9
Solving Cross-browser Responsive Challenges

In this final chapter, we will learn:

- The fundamental difference between progressive enhancement and graceful degradation
- How to make older versions of Internet Explorer responsive
- How to use Modernizr to conditionally load CSS files
- How to use Modernizr to conditionally load JavaScript polyfills
- How to change long lists of navigation to select menus on small viewports
- How to provide images for high resolution (retina) displays

Before we get to the meat of this final chapter, let's recap where we are and what we know.

Mobile usage is exploding. Consequently users view websites with a variety of viewports (different sizes and orientations) and with varying bandwidths. For the foreseeable future, we need to design and build our websites starting with the essential content and layering on features and enhancements progressively. Furthermore, due to the bandwidth considerations, the code base should be as lean and flexible as possible.

Design-wise, we've embraced all three legs of the Ethan Marcotte responsive design methodology. CSS3's media queries (covered in *Chapter* 2, *Media Queries: Supporting Differing Viewports*) are used to create design breakpoints where the layout can adapt dramatically to the viewport. Then flexible images and media alongside a fluid grid (covered in *Chapter 3*, *Embracing Fluid Layouts*) to provide a smooth flex between these media query breakpoints. The result is a design that not only works for today's popular viewports but for the future's too.

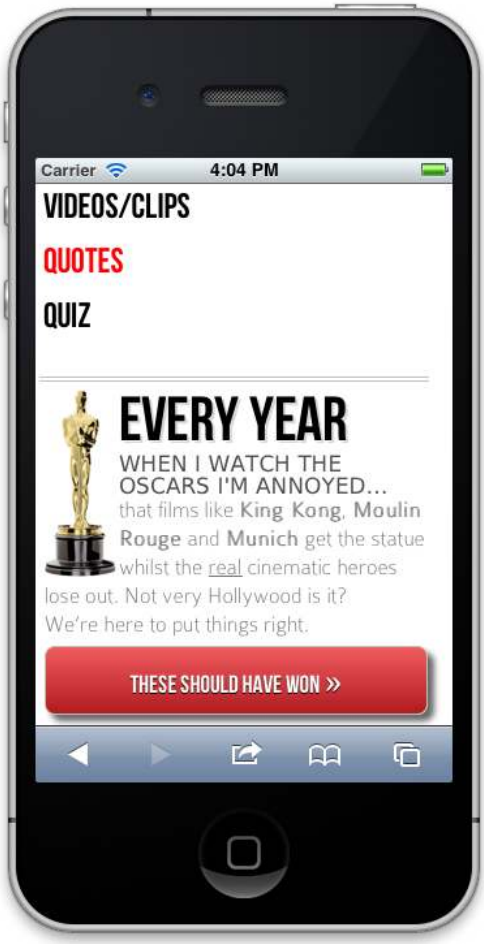
To keep our code base lean, in *Chapter 4*, *HTML5 for Responsive Designs*, we switched our markup to HTML5. It provides economies, more semantic code, and makes features such as offline viewing possible. Going further, we added some WAI-ARIA accessibility to our code, providing additional aids for screen readers and assistive technologies.

In *Chapter 5*, *CSS3: Selectors, Typography, and Color Modes* and *Chapter 6*, *Stunning Aesthetics with CSS3*, we looked at the incredible power and flexibility of CSS3, learning about new RGBA and HSLA color modes and how common design flourishes such as box-shadows, text-shadows, background gradients, and so on can be achieved without images, using CSS3 alone. In addition, the powerful selectors of CSS3 have allowed us to select anything we need from the DOM, a level of selection power that previously required JavaScript. Yet CSS3 hasn't just given us the ability to adapt the design and drastically lower the amount of bandwidth required to view our site. It has also added functionality we could never enjoy before without employing Flash of JavaScript: custom typography (*Chapter 5*, *CSS3: Selectors, Typography, and Color Modes*) and beautiful smooth transitions (*Chapter 7*, *CSS3 Transitions, Transformations, and Animations*) between different visual states. Keeping one eye on the future, we also glimpsed at sophisticated features such as CSS3 3D transformations.

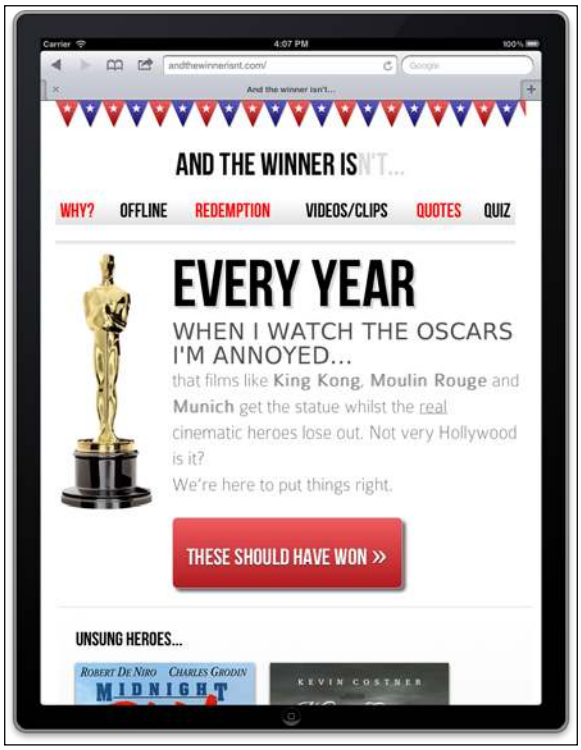
Finally, in the last chapter, we tackled the humdrum task of form building, relishing the opportunity to handle the heavy work of form validation and form UI element creation using HTML5 markup. Importantly, we also added a JavaScript fall back to conditionally enhance the experience for older browsers such as Internet Explorer versions 6, 7, and 8.

Throughout this book we've built up a fairly simple responsive website in HTML and CSS3 called *And The Winner Isn't…*. You can visit this site in your browser at `http://www.andthewinnerisnt.com`.

