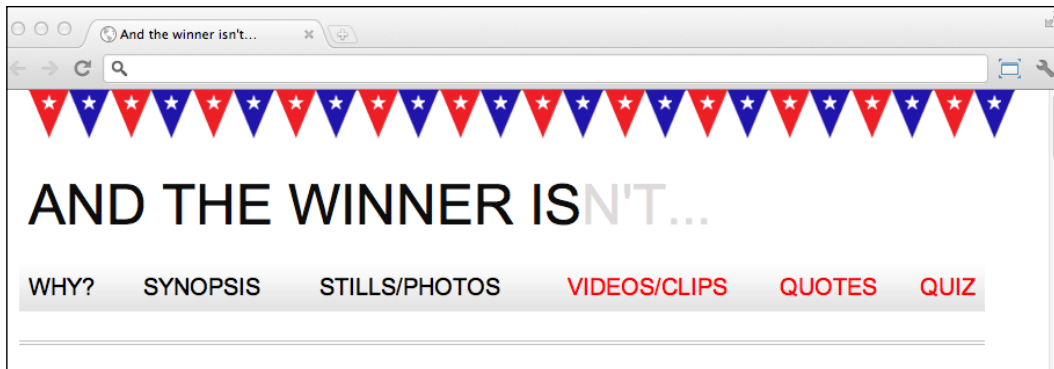
```
nav ul li:nth-child(3n-2) a {
  color: #fe0208;
}
```

And here's what it gives us in the browser:



Hopefully, that's making perfect sense now?

The `child` and `last-child` differ in that the `last-child` variant works from the opposite end of the document tree. For example, `:nth-last-child(-n+3)` starts at 3 from the end and then selects all the items after it. Here's what that rule gives us in the browser:



Finally, let's consider `:nth-last-of-type`. Whilst the previous examples count any children regardless of type, `:nth-last-of-type` lets you be specific about the type of item you want to select. Consider the following markup:

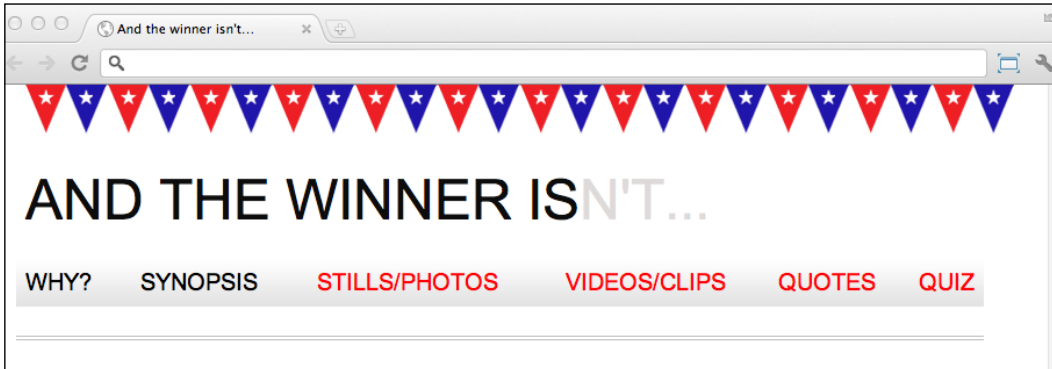
```
<ul>
  <li class="internal"><a href="#">Why?</a></li>
  <li><a href="#">Synopsis</a></li>
  <li class="internal"><a href="#">Stills/Photos</a></li>
  <li class="internal"><a href="#">Videos/clips</a></li>
  <li class="internal"><a href="#">Quotes</a></li>
  <li class="internal"><a href="#">Quiz</a></li>
</ul>
```

Note that the second list item doesn't have the `internal` class added to it.

Consider the following rule:

```
nav ul li.internal:nth-of-type(n+2) a {
  color: #fe0208;
}
```

You can see that we are telling the CSS, "From the second matching item, target every `<li>` item with a class called `internal`. And here's what we see in the browser:



**CSS3 doesn't count like jQuery!**

If you're used to using jQuery you'll know that it counts from 0 upwards. For example, if selecting an element in jQuery, an integer value of 1 would actually be the second element. CSS3 however, starts at 1 so that a value of 1 is the first item it matches.

## The negation (`:not`) selector

Another handy selector is the negation pseudo-class selector. This is used to select everything that isn't something else. For example, keeping the same markup as the previous example, if we change our rule as follows:

```
nav ul li:not(.internal) a {
  color: #fe0208;
}
```

You can see that we are opting to select every list item that doesn't have the `internal` class. So in the browser, we see this:



So far we have looked primarily at what's known as **structural pseudo-classes** (full information on this is available at <http://www.w3.org/TR/selectors/#structural-pseudos>). However, CSS3 has many more selectors. If you're working on a web application, it's worth looking at the full list of UI element states pseudo-classes (<http://www.w3.org/TR/selectors/#UIstates>), as they can, for example, help you target rules based on whether something is selected or not.

