

/*

CSS3

Pseudo-Classes

Advanced CSS Selectors

CSS3 Keyframe Animations

Specificity and Inheritance

Design for Mobile

Responsive Web Design

MASTERING

CSS3

FOR WEB DEVELOPERS

CSS3 Media Queries

Rich Web Typography

CSS3 vs. CSS

Handling Old Browsers

/*

Imprint

Published in August 2011

Smashing Media GmbH, Freiburg, Germany

Cover Design: Ricardo Gimenes

Editing: Thomas Burkert

Proofreading: Brian Goessling

Concept: Sven Lennartz, Vitaly Friedman

Founded in September 2006, [Smashing Magazine](#) delivers useful and innovative information to Web designers and developers. Smashing Magazine is a well-respected international online publication for professional Web designers and developers. Our main goal is to support the Web design community with useful and valuable articles and resources, written and created by experienced designers and developers.

ISBN: 9783943075137

Version: July 25, 2011

Table of Contents

[Preface](#)

[Why We Should Start Using CSS3 and HTML5 Today](#)

[CSS Three — Connecting the Dots](#)

[Modern CSS Layouts: The Essential Characteristics](#)

[Modern CSS Layouts, Part 2: The Essential Techniques](#)

[How to Use CSS3 Pseudo-Classes](#)

[Taming Advanced CSS Selectors](#)

[!important CSS Declarations: How and When to Use Them](#)

[An Introduction to CSS3 Keyframe Animations](#)

[CSS Specificity and Inheritance](#)

[How to Use CSS3 Media Queries to Create a Mobile Website](#)

[Responsive Web Design: What It Is and How to Use It](#)

[The Future of CSS: Experimental CSS Properties](#)

[Technical Web Typography: Guidelines and Techniques](#)

[The Future of CSS Typography](#)

[Using CSS3: Older Browsers and Common Considerations](#)

[The Authors](#)

Preface

Many Web designers are reluctant to embrace the new technologies such as CSS3 or HTML5 because of the lack of full cross-browser support for these technologies. Many designers are complaining how this situation is holding them back and tying their hands.

But the day of full cross-browser support is never truly going to dawn and deliver us this wonderful new Web where our work looks the same with any Web browser. Some users will still use older browsers and some users will still have browsers with deactivated JavaScript or images. Sometimes it feels that we are hiding behind the lack of cross-browser compatibility to avoid learning new techniques that would actually dramatically improve our workflow.

This Smashing eBook *“Mastering CSS for Web Developers”* is created to help Web designers embracing the Web’s flexibility and using CSS techniques that work today. This eBook contains 15 articles that cover useful techniques, tricks and guidelines from experts on topics such as modern CSS layouts, responsive Web design, CSS typography, CSS cross-browser compatibility, as well as many other advanced CSS techniques.

The articles have been published on Smashing Magazine in 2010 and 2011, and they have been carefully edited and prepared for this eBook.

We hope that you will find this eBook useful and valuable. We are looking forward to your feedback on [Twitter](#) or via our [contact form](#).

— Thomas Burkert, Smashing eBook Editor

Why We Should Start Using CSS3 and HTML5 Today

Vitaly Friedman

For a while now, we have taken notice of how many designers are reluctant to embrace the new technologies such as CSS3 or HTML5 because of the lack of full cross-browser support for these technologies. Many designers are complaining about the numerous ways how the lack of cross-browser compatibility is effectively holding us back and tying our hands — keeping us from completely being able to shine and show off the full scope of our abilities in our work. Many are holding on to the notion that once this push is made, we will wake to a whole new Web — full of exciting opportunities just waiting on the other side. So they wait for this day. When in reality, they are effectively waiting for Godot.

Just like the elusive character from Beckett's classic play, this day of full cross-browser support is not ever truly going to find its dawn and deliver us this wonderful new Web where our work looks the same within the window of any and *every* Web browser. Which means that many of us in the online reaches, from clients to designers to developers and on, are going to need to adjust our thinking so that we can realistically approach the Web as it is now, and more than likely how it will be in the future.

Sometimes it feels that we are hiding behind the lack of cross-browser compatibility to avoid learning new techniques that would actually dramatically improve our workflow. And that's just wrong. Without an adjustment, we will continue to undersell the Web we have, and the

landscape will remain unexcitingly stale and bound by this underestimation and mindset.

Adjustment in Progress

Sorry if any bubbles are bursting here, but we have to wake up to the fact that full cross-browser support of new technologies is just not going to happen. Some users will still use older browsers and some users will still have browsers with deactivated JavaScript or images; some users will be having weird view port sizes and some will not have certain plugins installed.

