2
Media Queries: Supporting Differing Viewports

As we noted in the last chapter, CSS3 consists of a number of bolt-on modules. **Media queries** is just one of these CSS3 modules. Media queries allow us to target specific CSS styles depending upon the display capabilities of a device. For example, with just a few lines of CSS we can change the way content displays based upon things such as viewport width, screen aspect ratio, orientation (landscape or portrait), and so on.

In this chapter, we shall:

- Learn why media queries are needed for a responsive web design
- Learn how a CSS3 media query is constructed
- Understand what device features we can test for
- Write our first CSS3 media query
- Target CSS style rules to specific viewports
- Learn how to make media queries work on iOS and Android devices

You can use media queries today

Media queries are already widely used and enjoy a broad level of browser support (Firefox 3.6+, Safari 4+, Chrome 4+, Opera 9.5+, iOS Safari 3.2+, Opera Mobile 10+, Android 2.1+, and Internet Explorer 9+). Furthermore, there are easy to implement (albeit JavaScript based) **fixes** for common aged browsers such as Internet Explorer versions 6, 7, and 8. If you need to grab the fixes for Internet Explorer versions 6, 7, and 8 now, you'll need to look at *Chapter 9*, *Solving Cross-browser Responsive Challenges*. In short, there's no good reason why we can't get using media queries today!

Specifications at the W3C go through a ratification process (if you have a spare day, knock yourself out with the official explanation of the process at `http://www.w3.org/2005/10/Process-20051014/tr`), from **Working Draft** (**WD**), to **Candidate Recommendation** (**CR**), to **Proposed Recommendation** (**PR**) before finally arriving, many years later, at **W3C Recommendation** (**REC**). So modules at a greater maturity level than others are generally safer to use. For example, CSS Transforms Module Level 3 (`http://www.w3.org/TR/css3-3d-transforms/`) has been at WD status since March 2009 and browser support for it is far scanter than CR modules such as media queries.

Why responsive designs need media queries?

Without the CSS3 media query module, we would be unable to target particular CSS styles at particular device capabilities, such as the viewport width. If you head over to the W3C specification of the CSS3 media query module (`http://www.w3.org/TR/css3-mediaqueries/`), you'll see that this is their official introduction to what media queries are all about:

> *HTML 4 and CSS2 currently support media-dependent style sheets tailored for different media types. For example, a document may use sans-serif fonts when displayed on a screen and serif fonts when printed. 'screen' and 'print' are two media types that have been defined. Media queries extend the functionality of media types by allowing more precise labeling of style sheets.*
>
> *A media query consists of a media type and zero or more expressions that check for the conditions of particular media features. Among the media features that can be used in media queries are 'width', 'height', and 'color'. By using media queries, presentations can be tailored to a specific range of output devices without changing the content itself.*

Media query syntax

So what does a CSS media query look like and more importantly, how does it work?

Enter the following code at the bottom of any CSS file and preview the related web page:

```
body {
  background-color: grey;
}
@media screen and (max-width: 960px) {
  body {
    background-color: red;
  }
}
@media screen and (max-width: 768px) {
  body {
    background-color: orange;
  }
}
@media screen and (max-width: 550px) {
  body {
    background-color: yellow;
  }
}
@media screen and (max-width: 320px) {
  body {
    background-color: green;
  }
}
```

Now, preview the file in a modern browser (at least IE 9 if you use IE) and resize the browser window. The background color of the page will vary depending upon the current viewport size. I've used named colors here for clarity but ordinarily you'd use a HEX code; for example, `#ffffff`.

Now, let's go ahead and break down these media queries to understand how we can make best use of them.