## Amendments to pseudo-elements

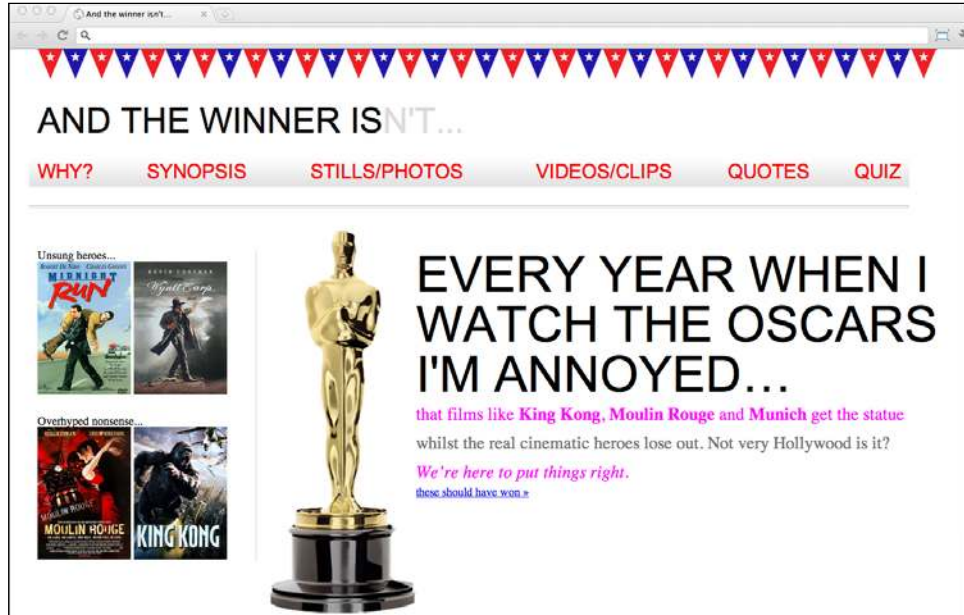
Pseudo-elements have been around since CSS2 but the CSS3 specification revises the syntax of their use very slightly. To refresh your memory, until now, `p:first-line` would target the first line in a `<p>` tag. Or `p:first-letter` would target the first letter. Well, CSS3 asks us to separate these pseudo-elements with a double colon to differentiate them from pseudo-classes. Therefore, we should write `p::first-letter` instead. Note that however Internet Explorer 8 and lower versions don't understand the double colon syntax; they understand only the single colon syntax.

## Is `:first-line` handy for responsive designs?

One thing that you may find particularly handy about the `:first-line` pseudo-element is that it is specific to the viewport. For example, if we write the following rule:

```
p::first-line {
  color: #ff0cff;
}
```

As you might expect, the first line is rendered in an awful shade of pink (I was thinking of Moulin Rouge at the time):



However, on a different viewport, it renders a different selection of text:





So, without needing to alter the markup, with a responsive design, there's a handy way of having the first visual (as the browser renders it, not as it appears in the markup) line of text appear differently than the others.

Hopefully this brief foray into CSS3 selectors illustrates how they help keep a responsive design and code base free of additional markup. In the past, I've needed to use a JavaScript library such as jQuery to make complicated selections but CSS3 often negates that need. It's also comforting to know that the CSS3 selectors module is already at the W3C Recommendation status; so it's a very mature module that's unlikely to change much from here on.

## Custom web typography

For years we've made do with a boring selection of web safe fonts. When some fancy typography was essential for a design, we've typically substituted a graphical element for it and used a text-indent rule to shift the actual text from the viewport.

There have been a few further options for adding fancy typography to a page. sIFR (<http://www.mikeindustries.com/blog/sifr/>) and Cufón (<http://cufon.shoogolate.com/generate/>) used Flash and JavaScript respectively to re-make text elements appear as the fonts they were intended to be. However, with a responsive design, we want a lean, mean, content-serving machine, and images and code flab should be avoided where possible. Thankfully, CSS provides a means of custom web typography that is now ready for the big time.

## The @font-face CSS rule

The @font-face CSS rule has been around since CSS2 (but subsequently absent in CSS 2.1). It was even supported partially by Internet Explorer 4 (no, really)! So what's it doing here, when we're supposed to be talking about CSS3?