But that's OK, really.

The Web is a damn flexible medium, and rightly so. We should embrace its flexibility rather than trying to set boundaries for the available technologies in our mindset and in our designs. The earlier we start designing with the new technologies, the quicker their wide adoption will progress and the quicker we will get by the incompatibility caused by legacy browsers. More and more users are using more advanced browsers every single day, and by using new technologies, we actually encourage them to switch (if they can). Some users will not be able to upgrade, which is why our designs should have a basic fallback for older browsers, but it can't be the reason to design only the fallback version and call it a night.

:select[ivizr]



CSS3 selectors for IE

selectivizr is a JavaScript utility that emulates CSS3 pseudo-classes and attribute selectors in Internet Explorer 6-8. Simply include the script in your pages and selectivizr will do the rest.



Enhancing IE's selector engine

Selectivizr adds support for 19 CSS3 pseudo-classes, 2 pseudo-elements and every attribute selector to older versions of IE. It can also fix a few of the browsers native selector implementations.



JavaScript-knowledge: none

Selectivizr works automatically so you don't need any JavaScript knowledge to use it — you won't even have to modify your style sheets. Just start writing CSS3 selectors and they will work in IE.



Works with existing tools

Selectivizr requires a JavaScript library to work. If your website already uses one of the 7 supported libraries you just need to add the selectivizr script to your pages. If not, you will need to pick a library too.

[Selectivizr](#) is one of the many tools that make it possible to use CSS3 today.

There are so many remarkable things that we, designers and developers, can do today: be it responsive designs with CSS3 media queries, rich Web typography (with full support today!) or HTML5 video and audio. And there are so many useful tools and resources that we can use right away to incorporate new technologies in our designs while still supporting older browsers. There is just no reason *not* to use them.

We are the ones who can push the cross-browser support of these new technologies, encouraging and demanding the new features in future browsers. We have this power, and passing on it just because we don't feel

like there is no full support of them yet, should not be an option. We need to realize that we are the ones putting the wheels in motion and it's up to us to decide what will be supported in the future browsers and what will not.

More exciting things will be coming in the future. We should design for the future and we should design for today — making sure that our progressive designs work well in modern browsers and work fine in older browsers. The crucial mistake would be clinging to the past, trying to work with the old nasty hacks and workarounds that will become obsolete very soon.

We can continue to cling to this notion and wait for older browsers to become outdated, thereby selling ourselves and our potential short, or we can adjust our way of thinking about this and come at the Web from a whole new perspective. One where we understand the truth of the situation we are faced with. That our designs are not going to look the same in every browser and our code will not render the same in every browser. And that's the bottom line.



Yaili's beautiful piece [My CSS Wishlist on 24ways](#). Articles like these are the ones that push the boundaries of Web design and encourage more innovation in the industry.

Andy Clarke spoke about this at the DIBI Conference earlier this year (you can check his presentation [Hardboiled Web Design on Vimeo](#)). He really struck a nerve with his presentation, yet still we find so many stalling in this dream of complete Web standardization. So we wanted to address this issue here and keep this important idea being discussed and circulated. Because this waiting is not only hurting those of us working with the Web,

but all of those who use the Web as well. Mainly through this plethora of untapped potential which could improve the overall experience across the spectrum for businesses, users and those with the skills to bring this sophisticated, rich, powerful new Web into existence.

For Our Clients

Now this will mean different things for different players in the game. For example, for our clients this means a much more developed and uniquely crafted design that is not bound by the boxes we have allowed our thinking to be contained in. However, this does come with a bit of a compromise that is expected on the parts of our clients as well. At least it does for this to work in the balanced and idealized way these things should play out. But this should be expected. Most change does not come without its compromises.

In this case, our clients have to accept the same truism that we do and concede that their projects will not look the same across various browsers. This is getting easier to convince them of in these times of the expanding mobile market, but they may still not be ready to concede this inch on the desktop side of the coin. Prices might be adjusted in some cases too, and that may be another area that the clients are not willing to accept. But with new doors being opened and more innovation, comes more time and dedicated efforts. These are a few of the implications for our clients, though the expanded innovation is where we should help them focus.

In short:

- Conceding to the idea that the project will not be able to look the same across various browsers,
- This means more developed and unfettered imaginative designs for our clients
- This could lead to increased costs for clients as well, but with higher levels of innovation
- Client's visions for what they want will be less hindered by these limitations

For the Users

The users are the ones who have the least amount invested in most of what is going on behind the scenes. They only see the end result, and they often do not think too much about the process that is involved which brings it to the screens before them. Again, with the mobile market, they have already come across the concept of varying interfaces throughout their varied devices. They only care about the functionality and most probably the style that appeals to them — but this is where their interest tends to end. Unless of course, they too are within the industry, and they may give it a second thought or more. So all this talk of cross-browser compatibility really doesn't concern them, they really leave all that up to us to worry about.