If you are used to working with CSS2 stylesheets you'll know it's possible to specify the type of device (for example, screen or print) applicable to a stylesheet with the `media` attribute of the `<link>` tag. You could do so by placing a link as done in the following code snippet within the `<head>` tags of your HTML:

```
<link rel="stylesheet" type="text/css" media="screen" href="screen-
styles.css">
```

What media queries principally provide is the ability to target styles based upon the *capability* or *features* of a device, rather than merely the *type* of device. Think of it as a question to the browser. If the browser's answer is "true", the enclosed styles are applied. If the answer is "false", they are not. Rather than just asking the browser "Are you a screen?"—as much as we could effectively ask with just CSS2—media queries ask a little more. Instead, a media query might ask, "Are you a screen and are you in portrait orientation?" Let's look at that as an example:

```
<link rel="stylesheet" media="screen and (orientation: portrait)"
href="portrait-screen.css" />
```

First, the media query expression asks the type (are you a screen?), and then the feature (is your screen in portrait orientation?). The `portrait-screen.css` stylesheet will be loaded for any screen device with a portrait screen orientation and ignored for any others. It's possible to reverse the logic of any media query expression by adding `not` to the beginning of the media query. For example, the following code would negate the result in our prior example, loading the file for anything that wasn't a screen with a portrait orientation:

```
<link rel="stylesheet" media="not screen and (orientation: portrait)"
href="portrait-screen.css" />
```

It's also possible to string multiple expressions together. For example, let's extend our first media query example and also limit the file to devices that have a viewport greater than 800 pixels.

```
<link rel="stylesheet" media="screen and (orientation: portrait) and
(min-width: 800px)" href="800wide-portrait-screen.css" />
```

Further still, we could have a list of media queries. If any of the listed queries are true, the file will be loaded. If none are true, it won't. Here is an example:

```
<link rel="stylesheet" media="screen and (orientation: portrait) and
(min-width: 800px), projection" href="800wide-portrait-screen.css" />
```

There are two points to note here. Firstly, a comma separates each media query. Secondly, you'll notice that after `projection`, there is no trailing `and` or feature/value combination in parentheses. That's because in the absence of these values, the media query is applied to all media types. In our example, the styles will apply to all projectors.

Just like existing CSS rules, media queries can conditionally load styles in a variety of ways. So far, we have included them as links to CSS files that we would place within the `<head></head>` section of our HTML. However, we can also use media queries within CSS stylesheets themselves. For example, if we add the following code into a stylesheet, it will make all `h1` elements green, providing the device has a screen width of 400 pixels or less:

```
@media screen and (max-device-width: 400px) {
  h1 { color: green }
}
```

We can also use the `@import` feature of CSS to conditionally load stylesheets into our existing stylesheet. For example, the following code would import the stylesheet called `phone.css`, providing the device was screen based and had a maximum viewport of 360 pixels:

```
@import url("phone.css") screen and (max-width:360px);
```

Remember that using the `@import` feature of CSS, adds to HTTP requests (which impacts load speed); so use this method sparingly.

What can media queries test for?

When building responsive designs, the media queries that get used most often relate to a device's viewport width (`width`) and the width of the device's screen (`device-width`). In my own experience, I have found little call for the other capabilities we can test for. However, just in case the need arises, here is a list of all capabilities that media queries can test for. Hopefully, some will pique your interest:

- `width`: The viewport width.
- `height`: The viewport height.
- `device-width`: The rendering surface's width (for our purposes, this is typically the screen width of a device).
- `device-height`: The rendering surface's height (for our purposes, this is typically the screen height of a device).
- `orientation`: This capability checks whether a device is portrait or landscape in orientation.
- `aspect-ratio`: The ratio of width to height based upon the viewport width and height. A 16:9 widescreen display can be written as `aspect-ratio: 16/9`.