Well, as it turns out, @font-face has been re-introduced for the CSS3 Fonts module (<http://www.w3.org/TR/css3-fonts>). Due to the historic legal quagmire of using fonts on the web, it's only recently started to gain serious traction as the de facto solution for web typography. There's also the issue of the varying font formats and implementations from different vendors. For example, the **Embedded OpenType (EOT)** font was Internet Explorer's (and not anyone else's) preferred choice of font format. Others favor the more common place **TrueType (TTF)**, whilst there is also **Scalable Vector Graphics (SVG)** and **Web Open Font Format (WOFF)**. When it comes to using @font-face for your web typography, there is both good news and bad. First the bad...

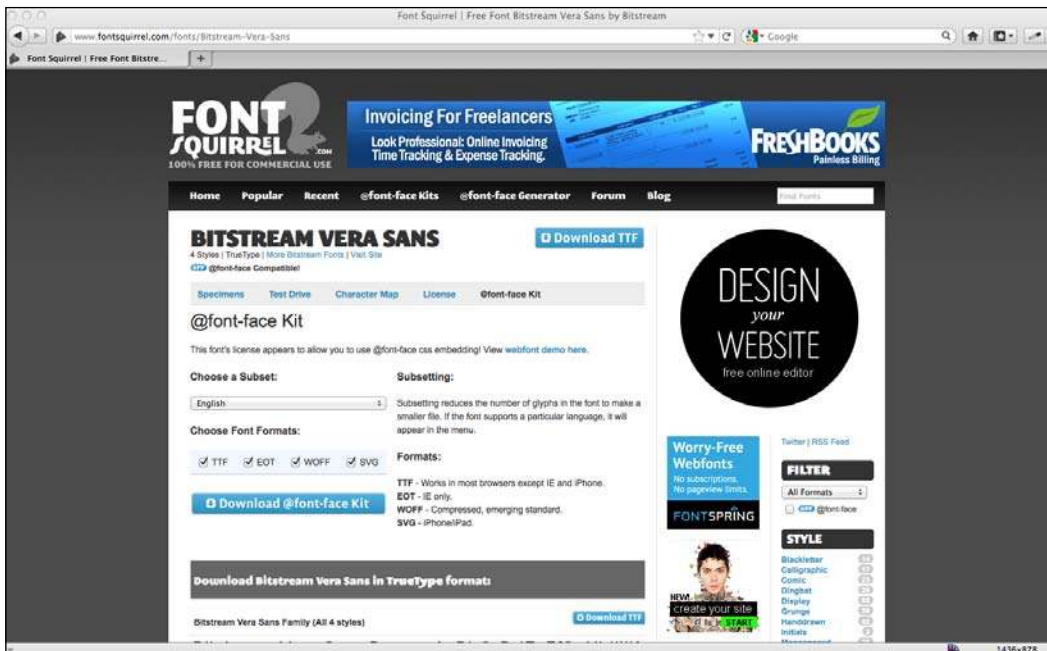
Until a single universal format wins out, it's necessary to serve multiple versions of the same font to cover the different browser implementations. Much as there are competing video formats, we also need a single font format for the web to emerge victorious before dropping support for the others.

However, the good news is that adding custom fonts for every browser is now easy. Let's do it!

## Implementing web fonts with @font-face

Let's get the *And the winner isn't...* site typography licked into shape with the @font-face CSS rule.

First we need some fonts. There are now a number of great sources for web fonts; both free and paid. My personal favorite is Font Squirrel ([www.fontsquirrel.com](http://www.fontsquirrel.com)) although Google also offers free web fonts, ultimately served with the @font-face rule ([www.google.com/webfonts](http://www.google.com/webfonts)). There are also great, paid services from Typekit ([www.typekit.com](http://www.typekit.com)) and Font Deck ([www.fontdeck.com](http://www.fontdeck.com)).



As chance would have it the fonts used in my composite are all available free from Font Squirrel (I know, I'm a cheapskate!). They are Bebas Neue, Bitstream Vera Sans and Collaborate Thin. Having downloaded the relevant `@font-face` kit for each font from Font Squirrel a look inside the ZIP file of each reveals the font itself in various formats (WOFF, TTF, EOT, and SVG) plus a `stylesheet.css` file containing a font stack for the font needed. For example, the rule for Bebas Neue is as follows:

```
@font-face {
  font-family: 'BebasNeueRegular';
  src: url('BebasNeue-webfont.eot');
  src: url('BebasNeue-webfont.eot?#iefix') format('embedded-
opentype'),
      url('BebasNeue-webfont.woff') format('woff'),
      url('BebasNeue-webfont.ttf') format('truetype'),
      url('BebasNeue-webfont.svg#BebasNeueRegular') format('svg');
  font-weight: normal;
  font-style: normal;
}
```

Much like the way vendor prefixes work, the browser will apply styles from that list of properties (with the lower properties, if applicable, taking precedence) and ignore ones it doesn't understand. That way, no matter what the browser, there should be a font that it can use.