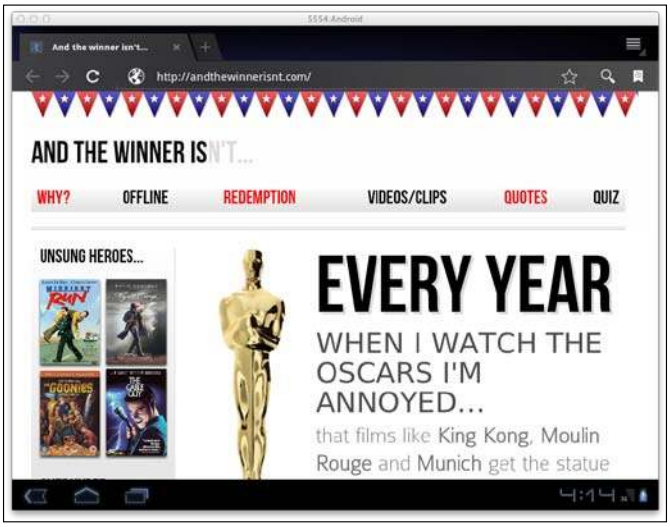
The following screenshot shows how the front page currently looks on an iPhone:



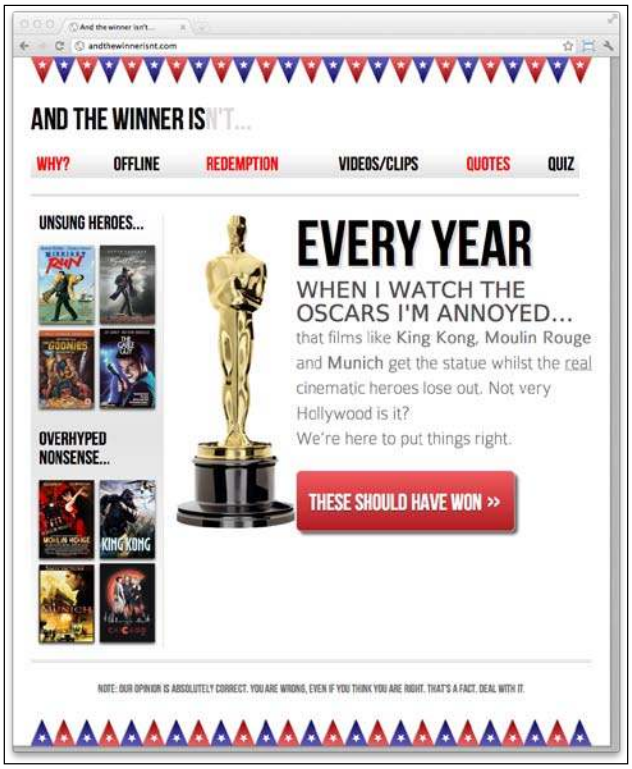
The following screenshot shows how the front page looks on an iPad:



The following screenshot shows how it looks in the Android browser (emulator):



The following screenshot shows how it looks in a modern desktop browser (Google Chrome v16):



Finally, the following screenshot shows how it looks presently in Internet Explorer 8:

Oh Momma! Pass the service revolver…

Looking at the site in Internet Explorer 8, which doesn't understand HTML5 elements, such as `<aside>`, `<header>`, `<nav>`, and `<footer>`, by default brings us to the thrust of this chapter—solving cross-browser responsive challenges.

Progressive enhancement versus graceful degradation

You're probably aware of the phrases "progressive enhancement" and "graceful degradation". These two concepts are methodologies for dealing with wide and varied browser support and spark fierce debate within the web community. Whilst initially they may seem inter-changeable terms, they are actually fundamentally opposed. Here's my take…

Graceful degradation means creating a site for modern browsers and then ensuring that certain older browsers are afforded a usable experience. Features degrade in older browsers and there is usually a cut-off point in which the oldest browsers aren't supported. There are also occasions where users are merely warned that there is a problem with their browser and workarounds are suggested (for example, "your browser is a joke—get a new one!")

Progressive enhancement is the reversal of graceful degradation. Progressive enhancement begins with markup that adheres to web standards, meaning it will be usable by all browsers (irrespective of technologies such as JavaScript and even CSS). The experience is then progressively enhanced for more capable browsers through CSS styling and eventually JavaScript (if required).

There are hundreds of articles discussing the merits and failures of each approach. For starters, I'd take a look at this piece on the Opera developer's site: `http://dev.opera.com/articles/view/graceful-degradation-progressive-enhancement/` and this excellent piece by Aaron Gustafson: `http://www.alistapart.com/articles/understandingprogressiveenhancement`.

Reality

Currently, progressive enhancement is largely considered to be the best practice way of developing a website. However, the cold hard truth is that whilst I fundamentally favor and build sites using the progressive enhancement methodology, there are plenty of instances where I am arguably doing things in a graceful degradation manner. How so?

The `www.andthewinnerisnt.com` site we have just built up uses HTML5 as its code base. Older browsers such as Internet Explorer versions 6, 7, and 8 (from this point on, also referred to as "old IE") were built and released before HTML5 (which you'll remember isn't a ratified standard despite its growing ubiquity) and so don't understand what `<aside>`, `<section>`, and `<footer>` elements are for. So, from a purist sense it could be argued I shouldn't be using HTML5 elements. By adding a piece of JavaScript to fix this basic functionality problem—is this really progressive enhancement?

Despite this, unless there is a compelling reason not to, I always opt to use HTML5 over HTML 4.01. The reality is that for the work I do on a week-in week-out basis, HTML5 offers more benefits than shortcomings. So, if using HTML5 (and I certainly recommend you do), give all devices the best shot at handling it natively by coding standards compliant HTML code (use the HTML5 validator at `http://validator.nu/` or at `http://validator.w3.org/` to eliminate any errors).

Regardless, there will inevitably be a point in which you choose (or are forced) to make some portion of the enhanced functionality afforded by modern browsers, possible in ailing versions of Internet Explorer. Maybe you want border-radius to work in old IE, for example. However, before you go there, I'm going to bend your ear just a little more…

Should you fix old versions of Internet Explorer?

At this point I'd like to re-iterate an earlier point: it's almost certainly possible to polyfill the majority of HTML5 and CSS3 features for older browsers but the resulting user experience will be heavily laden with JavaScript and potentially less usable than it would be without the polyfills. Needless to say, it's important to consider the performance implications of such a choice. Just because you can, doesn't mean you should!