- `device-aspect-ratio`: This capability is similar to `aspect-ratio` but is based upon the width and height of the device rendering surface, rather than viewport.
- `color`: The number of bits per color component. For example, `min-color: 16` will check that the device has 16-bit color.
- `color-index`: The number of entries in the color lookup table of the device. Values must be numbers and cannot be negative.
- `monochrome`: This capability tests how many bits per pixel are in a monochrome frame buffer. The value would be a number (integer), for example `monochrome: 2`, and cannot be negative.
- `resolution`: This capability can be used to test screen or print resolution; for example, `min-resolution: 300dpi`. It can also accept measurements in dots per centimetre; for example, `min-resolution: 118dpcm`.
- `scan`: This can be either progressive or interlace features largely particular to TVs. For example, a 720p HD TV (the *p* part of 720p indicates "progressive") could be targeted with `scan: progressive` whilst a 1080i HD TV (the *i* part of 1080i indicates "interlaced") could be targeted with `scan: interlace`.
- `grid`: This capability indicates whether or not the device is grid or bitmap based.

All the above features, with the exception of `scan` and `grid`, can be prefixed with `min` or `max` to create ranges. For example, consider the following code snippet:

```
@import url("phone.css") screen and (min-width:200px) and (max-
width:360px);
```

Here, a minimum (`min`) and maximum (`max`) have been applied to `width` to set a range. The `phone.css` file will only be imported for screen devices with a minimum viewport width of 200 pixels and a maximum viewport width of 360 pixels.

Using media queries to alter our design

As you're, no doubt, aware that CSS stands for Cascading Style Sheet. By their very nature styles further down a cascading stylesheet override equivalent styles higher up (unless styles higher up are more specific). We can therefore set base styles at the beginning of a stylesheet, applicable to all versions of our design, and then override relevant sections with media queries further on in the document. For example, set navigation links as simple text links for the large viewport version of a design (where it's more likely that users will be using a mouse) and then overwrite those styles with a media query to give us a larger target area (for finger presses) for more limited viewports.

The best way to load media queries for responsive designs

Although modern browsers are smart enough to ignore media query targeted files that are not intended for them, it doesn't always stop them actually downloading the files. There is therefore little advantage (apart from personal preference and/or compartmentalization of code) in separating different media query styles into separate files. Using separate files increases the number of HTTP requests needed to render a page, which in turn makes the page slower to load.

The fastest JavaScript tool, Respond.js (`https://github.com/scottjehl/Respond`) for adding partial media query support to Internet Explorer 8 and lower versions is also currently unable to parse CSS referenced by the `@import` command. I'd therefore recommend adding media queries styles within an existing stylesheet. For example, in the existing stylesheet, simply add the media query using the following syntax:

```
@media screen and (max-width: 768px) { YOUR STYLES }
```

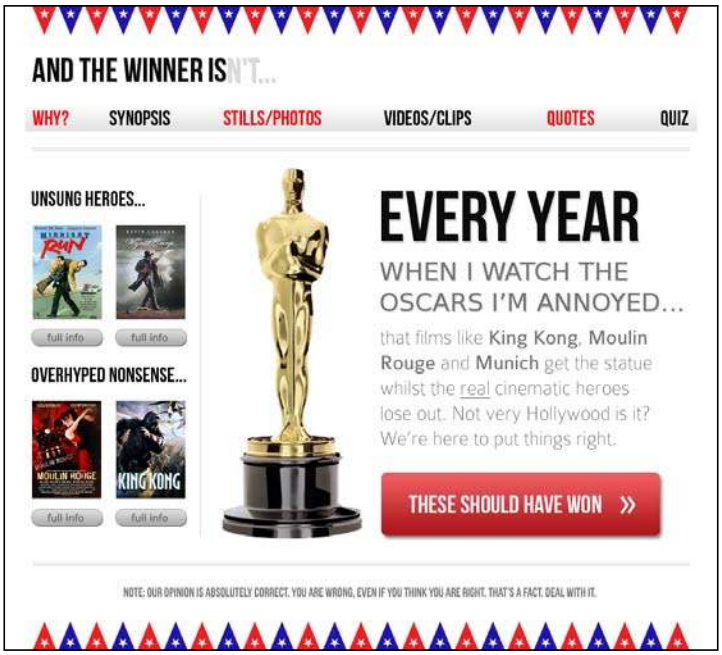
Our first responsive design

I don't know about you but I'm itching to get started with a responsive web design! Now we understand the principles of media queries, let's test drive them and see how they work in practice. And I have just the project we can test them on. Indulge me a brief digression…

