JavaScript, start with a baseline of semantic, usable markup and baseline styles. Then instruct the script to make necessary HTML and CSS changes required by the new interface once it has determined that it can run without encountering errors.

For an example of this in action, let's return to the Retreats 4 Geeks page.

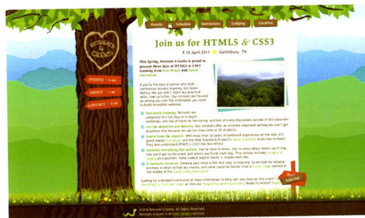

Figure 4.4: *The Retreats 4 Geeks web page.*

I wanted to make the best possible use of space on a mobile device. The horizontal navigation will work on a small browser, but the target areas would be too small to click easily. Of course, I could switch the site to use vertical navigation, allowing for larger links, but that would take up precious screen real estate.

An alternative to these two approaches is creating a dropdown using either CSS or a `select` element. While the pure CSS dropdown option is tempting, the latter approach has an edge because it provides users with a familiar user interface. For that reason, I chose the `select` route.

Based on the markup introduced back in Chapter 2, I'll walk you through creating a script that converts the contents of the

`nav` element into a `select` when the browser shrinks below a particular size. To keep the example short and a little easier to follow, I've used the jQuery JavaScript library.[9] Libraries are great tools as they are composed of dozens if not hundreds of functions that solve common problems (like adding and removing `class`es). Don't worry if you can't completely follow the code, I'll explain what's happening so you don't have to decipher it on your own.

We'll begin by isolating the script in an anonymous function[10] that runs as soon as the DOM is available, but before assets like images, CSS files, and videos have been downloaded (a.k.a. `onDOMReady`). This makes the page more responsive than running a script when the window loads (a.k.a. `window.onload`). For the remainder of this example, all of the code will be sequestered within this function:

```
$(function(){
  // Exciting stuff will go here
});
```

Next, we create the variables we need for this script to work. By instantiating them all at once, we'll reduce the number of `var` statements (which helps with minification).[11]

```
var
$window  = $(window),              // reference the window
$old_nav = $('#top nav > *'),      // get the navigation
$links   = $old_nav.find('a'),     // get the links
showing  = 'old',                  // track what's showing
trigger  = 765,                    // the browser width
                                   // that triggers the change
$new_nav, $option,                 // we'll use these shortly
timer    = null;                   // we'll need a timer too
```

9. http://jQuery.com

10. Anonymous functions are functions which have not been given a name.

11. http://www.alistapart.com/articles/javascript-minification-part-II/

The comments should give you a good sense of what each variable is for.

If you're familiar with jQuery, but confused as to why we're assigning elements to local variables rather than just referencing the jQuery-based lookup (e.g., `$('#top nav > *')`) each time we need it, rest assured that there's a method to my madness: creating a local reference reduces the performance hit of running the script because the look up only happens once instead of every time `$()` is used. Also, to make it easy to differentiate jQuery results from other variables, I've prefaced each associated variable name with a dollar sign ($). You'll see these techniques used throughout this script as they are helpful habits to get into.

With all of our variables in place, you *might* think we could move on to the meat of the script, but we're not quite ready for that yet. Before we try to execute code against the page, we should make sure that the elements we need actually exist:

```
if ( $old_nav.length && $links.length ) {
  // We know the DOM elements we need exist
  // and can do something with them
}
```

Testing for dependencies is very important and is something I'll cover more thoroughly in the next section. Now for the meat (or nutmeat if you're a vegetarian). We'll begin our script in earnest by generating the new `select`-based navigation, creating the `select`, and the first of several `option` elements it will contain:

```
$new_nav = $('<select></select>');
$option = $('<option>-- Navigation --</option>')
  .appendTo($new_nav);
```

With new markup to work with, we can now loop through the links we collected (as `$links`) and build a new option for each by repeatedly cloning the `option` we just created:


```
$links.each(function(){
  var $a = $(this);
  $option.clone()
    .attr( 'value', $a.attr('href') )
    .text( $a.text() )
    .appendTo( $new_nav );
});
```

With the `options` created and appended to the `select` we can move on to adding the final markup touches and setting up the event handler for the `select`'s `onchange` event:

```
$new_nav = $new_nav
  .wrap('<div id="mobile-nav"/>')
  .parent()
  .delegate('select', 'change', function(){
    window.location = $(this).val();
  });
```

This is a slightly simplified version of what you'll find on the live Retreats 4 Geeks site (I've taken out some of the URL hash trickery), but I wanted to make sure you were able to follow it without distraction. Here's what's going on: the first three lines wrap our `select ($new_nav)` in a `div` and then re-assign that `div` to the variable `$new_nav` so the whole thing is viewed by JavaScript as a neat little package; the next line uses event delegation (which we discussed earlier) to observe the `onchange` event on the `select` from further up the DOM tree (from the `div`, in fact), assigning an anonymous function to that event that pushes a new location to the browser's address bar (causing the browser to jump to the new section or load a new page, depending on the link type).

Boom! Functional `select`-based navigation. Now to get it into the page when conditions are right. For that, we'll create a new function, called `toggleDisplay`, that will observe the size of the browser window and handle swapping one navigation style for another:

```
function toggleDisplay() {
  var width = $window.width();
  if ( showing == 'old' && width <= trigger ) {
    $old_nav.replaceWith($new_nav);
    showing = 'new';
  } else if ( showing == 'new' && width > trigger ) {
    $new_nav.replaceWith($old_nav);
    showing = 'old';
  }
}
```

Again, this is a slightly simplified version of the final script, but it highlights the important part: the navigation is only swapped in the event that the appropriate browser width threshold is met (`trigger`) *and* the other navigation style is showing (tracked using `showing`). With that function in place, we just need to run it once (to initialize everything and make sure the right navigation is showing from the get-go) and then assign it to the window's `onresize` event:

```
toggleDisplay(); // initialize the right view
$window.resize(function(){
  if ( timer ) { clearTimeout(timer); }
  timer = setTimeout( toggleDisplay, 100 );
});
```

If you're wondering why `toggleDisplay()` isn't passed in as the actual event handler, that's because doing so would cause the function to be executed numerous times (possibly several hundred) while a user is resizing his or her browser. To keep the number of executions to a minimum (and reduce the burden the script places on a user's CPU), the event handler uses a timer to call `toggleDisplay()` after .1 seconds. As the function is triggered repeatedly during a resize event, it destroys the timer if it exists and then recreates it. This setup ensures `toggleDisplay()` is only called once when a user resizes his or her browser (unless he or she does so *very* slowly).

And there you have it: a perfect example of progressive enhancement with JavaScript.

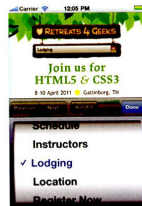

Figure 4.5: `select`*-based navigation on an iPhone.*

As this simple example demonstrates, JavaScript is perfectly capable of generating everything it needs and getting rid of anything it doesn't. You could even take this particular function a step farther and make it even more markup agnostic by allowing the root starting point (in our case, the child elements of `nav`) to be passed dynamically into the function. But I leave that to you to experiment with. Onward!

KEEP IT COPACETIC

As we've covered, many of the progressive enhancement techniques available to us in HTML and CSS are pretty straightforward and may even have been part of your repertoire prior to picking up this book. Progressive enhancement with JavaScript, on the other hand, is a bit more complicated; JavaScript cannot be fault tolerant like the others because it is a programming language.