Furthermore, even without polyfills (which we shall look at shortly), in my experience, adding, testing, and configuring IE specific CSS code to make IE6 and IE7 (and to a lesser extent IE8 and IE9) render pages so they look as similar as possible to a modern standards compliant browser takes at least as much time as visually enhancing a site for modern browsers—just far less enjoyable! Is that how you or your client want to spend the allocated development time?

Statistics (again)

Let's revisit some of the ground we covered in *Chapter 1, Getting Started with HTML5, CSS3, and Responsive Web Design*. Whilst conceding that statistics are always open to interpretation, we noted that from July 2010 to July 2011 global mobile browser usage (as measured by Global Stats at `http://gs.statcounter.com`) had risen (from 2.86 percent to 7.02 percent) whilst usage of Internet Explorer 7 had dropped (to 5.45 percent). For the last month of 2011, the stats are even more revealing: Internet Explorer 7 was just 4 percent with Internet Explorer 6 enjoying just 1.78 percent. Mobile browser usage meanwhile had increased to 8.04 percent.

An even more interesting fact is that for December 2011, a single modern browser, Google's Chrome (I'm including both, versions 15 and 16), accounted for 25.7 percent of global browser usage; almost the same amount accounted for by versions 6,7, and 8 of Internet Explorer (27.9 percent). Once you then factor the numbers for other modern browsers such as Safari (4.3 percent, excluding the iOS version) and all versions of Firefox (21.01 percent), and then the relevant mobile browsers, it's not difficult to appreciate that developing and enhancing the user experience for modern browsers, rather than patching up the holes in old ones makes more sense. At least to me!

The bottom line: usage of ailing versions of Internet Explorer (6, 7, and 8) is diminishing whilst usage of modern browsers (both desktop and mobile) is increasing.

Personal choice

Currently, my personal stance for new website builds is that I ensure tight visuals in the current version of Internet Explorer (v9 at the time of writing) and the nearest prior version (for example, IE8). Tweaking layout and style issues in older versions is then negotiable due to the additional time needed.

That doesn't mean I simply disregard any fundamental usability problems with versions such as IE7, I merely limit development time to ensure that basic layout and functionality works, and disregard minor alignment issues and visual enhancements that aren't supported within the browser such as background gradients, rounded corners, box-shadows, and so on. These things don't affect usability; for the most part they are merely progressive enhancements that I wouldn't expect (and nor should anyone else!) to see on aging browsers.

Testing sites across multiple browsers

Typically, standards compliant browsers, such as Chrome, Safari, and Firefox, render HTML5 and CSS3 based web pages pretty similarly. At present, the majority of smart phones (those based on Android and iOS), like their desktop Safari and Chrome counterparts, use WebKit as their base and also render pages as you would expect. However, the different versions of Internet Explorer are entirely different and there'll no doubt come a point where you'll need to check your design there too (unless it's your default browser in which case you have my sympathy). I usually use **IE Tester** (`http://www.my-debugbar.com/wiki/IETester/HomePage`)—a free utility to run multiple versions of Internet Explorer on a single machine. However, there are plenty of alternatives and this feature on Smashing Magazine gives a good overview of some common choices:

`http://www.smashingmagazine.com/2011/08/07/a-dozen-cross-browser-testing-tools/`

To illustrate this approach, after looking at `http://www.andthewinnerisnt.com` in IE8, it's obvious we've got some fundamental work to do, merely making it functional. We're going to use a great JavaScript tool called Modernizr and a polyfill to patch things up for old IE. I'm not sure that IE deserves it after all the pain it causes but that's just the kind of guy I am. However, before we get to that, let's understand Modernizr a little more.

Modernizr—the frontend developer's Swiss army knife

The web community's ability to figure out the many and varied issues of cross browser compatibility and create solutions for mere mortals like myself never ceases to amaze and delight me. Modernizr was mentioned briefly in *Chapter 4*, *HTML5 for Responsive Designs* and again in the last chapter. To reiterate, Modernizr is an open source JavaScript library that feature tests a browser's capabilities. Fauk Ateş wrote the first version, and the project now also includes Alex Sexton and the incredibly talented Paul Irish as the lead developer. It's a tool of choice for a few companies you may have heard of—Twitter, Microsoft, and Google. I mention this not merely to blow smoke up the Modernizr team (although they certainly deserve it) but more to illustrate that this isn't merely *today's* great piece of JavaScript. Put bluntly, it's a tool that is worth understanding.

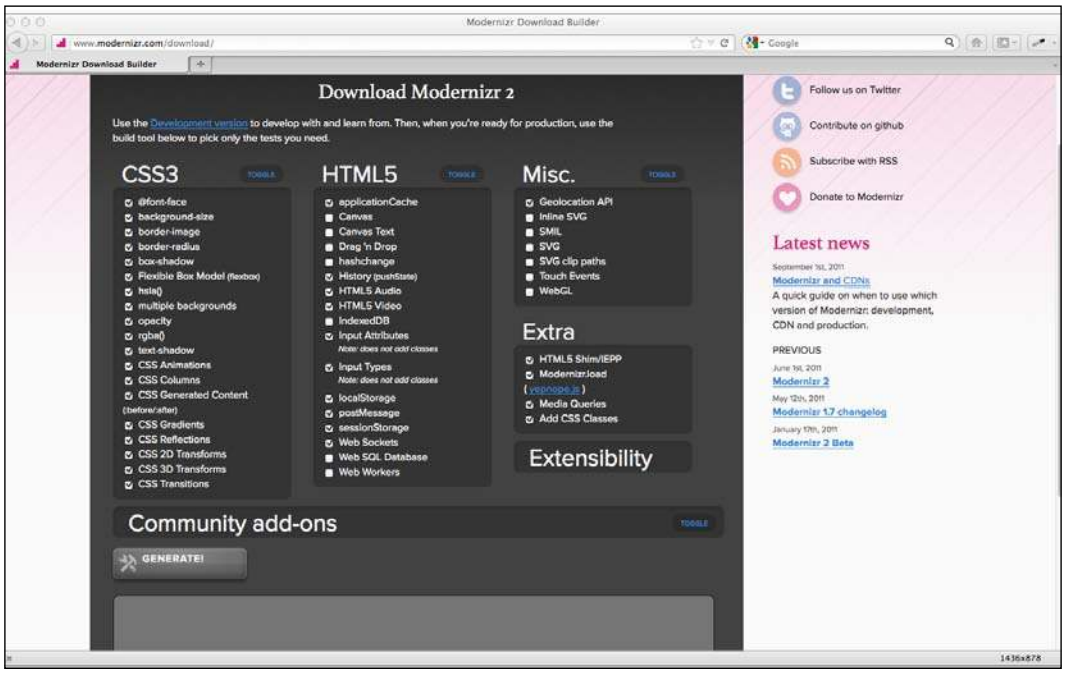
So what does it actually do? How does it enable us to both polyfill older browsers and progressively enhance the user experience for newer ones and how do we make it do what we need? Read on grasshopper...