I like films. However, I commonly find myself disagreeing with others (perhaps that is a contributing factor of me spending my days writing code… alone!), specifically about what is and what isn't a good film. When the Oscar nominees are announced I often have a strong feeling of revulsion in the pit of my stomach. I can't help feeling that different films should be picking up the accolades. I'd like to launch a small site called *And the winner isn't…*, which you'll be able to view online at http://www.andthewinnerisnt.com/ on the Web. It will celebrate the films that should have won, berate the ones that did (and shouldn't have) and have video clips, quotes, images, and quizzes thrown in to illustrate I'm correct (I know, I shouldn't need to but I'm good like that).

Don't panic but our design is fixed-width

Much like the graphic designers whom I previously scolded for not considering differing viewports, I started a graphical mockup based around a fixed, 960 pixel-wide grid. In reality, although theoretically it would always be best to start a design thinking about the mobile/small screen experience and building up from there, it's going to be some years until everyone understands the benefits of that thinking. Until then, it's likely you'll need to take existing desktop designs and "retro-fit" them to work responsively. As this is the scenario we are likely to find ourselves in for the foreseeable future, we will begin our process with a fixed-width design of our own. The following screenshot shows what the unfinished fixed-width mockup looks like:



Breaking it down, it has a very simple and common structure—header, navigation, sidebar, content, and footer. Hopefully, this is typical of the kind of structure you're asked to build week in and week out.

In *Chapter 4*, *HTML5 for Responsive Designs*, I'll tell you why you should be using HTML5 for your markup. However, I'm going to let this slide for now, as we're so eager to test our new media queries skills. So, let's take our first stab at using media queries using good ol' HTML 4 markup. Without the actual content, the basic structure in HTML 4 markup looks like the following code:

```
<!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" />
<title>And the winner isn't</title>
<link href="css/main.css" rel="stylesheet" type="text/css" />
</head>

<body>

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

Looking at the design file in Photoshop, we can see that the header and footer are 940 pixels wide (with 10-pixels margin on either side), and the sidebar and content occupy 220 and 700 pixels, respectively, with a 10-pixel margin on either side of each.

First off, let's set up our structural blocks (header, navigation, sidebar, content, and footer) in the CSS. After inserting the "reset" styles, our super exciting (not!) CSS for the page looks as follows:

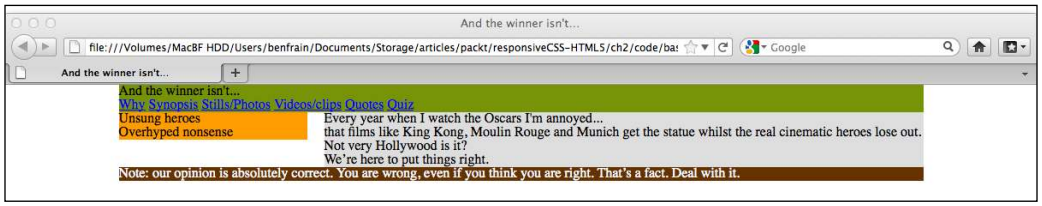
```
#wrapper {
  margin-right: auto;
  margin-left: auto;
  width: 960px;
}

#header {
  margin-right: 10px;
  margin-left: 10px;
  width: 940px;
  background-color: #779307;
}

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

```
}

#sidebar {
  margin-right: 10px;
  margin-left: 10px;
  float: left;
  background-color: #fe9c00;
  width: 220px;
}

#content {
  margin-right: 10px;
  float: right;
  margin-left: 10px;
  width: 700px;
  background-color: #dedede;
}

#footer {
  margin-right: 10px;
  margin-left: 10px;
  clear: both;
  background-color: #663300;
  width: 940px;
}
```

To illustrate how the structure works, besides adding the additional content (*sans* images) I've also added a background color to each structural section.