Users only ever tend to notice anything if and when something does not work the way they expect it to from one place to the next. In most cases, they are willing to show something to a relative, friend or colleague, and suddenly from one device to the next, something is different that disrupts their ability to do so. That is when they actually take notice. But if we have

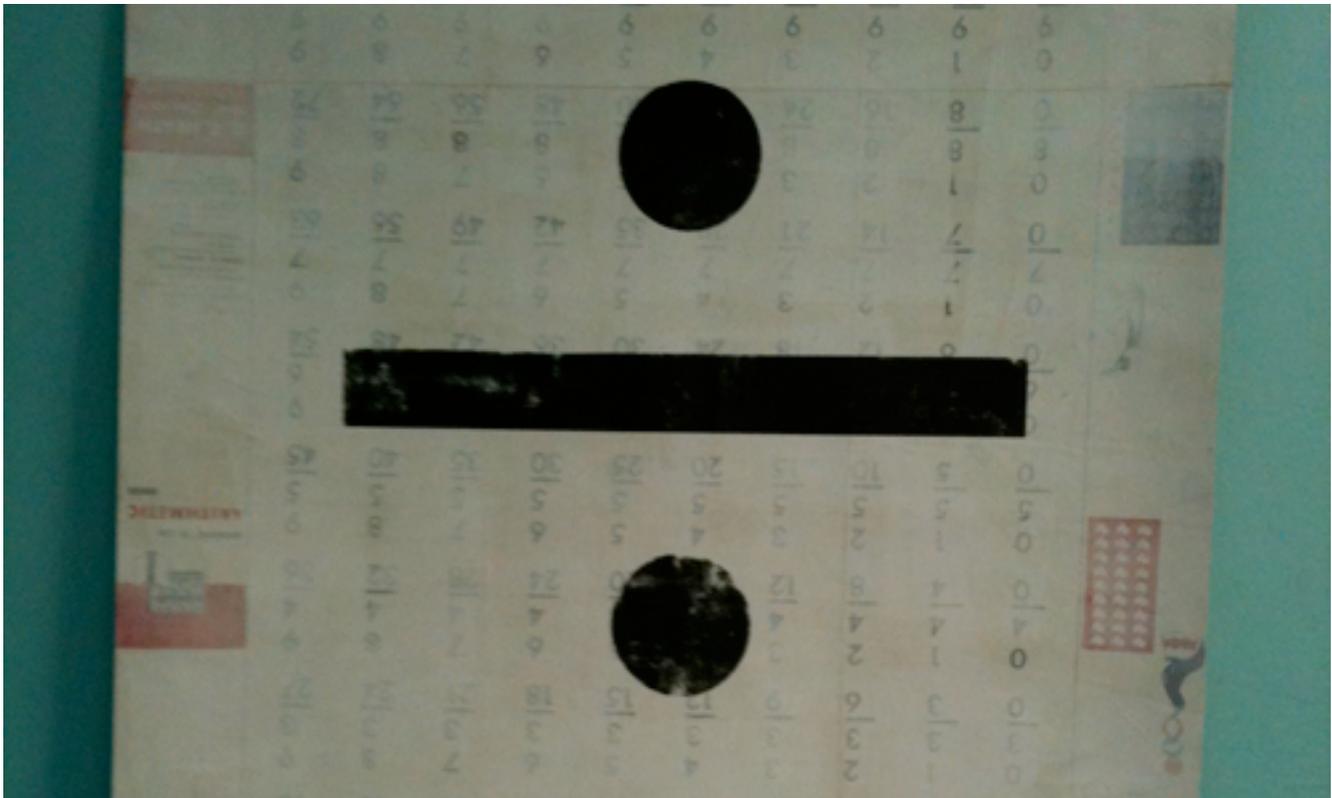
done our jobs correctly, these transitions will remain smooth — even with the pushing of the envelopes that we are doing. So there is not much more that is going to change for the users other than a better experience. An average user is not going to check if a given site has the same rounded corners and drop-shadow in two different browsers installed on the user's machine.