In terms of actions, Modernizr does little, by default, other than add Remy Sharp's HTML5 shim (when selected) to enable structural HTML5 elements such as `<aside>` and `<section>` in non-HTML5 capable browsers such as IE 8 and lower versions. What it does is "feature test" the browser. Consequently, it knows whether said browser supports various features of HTML5 and CSS3. This then provides the means to take a different action depending upon that information. The rest is for us to implement. So, let's add Moderniz to our pages and make a start.

First, download Modernizr (`http://www.modernizr.com`).

Which version of Modernizr—development or production?

If you're interested in how it works, grab the development version of Modernizr as each option/test is documented. However, using the production option allows you to select only the tests that are relevant to the site or web application you are building, keeping the file nice and lean.



Now, save the file to a suitable location (as before I've used a `js` folder in the root). And then call the file in `<head>` of your page:

```
<head>
<meta charset=utf-8>
<meta name="viewport" content="width=device-width,initial-
scale=1.0,maximum-scale=1" />
<title>And the winner isn't…</title>
<link href="css/main.css" rel="stylesheet" />
<script src="js/modernizr.js"></script>
</head>
```

With Modernizr added, when viewing the source code of a page in Firebug or similar, it shows a variety of different classes added to the HTML tag. Here's an example from Firefox v9.01:

```
<html class=" js flexbox geolocation postmessage indexeddb history
websockets rgba hsla multiplebgs backgroundsize borderimage
borderradius boxshadow textshadow opacity cssanimations csscolumns
cssgradients no-cssreflections csstransforms no-csstransforms3d
csstransitions fontface generatedcontent video audio localstorage
sessionstorage applicationcache" lang="en">
```

This is great. It tells us, on a browser-by-browser basis, what features it has tested and which features the browser does or doesn't support (where there is no support for a feature, it prefixes the feature with `no-`). This lets us do two major things—fix styling issues on a feature-by-feature basis in our CSS files and also conditionally load additional CSS or JS files only when needed.

Fix styling issues with Modernizr

Our responsive *And the winner isn't…* site is presenting the perfect opportunity to fix a problem with Modernizr. Whilst the Quiz page (`http://www.andthewinnerisnt.com/3Dquiz.html`) works fine in browsers (such as Safari and Chrome) that support 3D transforms it's just a simple hover effect in browsers that don't. Currently, regardless of whether a browser can render the 3D transforms or not, we have a note telling people: This page relies on 3D transforms. If the posters don't flip on hover, try viewing in Safari or Chrome.

But thanks to Modernizr's additional classes, we now have a means of only showing a relevant note if their browser *doesn't* have the 3D transform feature.

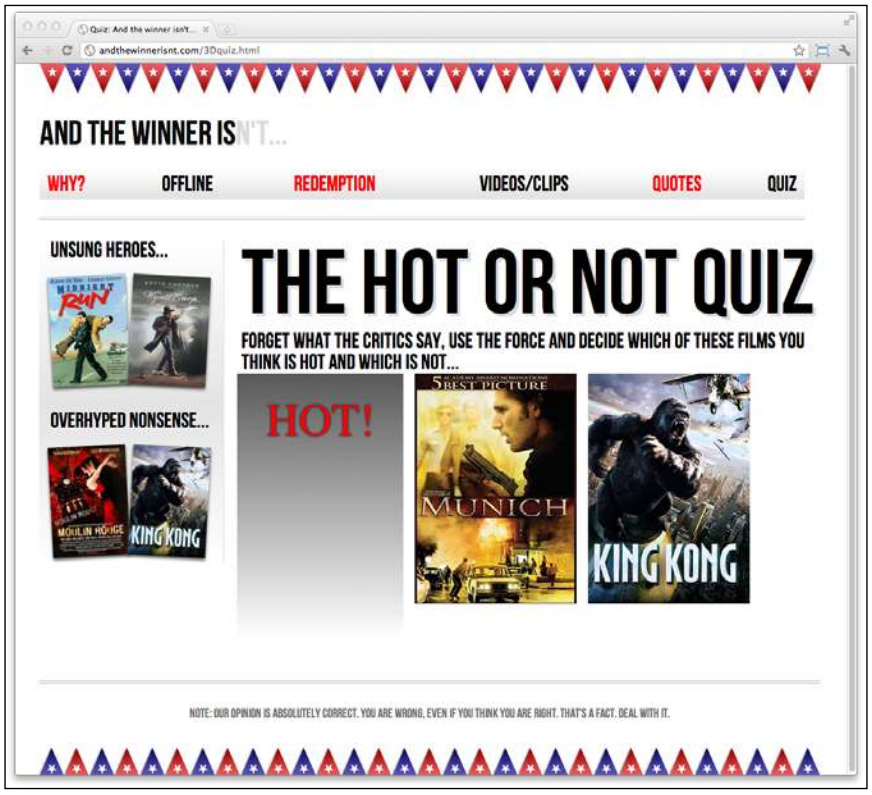
```
.note {
  display: none;
```

```
}
.no-csstransforms3d .note {
  display: block;
}
```

Breaking that down, first we set the CSS to not show the note by default:

```
.note {
  display: none;
}
```