Just in case you missed the memo, "reset" styles are a bunch of cover-all CSS declarations that reset the various default styles that different browsers render HTML elements with. They are added to the beginning of the main stylesheet in an attempt to reset each browser's own styles to a level playing field so that styles added afterwards in the stylesheet have the same effect across different browsers. There is no "perfect" set of reset styles and most developers have their own variation on the theme. The reset styles I use in HTML 4 documents are a combination of Eric Meyer's original (`http://meyerweb.com/eric/tools/css/reset/`) and a bunch of personal preferences and tricks I have picked up from studying the code of other incredibly clever folks such as Dan Cederholm (`http://simplebits.com`). If you don't currently use reset styles, inserting Eric's reset styles at the start of your HTML 4 document will be a good first step. I feel there are better starting points for HTML5 documents, such as `normalize.css` (`http://necolas.github.com/normalize.css/`) and we'll look at that in *Chapter 4, HTML5 for Responsive Designs*.

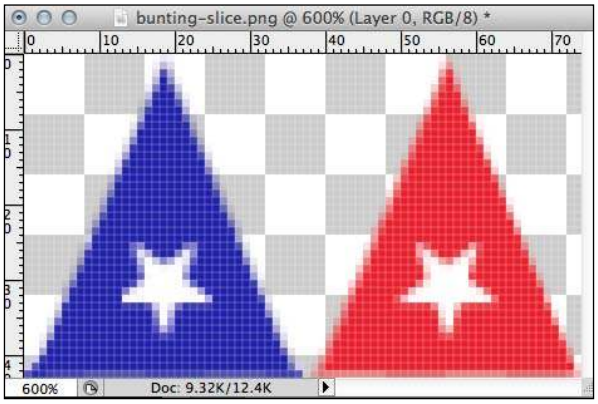
In a browser with a viewport larger than 960 pixels, the following screenshot shows how the basic structure looks:



There are numerous other ways the same kind of fixed left/right content structure could be achieved with CSS; you'll no doubt have your own preference. What's universally true of them all however is that as the viewport decreases to less than 960 pixels, areas of the content at the right start getting clipped.

Responsive designs—making images as economical as possible

For the sake of illustrating the problems with the code structure as it is, I've gone ahead and added some of the aesthetic styling from our graphic file into the CSS. As this will ultimately be a responsive design, I've been sure to slice up the background images in the most economical way. For example, for the bunting flags at the top and bottom of the design, instead of creating one long strip as a graphic file, I have sliced around two flags. This slice will then be repeated horizontally as a background image across the viewport to give the illusion of one long strip (no matter how wide things get). In real terms, this makes a difference of 16 KB (the full 960 pixels wide strip was a 20 KB `.png` file whilst the slice was only 4 KB) on each strip. A mobile user viewing the site over a phone network will appreciate that economy! The following screenshot shows what the slice looks like (zoomed to 600 percent) before export:



With the background images in place and basic font sizes in place, here is how the *And the winner isn't…* site looks in a browser window:

Style wise, there is still a lot of work to do. For example, the navigation menu doesn't alternate between red and black, the main **THESE SHOULD HAVE WON** button in the content area and the **full info** buttons from the sidebar are missing and the fonts are all a far cry from the ones shown in the graphic file. However, all these things are fixable with HTML5 and CSS3. Using HTML5 and CSS3 to solve these problems, rather than merely inserting image files (as we may have done previously), will make a website in tune with our responsive goal. Remember that we want our code and data overheads as lean as possible to afford users with limited bandwidth speeds an enjoyable experience.

Content clipping in smaller viewports

For now, let's put aside the aesthetic problems and keep focused on the fact than when the viewport is reduced below 960 pixels, there is some seriously nasty clipping on our work in progress home page:

We've only reduced it to 673 pixels wide; imagine how bad it must look on something like an iPhone 3GS? That only has a 320 x 480 pixel display. Just take a look at the following screenshot:

Oh, hang on, this is embarrassing, as it looks just fine, well kind of… Of course, the iOS Safari browser automatically draws pages onto a 980 pixel wide canvas and then squeezes that canvas down to fit the viewport area. We still have to zoom in to see areas but there's no content being clipped. How do we stop Safari and other mobile browsers from doing this?