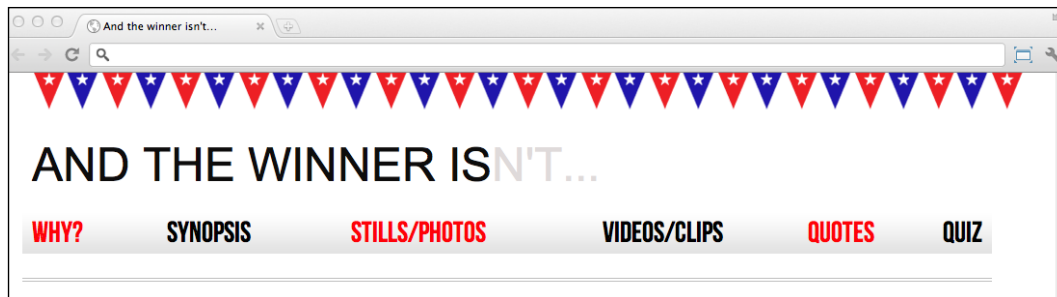
Now, although this block of code is great for fans of copy and paste, it's important to pay attention to the paths the fonts are stored in. For example, I tend to copy the fonts from the ZIP file and store them in a folder inventively called `fonts` on the same level as my `css` folder. Therefore, as I'm usually copying this font stack rule into my main stylesheet, I need to amend the paths. So, my rule becomes:

```
@font-face {
  font-family: 'BebasNeueRegular';
  src: url('../fonts/BebasNeue-webfont.eot');
  src: url('../fonts/BebasNeue-webfont.eot?#iefix')
format('embedded-opentype'),
      url('../fonts/BebasNeue-webfont.woff') format('woff'),
      url('../fonts/BebasNeue-webfont.ttf') format('truetype'),
      url('../fonts/BebasNeue-webfont.svg#BebasNeueRegular')
format('svg');
  font-weight: normal;
  font-style: normal;
}
```

It's then just a case of setting the correct font and weight (if needed) for the relevant style rule. In this case, I want to amend the navigation links to use the new Bebas Neue font:

```
nav ul li a {
  height: 42px;
  line-height: 42px;
  text-decoration: none;
  text-transform: uppercase;
  font-family: 'BebasNeueRegular';
  font-size: 1.875em; /*30 ÷ 16 */
  color: black;
}
```

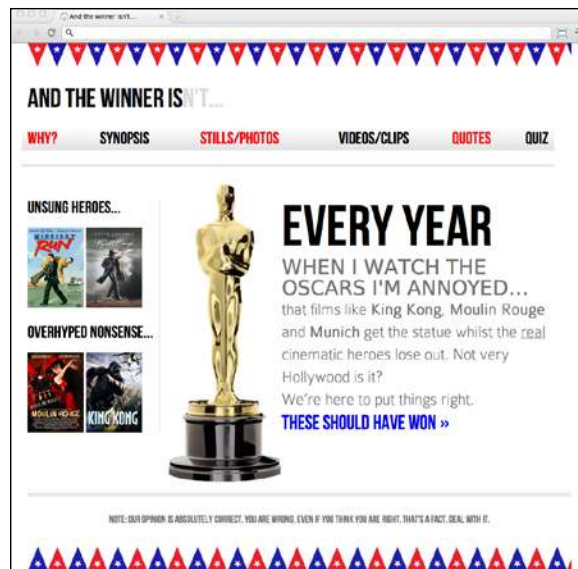
And here is how the navigation bar now looks in the browser:



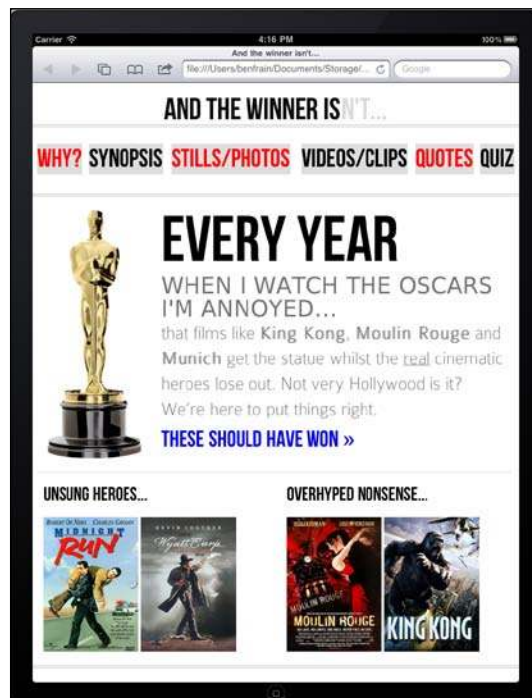
When replacing fonts you'll typically need to amend the font sizing. However, having put the existing font size calculation in a comment to the side, it's easy to amend accordingly. An added bonus is that, if the composite uses the same fonts you are using in the code, you can plug the sizes in direct from the composite file. For example, my composite shows the "EVERY YEAR..." text as 102 px, so using the tried and trusted  $target \div context = result$  technique I can convert this value to ems:

```
#content h1 {
  font-family: Arial, Helvetica, Verdana, sans-serif;
  text-transform: uppercase;
  font-family: 'BebasNeueRegular';
  font-size: 6.375em; /* 102 ÷ 16 */
}
```

Once I've amended the `font-family` and `font-size` declarations for all relevant rules, the front page now looks like the following in Google Chrome (using the WOFF font format):

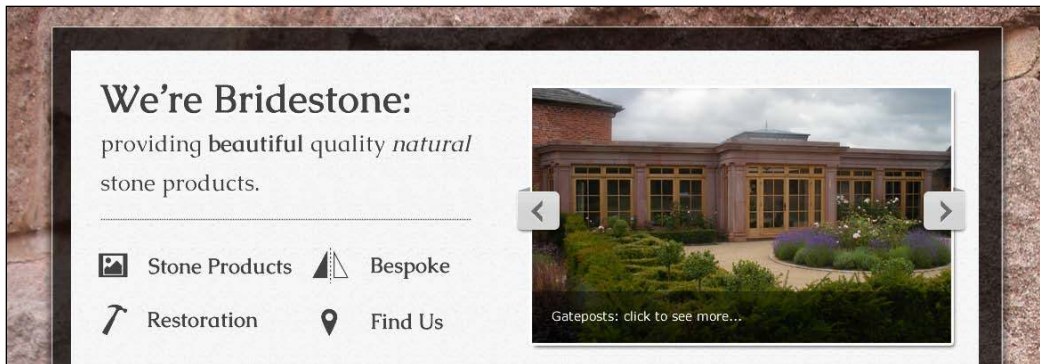


The design still isn't perfect but the typography now perfectly mirrors that of our original composite. For comparison, here's how it's looking on the iPad 2 (which supports TTF fonts from version iOS 4.2 onwards):



## Help—my CSS3 @font-face headings look messy

This problem drove me to distraction when I first started using @font-face fonts to set my web typography free. It's not particular to responsive designs, it can happen with any heading that has a @font-face font applied. Here's a portion of a design composite I was working on:



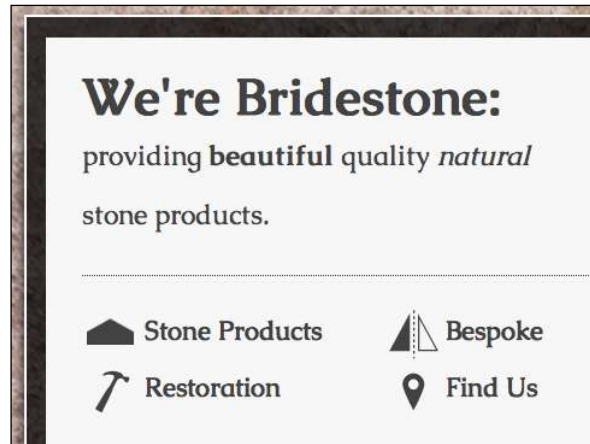
When I had built the site, the relevant markup was as follows:

```
<div class="intro">
  <h1>We're Bridestone: <span>providing <b>beautiful</b> quality
<i>natural</i> stone products.</span></h1>
  ...more code...
</div> <!-- intro:END -->
```

And here was the relevant CSS:

```
.intro h1 {
  font-family: CaudexBold, "Times New Roman", Times, serif;
  font-size: 2.63636364em;
  line-height: 1em;
}
.intro h1 span {
  font-size: 0.545454545em;
  font-family: CaudexRegular, "Times New Roman", Times, serif;
  font-weight: normal;
}
```

However, although I was using `@font-face` so that I could use exactly the same font as the composite, the header still looked a little messy in the browser:

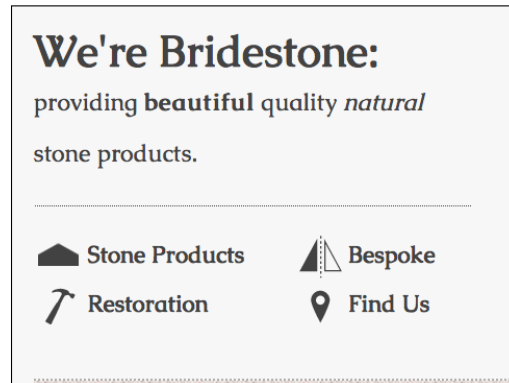


Hopefully you can make out that the **We're Bridestone** text doesn't match the composite. It's thicker, which degrades the clarity!