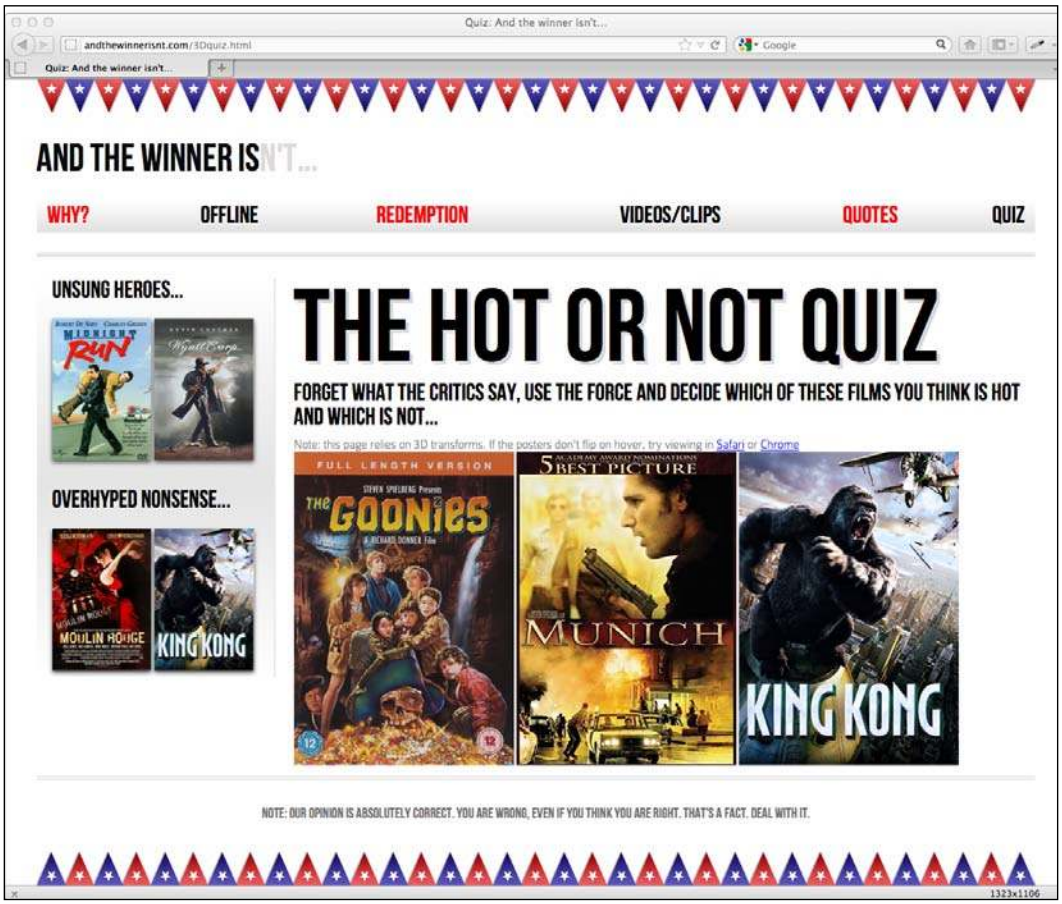
This means browsers that have the CSS 3D Transform feature (Google Chrome 16 for example) won't see the note (see the following screenshot):



Then the second rule uses the additional class added by Modernizr to show the note for browsers that don't have the 3D transforms feature:

```
.no-csstransforms3d .note {
  display: block;
}
```

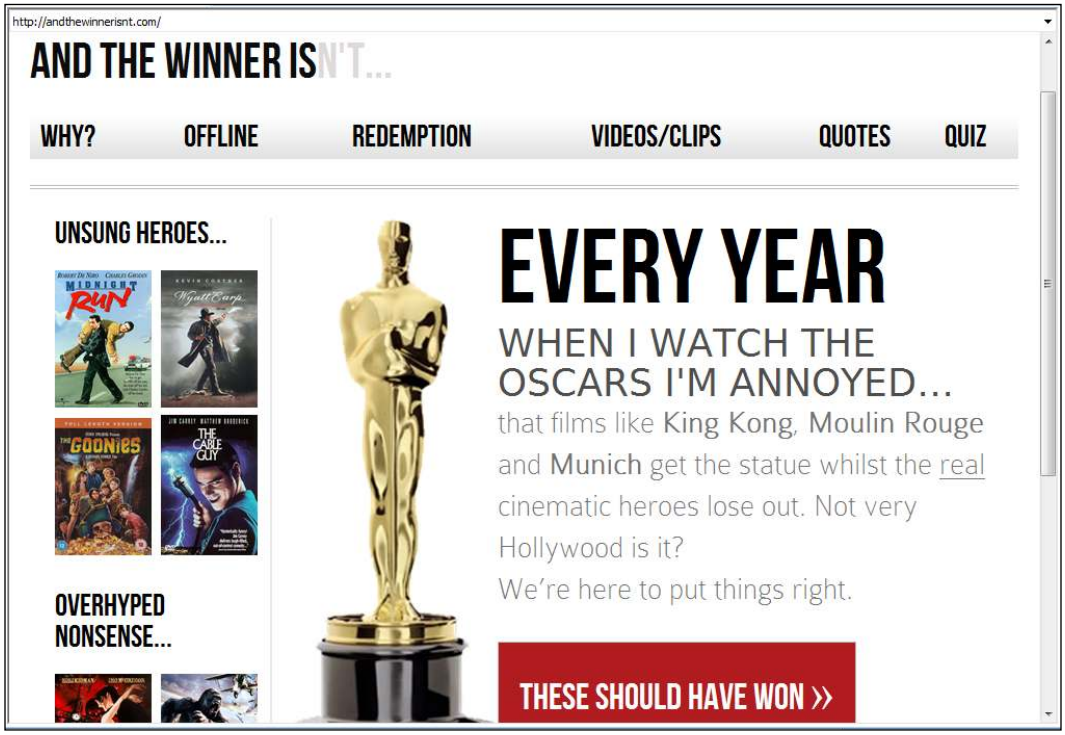
The following screenshot shows the same page in Firefox 9:



Modernizr allows us to stop thinking in terms of browsers and think in terms of features.

Modernizr adds HTML5 element support for old IE

As I've chosen a custom production version of Modernizr, that includes the HTML5 shim, refreshing the page in Internet Explorer 8 reveals a web page (as shown in the following screenshot) that looks a whole lot better than it did before:



I didn't need to do anything more. Because Modernizr has enabled HTML5 structural elements in old IE many standard CSS styles are now understood and the page renders as it should.