Stopping modern mobile browsers from auto-resizing the page

Both iOS and Android browsers are based on WebKit (`http://www.webkit.org/`). These browsers, and a growing number of others (Opera Mobile, for example), allow the use of a specific meta viewport element to override that default canvas shrinking trick. The `<meta>` tag is simply added within the `<head>` tags of the HTML. It can be set to a specific width (which we could specify in pixels, for example) or as a scale, for example 2.0 (twice the actual size). Here's an example of the viewport `meta` tag set to show the browser at twice (200 percent) the actual size:

```
<meta name="viewport"  content="initial-scale=2.0,width=device-width"
/>
```

Let's stick that into our HTML as done in the following code snippet:

```
<!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="initial-scale=2.0,width=device-width"
/>
<title>And the winner isn't…</title>
```

Now, we'll load that page up in Android and see how it looks:

As you can see, this isn't exactly what we're gunning for but it illustrates the point, in a big way!

Getting the iOS and Android emulators

Although there is no substitute for testing sites on real devices, there are emulators for Android and iOS. Android emulator for Windows, Linux and Mac is available free by downloading and installing the Android Software Development Kit (SDK) at `http://developer.android.com/sdk/`. It's a command line setup; so not for the faint hearted. The iOS simulator is only available to Mac OS X users and comes as part of the Xcode package (free from the Mac App Store). Once Xcode is installed, you can access it from `~/Developer/Platforms/iPhoneSimulator.platform/Developer/Applications iOS Simulator.app`.

Let's break the above `<meta>` tag down and understand what's going on. The `name="viewport"` attribute is obvious enough. The `content="initial-scale=2.0` section is then saying, *scale the content to twice the size* (where 0.5 would be half the size, 3.0 would be three times the size and so on) whilst the `width=device-width` part tells the browser that the width of the page should be equal to `device-width`.

The `<meta>` tag can also be used to control the amount a user can zoom in and out of the page. This example allows users to go as large as three times the device width and as small as half the device width:

```
<meta name="viewport" content="width=device-width, maximum-scale=3,
minimum-scale=0.5" />
```

You could also disable users from zooming at all, although as zooming is an important accessibility tool, it's rare that it would be appropriate in practice:

```
<meta name="viewport" content="initial-scale=1.0, user-scalable=no" />
```

The `user-scalable=no` being the relevant part.

Right, we'll change the scale to 1.0, which means that the mobile browser will render the page at 100 percent of its viewport. Setting it to the device's width means that our page should render at 100 percent of the width of all supported mobile browsers. Here's the `<meta>` tag we'll be using:

```
<meta name="viewport" content="width=device-width,initial-scale=1.0"
/>
```

Looking at our page on an iPad in portrait mode now shows the content being clipped but not as if we are looking through a pair of jam-jar spectacles! This is how we want it at this point. This is progress, trust me!

Noticing that the viewport meta element is seeing increasing use, the W3C is making attempts to bring the same capability into CSS. Head over to `http://dev.w3.org/csswg/css-device-adapt/` and read all about the new `@viewport` declaration. The idea is that rather than writing a `<meta>` tag in the `<head>` section of your markup, you could write `@viewport { width: 320px; }` in the CSS instead. This would set the browser width to 320 pixels. Some browsers already support the syntax (Opera Mobile, for example), albeit by using their own vendor prefix; for example, `@-o-viewport { width: 320px; }`.