It turns out that the problem relates to font weight. Unless explicitly stating the `font-weight` property, many browsers will apply a standard `font-weight` (typically, 700) to any heading elements. The solution therefore is to always define the `font-weight` of any `@font-face` fonts used in heading elements. For example, in this instance, I amended the CSS to:

```
.productIntro h1 {
  font-family: CaudexBold, "Times New Roman", Times, serif;
  font-weight: 400;
  font-size: 2.63636364em;
  line-height: 1em;
}
```

This then overrides the `font-weight` value that the browser would ordinarily use and as shown in the following screenshot, the design finally matches the composite in the browser:



## A note about custom @font-face typography and responsive designs

The `@font-face` method of web typography is, on the whole, great. The only caveats to be aware of when using the technique with responsive designs are in relation to the font file size. For example, the *And the winner isn't...* site is using three custom fonts – Bebas Neue, Bitstream Vera Sans, and Collaborate Thin. At worst, if the device rendering the page required the SVG font format, it will require an extra 70 KB of data, compared with using the standard web safe fonts such as Arial. These fonts are also fairly lightweight – others are not! Be sure to check the size of custom fonts if you want the best site performance.

### A truly responsive type unit on the way?



Amongst the current working draft of the CSS3 Fonts module is reference to viewport relative fonts (<http://www.w3.org/TR/css3-values/#viewport-relative-lengths>). The **vw** unit (for viewport width), **vh** unit (for viewport height) and **vm** unit (for viewport minimum; equal to the smaller of either **vm** or **vh**) could be crucial time savers in the years to come. Sadly, at present there is no browser support apart from Internet Explorer 9.



---

## New CSS3 color formats and alpha transparency

So far, CSS3 has given us new powers of selection and the ability to add custom typography to our designs. Now, we'll look at ways that CSS3 allows us to work with color that were simply not possible before.

Firstly, CSS3 allows us to use new methods, such as **RGB** and **HSL**, for declaring color. In addition, it enables us to use those two methods alongside an alpha channel (**RGBA** and **HSLA** respectively).

### RGB color

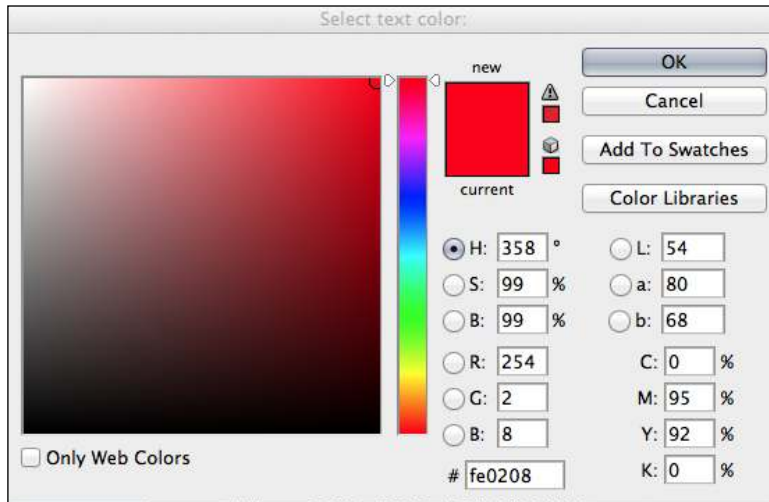
RGB (Red, Green, and Blue) is a coloring system that's been around for decades. It works by defining different values for the red, green, and blue components of a color. For example, the red color used for the odd numbered navigation links on the *And the winner isn't...* site is currently defined in the CSS as a hex (hexadecimal) value, #fe0208:

```
nav ul li:nth-child(odd) a {
  color: #fe0208;
}
```

However, with CSS3, it can equally be described as an RGB value:

```
nav ul li:nth-child(odd) a {
  color: rgb(254, 2, 8);
}
```

Most image editing applications show colors as both hex and RGB values in their color picker. The following screenshot shows the Photoshop color picker, with the **R**, **G**, and **B** boxes showing the values for each channel:



You can see that the **R** value is **254**, the **G** value is **2** and the **B** value is **8**. Which is easily transferable to the CSS color property value. In the CSS, after defining the color mode (for example, `rgb`) the values for red, green and blue colors are comma separated in that order within parenthesis.