For my money, that is perfectly usable. If you hadn't seen the same site in a modern browser you wouldn't necessarily know anything was different. However, due to IE8's lack of support for CSS3, we know there are some obvious visual shortcomings compared to a modern browser; there are no alternate colors in the navigation links (if needed we could easily fix this by adding an extra class to odd navigation links), no rounded corners on the button, no text or box shadows and perhaps more importantly, although our fluid grid flexes, a lack of CSS3 support means no media query support. No media queries—no significant layout changes at differing viewports in Internet Explorer 6, 7, or 8.

Although I don't consider this layout "broken" in any way, a tool such as Modernizr does give us the capability to add features that polyfill older browsers as we see fit. To illustrate, let's add media query min/max-width support so that our design responds to different viewports in Internet Explorer 6, 7, and 8.

Add min/max media query capability for Internet Explorer 6, 7, and 8

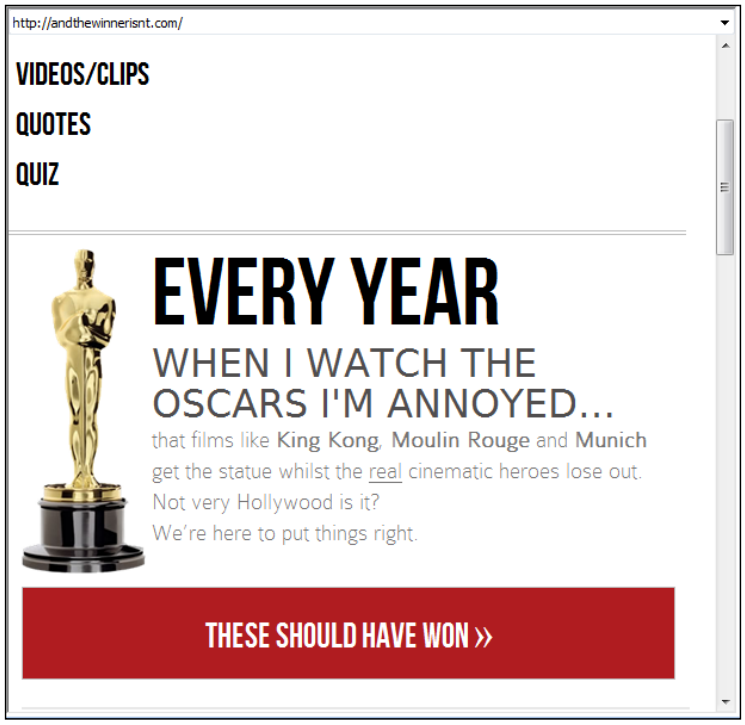
The polyfill that I generally use to add media query support to older versions of Internet Explorer only adds support for min/max-width media queries. There are more substantial media query polyfills that add a greater range of media query support but for a responsive design, **Respond.js** by Scott Jehl is simple to use, fast, and has always served me well.

Respond.js (`https://github.com/scottjehl/Respond`) can actually be used without Modernizr—just add it to the page in question, and as the author Scott Jehl himself says, "Crack open Internet Explorer and pump fists in delight".

So, before we integrate Respond.js with Modernizer, let's do just that. Drop Respond.js straight into our page (just add it after the Modernizr file we already added) and check it does what we want for IE. To do this, download the file, save it in a suitable location, and link to it in the `<head>` section:

```
<head>
<meta charset=utf-8>
<meta name="viewport" content="width=device-width,initial-
scale=1.0,maximum-scale=1" />
<title>And the winner isn't…</title>
<link href="css/main.css" rel="stylesheet" />
<script src="js/modernizr.js"></script>
<script src="js/respond.min.js"></script>
</head>
```

Now, once we load the page in Internet Explorer 8 and resize the browser window, we get our responsive design back (see the following screenshot):



Great, we've added a polyfill that sorts out min- and max-width media queries in Internet Explorer but here's the rub: this thing is now being loaded for every browser that loads the page—whether they need it or not. One solution would be to stick the script link in an IE conditional comment like the following:

```
<!--[if lte IE 8]>
        <script src="js/respond.min.js"/></script>
<![endif]-->
```

I'm sure you've come across conditional comments before. They are a simple way of loading CSS or JS files (or even content) that only the relevant version of Internet Explorer will use. All other browsers will see the code as a comment and ignore it.

In this example, our conditional comment says, "If you are *less than or equal* to (the `lte` part) Internet Explorer 8, (the `IE 8` part) do this".

All about conditional comments

Conditional comments are falling out of favor compared with feature detection but if you'd like to know more, read all about them at the following URL:

`http://msdn.microsoft.com/en-us/library/ms537512%28v=vs.85%29.aspx`

That will work fine. But do we really want to litter our markup with IE specific conditional comments? And what about polyfills for other browsers? This is where Modernizr steps up to the plate.

Conditional loading with Modernizr

A big pull of Modernizr when trying to keep websites and web applications as lean as possible is that it can load resources (both CSS and JS files) conditionally. So, rather than use a "scatter gun" approach and laden our pages with every polyfill a user *might* need (regardless of whether they actually need them or not), we only load the polyfills a user *actually* needs. This keeps our pages and load times as lean as they can be for each and every user.

So with Modernizr already added to the head of our pages, let's use it to conditionally load our Respond.js polyfill only if the browser in question doesn't natively understand CSS3 media queries (for example IE versions 6, 7, and 8).

Modernizr includes a JavaScript micro-library called **YepNope.js** (`http://yepnopejs.com/`). It uses a simple format:

```
Modernizr.load({
    test: Modernizr.mq('only all'),
    nope: 'js/respond.min.js'
});
```

First up is the call to the resource loading part of Modernizr:

```
Modernizr.load({
```

Within this is the test itself and a number of possible actions based on the result of that test. In this example, we have asked if the browser understands a media query:

```
    test: Modernizr.mq('only all'),
```

If not, the resource should load our `respond.min.js` file:

```
    nope: 'js/respond.min.js'
```

Here `only all` is the equivalent of "do you understand media queries?" Old IE will always fail the test, resulting in `nope` and therefore load the relevant resource. This enables `respond.min.js` to only be loaded when needed.