Fixing the design for different viewport widths

With our meta viewport problem fixed, no browsers are now zooming the page, so we can set about fixing the design for different viewports. In the CSS, we'll add a media query for devices such as tablets (for example, iPad) that have a viewport width of 768 pixels in portrait view (as the landscape viewport width is 1024 pixels, it renders the page fine when loaded in Landscape view).

```
@media screen and (max-width: 768px) {
  #wrapper {
    width: 768px;
  }
  #header,#footer,#navigation {
    width: 748px;
  }
}
```

Our media query is re-sizing the width of the wrapper, header, footer, and navigation elements if the viewport size is no larger than 768 pixels. The following screenshot shows how this looks like on our iPad:

I'm actually quite encouraged by that. The content now fits on the iPad display (or any other viewport no larger than 768 pixels) with no clipping. However, we need to fix the Navigation area as the links are extending off the background image and the main content area is floating below the sidebar (as it's too wide to fit in the available space). Let's amend our media query in the CSS, as demonstrated in the following code snippet:

```
@media screen and (max-width: 768px) {
  #wrapper {
    width: 768px;
  }
  #header,#footer,#navigation {
    width: 748px;
  }
  #content,#sidebar {
    padding-right: 10px;
    padding-left: 10px;
    width: 728px;
  }
}
```

Now the sidebar and content area are filling the entire page and are nicely spaced with a little padding on either side. However, this isn't very compelling viewing. I want the content first and the sidebar second (by it's nature it's a secondary area of interest). I've made another schoolboy error here, if I'm attempting to approach this design with a truly responsive design methodology.

With responsive designs, content should always come first

We want to retain as many features of our design across multiple platforms and viewports (rather than hiding certain parts with `display: none` or similar) but it's also important to consider the order in which things appear. At present, due to the order of the sidebar and main content sections of our markup, the sidebar will always want to display before the main content. It's obvious that a user with a more limited viewport should get the main content before the sidebar, otherwise they'll be seeing tangentially related content before the main content itself.

We could (and perhaps should) move our content above our navigation area, too. So that those with the smallest viewports get the content before anything else. This would certainly be the logical continuation of adhering to a "content first" maxim. However, in most instances, we'd like some navigation atop each page, so I'm happier simply swapping the order of the sidebar and content area in my HTML: making the content section come before the sidebar. For example, consider the following code:

```
<div id="sidebar">
  <p>here is the sidebar</p>
</div>
<div id="content">
  <p>here is the content</p>
</div>
```

Instead of the preceding code, we have code as follows:

```
<div id="content">
  <p>here is the content</p>
</div>
<div id="sidebar">
  <p>here is the sidebar</p>
</div>
```

Although we have altered the markup, the page still looks exactly the same in larger viewports due to the `float:left` and `float:right` properties on the sidebar and content areas. However, in the iPad, our content now appears first, with our secondary content (the sidebar) afterwards.

However, with our markup structured in the correct order I've also set about adding and altering more styles, specific to the 768 pixel wide viewport. This is what the media query now looks like:

```
@media screen and (max-width: 768px) {
  #wrapper,#header,#footer,#navigation {
    width: 768px;
    margin: 0px;
  }
  #logo {
    text-align:center;
  }
  #navigation {
    text-align: center;
    background-image: none;
```

```
      border-top-color: #bfbfbf;
      border-top-style: double;
      border-top-width: 4px;
      padding-top: 20px;
    }
    #navigation ul li a {
      background-color: #dedede;
      line-height: 60px;
      font-size: 40px;
    }
    #content, #sidebar {
      margin-top: 20px;
      padding-right: 10px;
      padding-left: 10px;
      width: 728px;
    }
    .oscarMain {
      margin-right: 30px;
      margin-top: 0px;
      width: 150px;
      height: 394px;
    }
    #sidebar {
      border-right: none;
      border-top: 2px solid #e8e8e8;
      padding-top: 20px;
      margin-bottom: 20px;
    }
    .sideBlock {
      width: 46%;
      float: left;
    }
    .overHyped {
      margin-top: 0px;
      margin-left: 50px;
    }
  }
```

Remember, the styles added here will only affect screen devices with a viewport of 768 pixels or less. Larger viewports will ignore them. Plus, because these styles are after any existing styles, they will override them where relevant. The upshot being that larger viewports get the design they got before. Devices with a 768 pixel wide viewport, look as shown in the following screenshot:

It goes without saying, we're not going to win any design awards here but with just a few lines of CSS code within a media query, we have created an entirely different layout for a different viewport. What did we do?