## HSL color

Besides RGB, CSS3 also allows us to declare color values as HSL (Hue, Saturation, and Lightness).



### HSL isn't the same as HSB!

Don't make the mistake of thinking that the HSB (Hue, Saturation, and Brightness) value shown in the color picker of image editing applications such as Photoshop is the same as HSL – it isn't!

---

What makes HSL such a joy to use is that it's relatively simple to understand the color that will be represented based on the values given. For example, unless you're some sort of color picking Ninja, I'd wager you couldn't instantly tell me what color `rgb(255, 51, 204)` is? Any takers? No, me neither. However, show me the HSL value of `hsl(315, 100%, 60%)` and I could take a guess that it is somewhere between Magenta and Red color (it's actually a festive pink color – perhaps I'm starting to like Moulin Rouge after all). How do I know this? Simple...

HSL works on a 360° color wheel. The first figure in a HSL color, represents Hue, and has Yellow at 60°, Green at 120°, Cyan at 180°, Blue at 240°, Magenta at 300° and finally Red at 360°. So as the aforementioned HSL color had a hue of 315, it's easy to know that it will be between Magenta (at 300°) and Red (at 360°). The following two values for saturation and lightness, specified as percentages, merely alter the base hue. For a more saturated or colorful appearance, use a higher percentage in the second value. The final value, controlling the lightness, can vary between 0 percent for black and 100 percent for white.

So, once you've defined a color as an HSL value, it's also easy to create variations on it, merely by altering the saturation and lightness percentages. For example, our red navigation links can be defined in HSL values as follows:

```
nav ul li:nth-child(odd) a {
    color: hsl(359, 99%, 50%);
}
```

If we wanted to make a slightly darker color on hover, we could use the same HSL value and merely alter the lightness (the final value) percentage value only, as shown in the following code snippet:

```
nav ul li:nth-child(odd) a:hover {
    color: hsl(359, 99%, 40%);
}
```

In conclusion, if you can remember the mnemonic Young Guys Can Be Messy Rascals (or any other mnemonic you care to memorize) for the HSL color wheel, you'll be able to approximately write HSL color values without resorting to a color picker and also create variations upon it. Show that trick to the savant backend PHP and .NET guys at the office party and earn some quick kudos!

## Fallback color values for IE6, IE7, and IE8

As you might have guessed, RGB and HSL are not supported in Internet Explorer versions below IE9. Therefore, if a fallback color declaration is needed for these browsers, specify it first before the RGB or HSL value. For example, the navigation link rule defined above could have a hex fallback specified like this:

```
nav ul li:nth-child(odd) a {
  color: #fe0208;
  color: hsl(359, 99%, 50%);
}
```

## Alpha channels

So far you'd be forgiven for wondering why on earth we'd bother using HSL or RGB instead of our trusty hex values we've been using for years. Where HSL and RGB differ from hex is that they allow the use of an alpha transparency channel. This means one element with an alpha transparency will show what's beneath it.