We could also opt to load additional files at the same time:

```
Modernizr.load({
    test: Modernizr.mq('only all'),
    nope: ['js/respond.min.js', 'css/extra.css']
});
```

This example uses an array to add the `respond.min.js` file and a CSS file called `extra.css`. You may opt to load CSS this way to maintain separate styles that are only needed in the presence or absence of certain features. It's worth remembering that it's also possible to load different resources based on different outcomes:

```
Modernizr.load({
    test: Modernizr.mq('only all'),
    yep: 'js/pass.js',
    nope: 'js/respond.min.js'['fail-polyfill.js', 'fail.css'],
    both: 'js/for-all.js'
});
```

Here, we load one file if the browser passes, another two (in the array) if it fails and a final file if it passes *or* fails.

The conditional loading code tests can be written in another separate JavaScript file. In this instance, I have called mine the `conditional.js` file and saved it in the `js` folder, alongside `modernizr.js` and `respond.min.js`. So, the `<head>` section now looks as follows:

```
<head>
<meta charset=utf-8>
<meta name="viewport" content="width=device-width,initial-
scale=1.0,maximum-scale=1" />
<title>And the winner isn't…</title>
<link href="css/main.css" rel="stylesheet" />
<script src="js/modernizr.js"></script>
<script src="js/conditional.js"></script>
</head>
```

Note that I've removed `respond.min.js` from the `head` as it's now loaded in conditionally as and when needed.

More documentation on how to conditionally load resources with Modernizr can be found at `http://www.modernizr.com/docs/#load`

Get your polyfills here

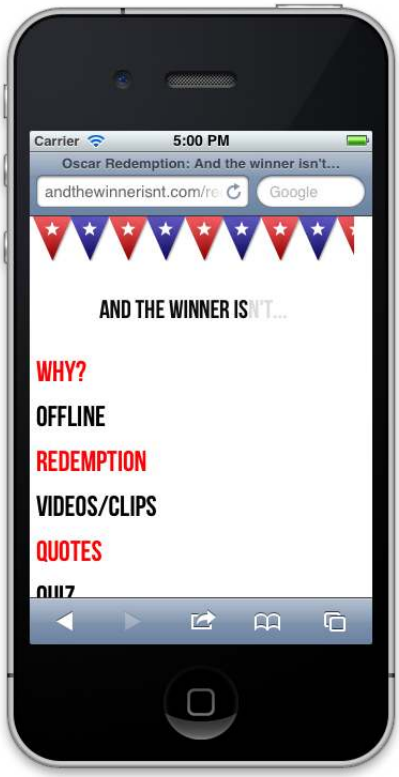
Remember, there's a great repository (pun intended) of useful polyfills at the following Github location:

`https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-browser-Polyfills`

Changing navigation links to a drop menu (conditionally)

A common issue with responsive designs is that if you have lots of navigation links on a page they can take up a sizeable portion of your screen real estate in smaller viewports.

For example, with only six page links, this is how any page currently loads for the *And the winner isn't…* website on a smaller viewport:



I'd like to swap those links out for a drop menu but only if a browser is below a certain viewport width. Now, you can roll your own piece of JavaScript to convert the menu items to a drop menu. The venerable Chris Coyier has documented how this can be achieved (`http://css-tricks.com/convert-menu-to-dropdown/`). Alternatively, there are a few pre-written scripts that do this for you. For brevity and ease, I have opted to use one such script. The following screenshot shows what the drop menu does to our navigation links on smaller viewports:

Clicking on the **Select a page** button brings up the navigation, as shown in the following screenshot:

A perfect poster child for progressive enhancement—it isn't essential functionality but it gets more content "above the fold" for users with smaller viewports. So, let's get on and implement it. First off, download the Responsive Menu script (`https://github.com/mattkersley/Responsive-Menu`). As before, save the relevant file (`jquery.mobilemenu.js`) to the `js` folder. There is just one thing we need to do first in the markup, and that's give our navigation links in each page an `id`:

```
<nav role="navigation">
  <ul id="mainNav">
    <li><a href="why.html">Why?</a></li>
    <li><a href="offline.html">Offline</a></li>
    <li><a href="redemption.html">Redemption</a></li>
    <li><a href="video.html">Videos/clips</a></li>
    <li><a href="quotes.html">Quotes</a></li>
    <li><a href="3Dquiz.html">Quiz</a></li>
  </ul>
</nav>
```

We could live without doing this but jQuery selectors work much faster with a specific `id` to latch onto.

Now, in the `conditional.js` file, we'll add the following code:

```
Modernizr.load([
  {
    test: Modernizr.mq('only all'),
    nope: 'js/respond.min.js'
  },
  {
    // load the menu convertor if max-width is 600px;
    test: Modernizr.mq('only screen and (max-width: 600px)'),
    yep : ['js/jquery-1.7.1.js', 'js/jquery.mobilemenu.js'],
    complete : function () {
      // Run this after everything in this group has downloaded
      // and executed, as well everything in all previous groups
      $(document).ready(function(){

          $('#mainNav').mobileMenu({
            switchWidth: 600,                   //width (in px to
switch at)
            topOptionText: 'Select a page',     //first option text
            indentString: '   '  //string for indenting
nested items
          });
      });

    }
  }
]);
```

After the prior conditional load that adds Respond.js for old IE, we've added another test:

```
test: Modernizr.mq('only screen and (max-width: 600px)'),
```

The previous test asks, does this viewport understand media queries and if it does, is the maximum width of the viewport 600px? If it does...

```
    yep : ['js/jquery-1.7.1.js', 'js/jquery.mobilemenu.js'],
```

The previous line loads both the jQuery library and the Responsive Menu file:

```
complete : function () {
…more code…
```

The complete section effectively says, once any files are downloaded and executed, run the following:

```
$(document).ready(function(){

          $('#mainNav').mobileMenu({
            switchWidth: 600,                    //width (in px to
switch at)
            topOptionText: 'Select a page',      //first option text
            indentString: '   '  //string for indenting
nested items
          });
});
```

These are the variables for the Responsive Menu script. Most importantly, the first option defines what viewport width I want the existing menu to be converted to a drop down (I've set it to 600px).

Again, using Modernizr to perform this task removes extraneous code for users that don't need it and allows progressive enhancement of the user experience for those that do.

For website designers, especially those unfamiliar with JavaScript, plunging into Modernizr for the first time can be daunting. There's certainly a lot to take in but hopefully this short primer will illustrate some obvious advantages that can be utilized in any future responsive project you might work on.