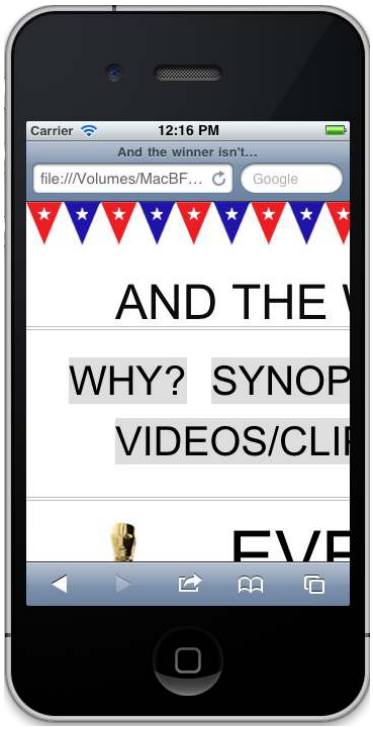
First off, we reset all the content areas to the full width of the media query, as demonstrated in the following code snippet:

```
#wrapper,#header,#footer,#navigation {
    width: 768px;
    margin: 0px;
}
```

Then it was merely a matter of adding styles to alter the aesthetic layout of the elements. For example, the following code snippet changes the navigation size, layout, and background, so that it would be easier for tablet users (or any users with a viewport of 768 pixels or less) to select a navigation item:

```
#navigation {
  text-align: center;
  background-image: none;
  border-top-color: #bfbfbf;
  border-top-style: double;
  border-top-width: 4px;
  padding-top: 20px;
}
#navigation ul li a {
  background-color: #dedede;
  line-height: 60px;
  font-size: 40px;
}
```

We now have exactly the same content displayed with a different layout depending upon viewport size. Media queries are good, no? Let's have a party. While you fetch the champagne, I'll just take a look on my iPhone to see how it looks there… You can have a look at it in the following screenshot:



Media queries—only part of the solution

Oh… best put that ice back in the freezer. Clearly our work is far from over; that looks horrible on the smaller 320 pixel wide viewport of our iPhone. Our media query is doing exactly what it should, applying styles dependent upon the features of our device. The problem is however, that the media query covers a very narrow spectrum of viewports. Anything with a viewport under 768 pixels is going to experience clipping and anything between 768 and 960 pixels will experience clipping as it will get the non-media query version of the CSS styles which, as we already know, doesn't adapt once we take it below 960 pixels wide (your author rests his head in his hands and lets out a long sigh).

We need a fluid layout

Using media queries alone to change a design is fine if we have a specific known target device; we've already seen how easy it is to adapt a device to the iPad. But this strategy has severe shortcomings; namely, it isn't really future-proof. At present, when we resize our viewport, the design snaps at the points that the media queries intervene and the shape of our layout changes. However, it then remains static until the next viewport "break point" is reached. We need something better than this. Writing CSS styles specific to each and every viewport permutation doesn't make allowances for future devices and a really great design is one with some degree of future proofing built in. At this point our solution is incomplete. This is more of an adaptive design rather than the truly responsive one we want. We need our design to flex before it snaps. To make that happen we need to move from a rigid and fixed layout to a fluid layout.

Summary

In this chapter, we've learned what CSS3 media queries are, how to include them in our CSS files, and how they can help our quest to create a responsive web design. We've also learned how to make modern mobile browsers render our pages in the same manner as their desktop counterparts and touched upon the need to consider a "content first" policy when structuring our markup. We've also learned the data economies that can be made when we use images in our design in the most economical way.

However, we've also learned that media queries can only provide an adaptable web design, not a truly responsive one. Media queries are an essential component in a responsive design but a fluid layout that allows our design to flex between the break points that the media queries handle is also essential. Creating a fluid base for our layout to smooth the transition between our media query break points is what we'll be covering in the next chapter.