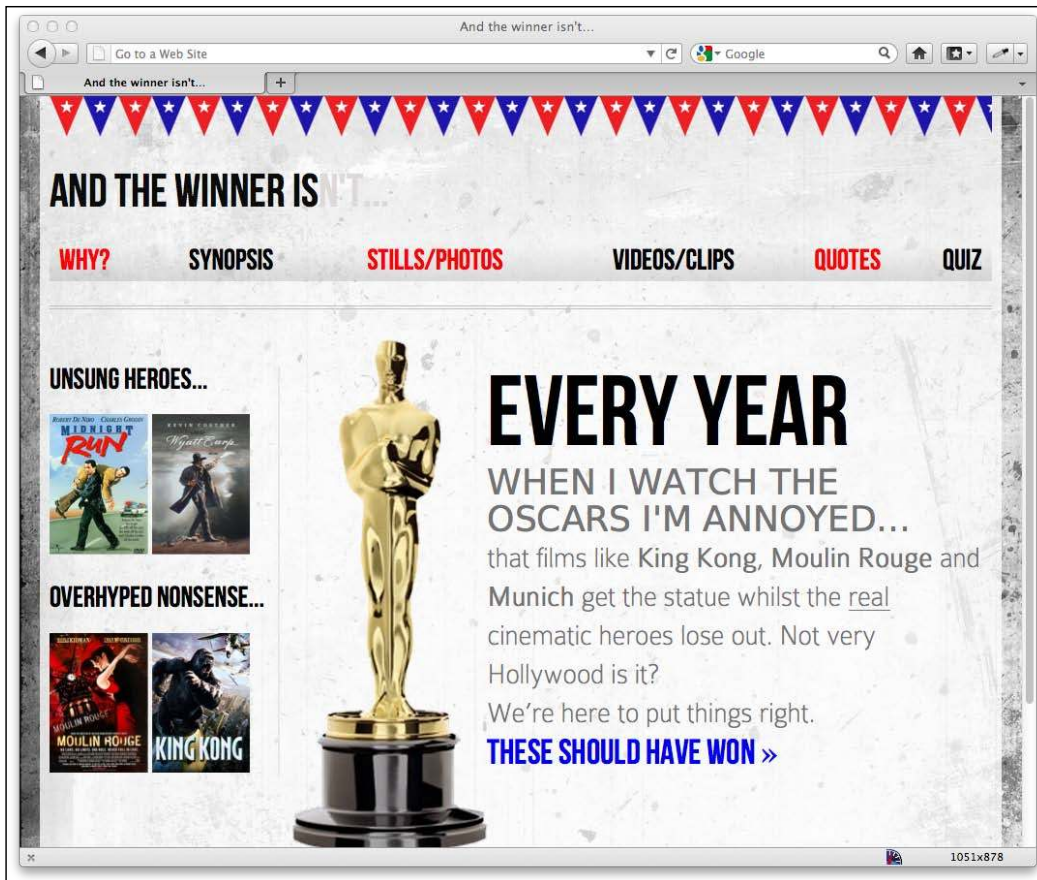
Let's make some amendments to the *And the winner isn't...* home page to illustrate. First, we'll set a grungy background image in the `body` element, as follows:

```
body {
  background: url(../img/grunge.jpg) repeat;
}
```

Now, we'll add a white background in the `#wrapper` div (which encloses all the other elements). However, instead of setting a solid white color with a hex value, we'll set a HSLA value as shown in the highlighted line in the following code snippet:

```
#wrapper {
  margin-right: auto;
  margin-left: auto;
  width: 96%; /* Holding outermost DIV */
  max-width: 1414px;
  background-color: hsla(0, 0%, 100%, 0.8);
}
```

An HSLA color declaration is similar in syntax to a standard HSL rule. However, in addition, you must declare the value as `hsla` (rather than merely `hsl`) and add an additional opacity value, given as a decimal value between 0 (completely transparent) and 1 (completely opaque). Here, we have specified that our white `#wrapper` isn't completely opaque. The following screenshot shows how it looks in the browser:



The RGBA syntax follows the same convention as the HSLA equivalent, using an additional opacity value after the color:

```
background-color: rgba(255, 255, 255, 0.8);
```

Hopefully you can see that the addition of an alpha channel to both the RGB and HSL color modes, allows us a great deal of flexibility when layering elements. It means that we no longer have to rely on the transparency of images (PNG and GIF images, for example) to achieve this type of visual effect, which is great news when building a responsive design.



### Why not just use opacity?

CSS3 also allows elements to have opacity set with the `opacity` declaration. A value is set between zero and one in decimal increments (for example, opacity set to 0.1 is 10 percent). However, this differs from RGBA and HSLA in that setting an opacity value on an element affects the entire element. Whereas, setting a value with HSLA or RGBA meanwhile allows particular parts of an element to have an alpha layer. For example, an element could have an HSLA value for the background but a solid color for the text within it.

The CSS3 Color module was the first of the CSS3 modules to reach the advanced Recommendation stage. Therefore, like the CSS3 Selectors module, CSS3 Colors are good to use right away, safe in the knowledge that the method of implementation is unlikely to change from this point onwards.

## Summary

In this chapter, we've learned how to easily select almost anything we need on the page with CSS3's new selectors. We've also looked at how we can make responsive columns for content in record time and solve common and annoying problems such as long URL wrapping. We now also have an understanding of CSS3's new color module and how we can apply colors with RGB and HSL complete with transparent alpha layers for great aesthetic effects. In this chapter, we've also learned how to add custom fonts to a design with the `@font-face` rule, finally freeing us from the shackles of the humdrum selection of "web-safe" fonts we're used to designing with. Despite all these great new features and techniques, we've only picked at the surface of what we can do with CSS3. Let's move on now and look at even more ways CSS3 can make a responsive design as fast, efficient, and maintainable as possible with CSS3 text shadows, box shadows, gradients, and multiple backgrounds.