High resolution devices (the future)

Devices and their capabilities are changing all the time. Indeed, it isn't just different viewport sizes we must contend with. Already, we need to consider viewports that have higher resolution displays. The iPhone 4 was the first widely used device to implement a **high-resolution** display. Its screen is 960 by 640 pixel resolution at 326 pixels per inch, double the resolution of the prior version (iPhone 3GS) and double the pixel per inch density of laptops such as the 2011 15" MacBook Pro. Expect many more devices from tablets and laptops to desktop screens to follow suit. Thankfully, our responsive tools already provide us with the capabilities to support enhancements for these devices.

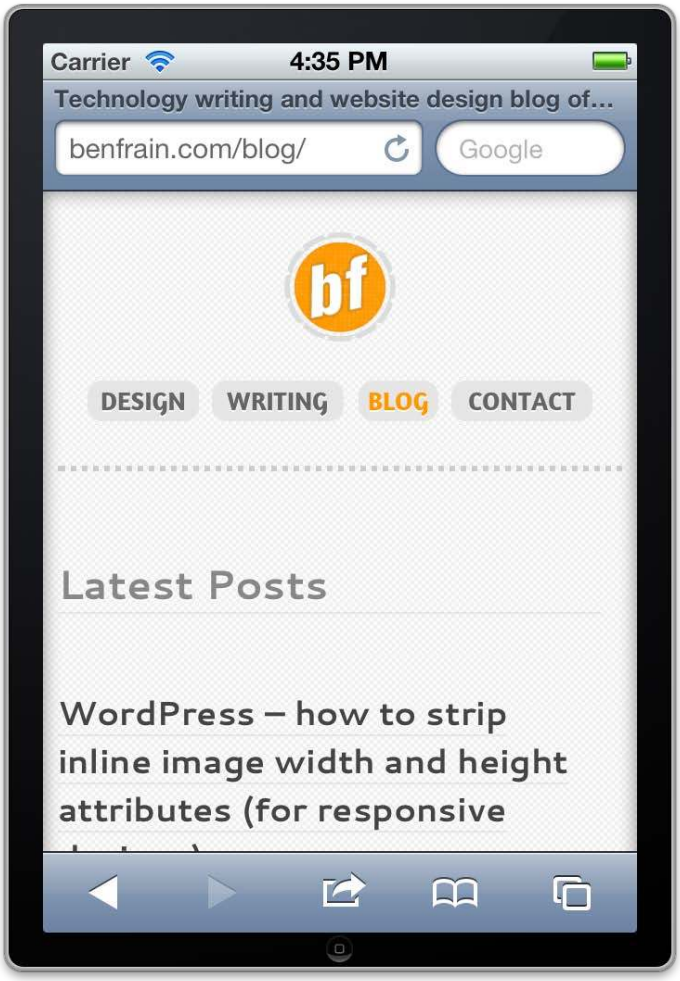
Let's suppose we wanted to load a higher resolution version of a site logo for users of high-resolution displays. It's a situation I encountered when performing a recent redesign of my own website at `http://www.benfrain.com`. Here is the markup for my logo area:

```
<div class="logo">
  <a href="http://benfrain.com/"></a>
</div>
```

And here is the CSS rule that loads the logo:

```
  #container header[role="banner"] .logo a {
    background-image: url("../img/logo2.png");
    background-repeat: no-repeat;
    background-size: contain;
    display: block;
    height: 7em;
    margin-top: 10px;
}
```

Initially, the logo looked like the one shown in the following screenshot:



Perfectly functional but I wanted the logo as crisp as possible on higher resolution displays. So, I made two further versions of my logo (one for the default state and one for the hover state) at double the size of the existing logo and named them `logo2@x2.png` and `logo2Over@x2.png`. I then added the following media query in my CSS:

```
@media all and (-webkit-min-device-pixel-ratio : 1.5) {
  #container header[role="banner"] .logo a {
    background-image: url("../img/logo2@x2.png");

  }
```

```
    #container header[role="banner"] .logo a:hover {
      background-image: url("../img/logo2Over@x2.png");
    }
  }
```

The media query targets devices with a minimum device pixel ratio of 1.5. Therefore, high-resolution displays like those on the iPhone 4 and later come into this category and render the styles within. You'll notice this rule includes a `-webkit-` prefix. As ever, remember relevant prefixes for the devices you need to target.

And now, with high-resolution devices, the higher quality version of the logo is loaded instead, as shown in the following screnshot:

Admittedly, the difference is subtle. It's probably best to look at the differences in the flesh to appreciate the difference but the more detailed the image, the more likely it is to appear beautifully crisp on a high resolution display.

There are considerations to using this technique. Larger images equate to larger file sizes and longer download times so again, just because you can, doesn't mean you should.

Where supported, **Scalable Vector Graphics** (**SVG**) alleviate many of the image scaling issues that we currently face. As the name suggests, they are designed to produce images that can display crisply at whatever scale is needed. However, media queries and SVG don't help with inline photos for high resolution displays. You'll need to implement JavaScript based solutions in those instances.

Summary

In this chapter, we've considered the fundamental differences between progressive enhancement and graceful degradation. We've then used a polyfill to make old IE understand our media queries so that our design responds there too. Finally, we used Modernizr to conditionally load CSS and JavaScript files based upon any number of feature tests, thereby allowing us to serve up polyfills and additional or alternate styles only when a browser lacks the requisite features. Finally, we've taken a sneak peek at the technologies that are becoming commonplace in the immediate future and how we can use CSS3 to serve yet further enhancements for the devices that support them.

At this point, your humble author believes (and hopes) he has related all the techniques and tools you'll need to start building your next website or web app responsively.

It's my firm conviction that currently, responsive web designs built with HTML5 and CSS3 represent the best frontend development option for the vast majority of websites. With only a little modification to our existing workflows, practices, and techniques they enable us to provide fast, flexible, and maintainable websites that can look incredible regardless of the viewport used to visit them.

As mobile device usage continues to grow exponentially, and new devices that we never before contemplated enter the browsing fray, this methodology arguably provides the surest and most future proof means of building designs that will work on any device, on any viewport, and render as quickly as possible however those devices connect to the web.