In short:

- Potentially less disruptions of experience from one device to another
- An overall improved user experience

For Designers/Developers

We, the designers and developers of the Web, too have to make the same concession our clients do and surrender the effort to craft the same exact presentation and experience across the vast spectrum of platforms and devices. This is not an easy idea to give up for a lot of those playing in these fields, but as has been already mentioned, we are allowing so much potential to be wasted. We could be taking the Web to new heights, but we allow ourselves to get hung up on who gets left behind in the process — and as a result we all end up getting left behind. Rather than viewing them as separate audiences and approaching them individually, so to speak, we allow the limitations of one group to limit us all.



Perhaps a divide and conquer mentality should be employed. [Image Credit](#)

So this could mean a bit more thought for the desired follow through, and we are not suggesting that we strive to appease one group here and damn the rest. Instead, we should just take a unified approach, designing for those who can see and experience the latest, and another for those who cannot. It wouldn't mean more work if we design with those users in mind and produce meaningful and clean code up front and then just adjust it for older browsers. Having to remember that not everyone is afforded the privilege of choosing which browser they are using. And if necessary, this approach can be charged for. So it could lead to more revenue along with exciting new opportunities — by bringing some of the fun back into the work that being boxed in with limitations has robbed us of.

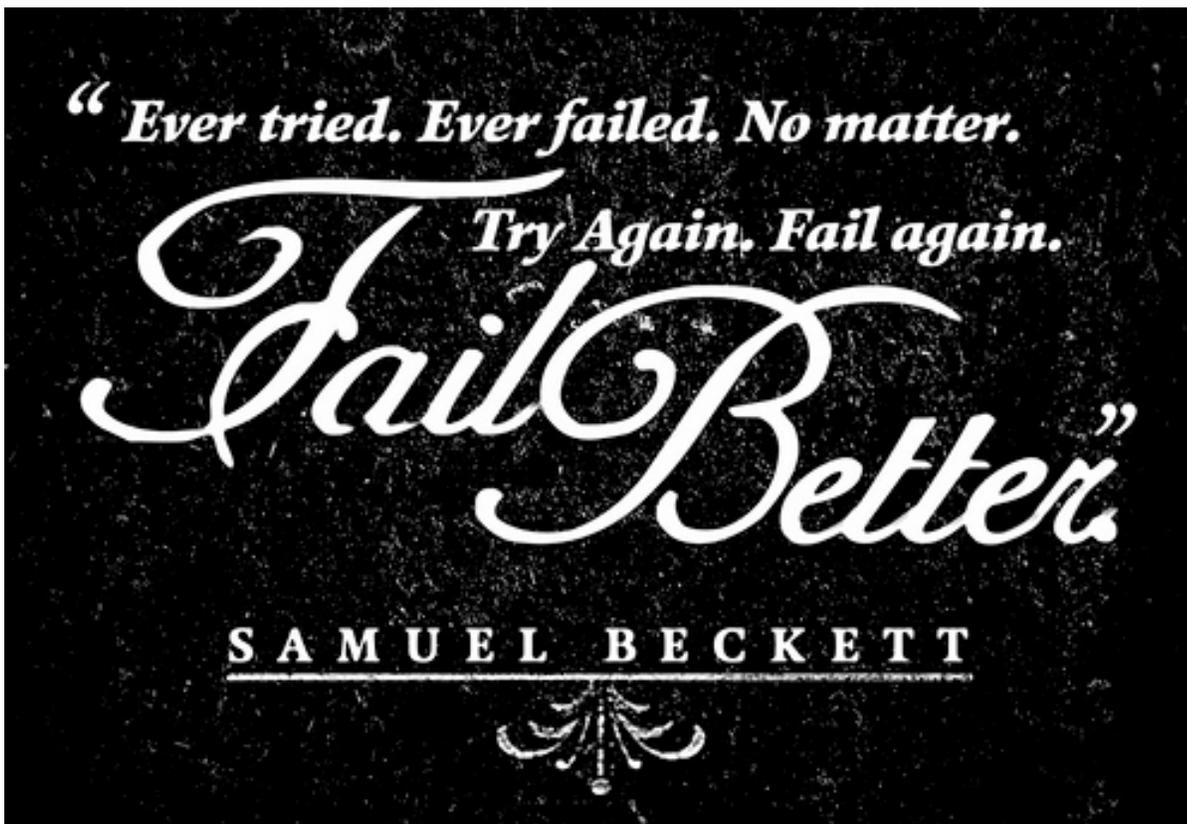
In short:

- Conceding to the idea that the project will not be able to look the same across various browsers
- A more open playing field for designers and developers all around; less restricted by this holding pattern
- More exciting and innovative landscape to attract new clientele
- Division of project audience into separate presentational approaches
- Probably less work involved because we don't need the many hacks and workarounds we've used before

So What Are We Waiting For?

So if this new approach, or adjusted way of thinking, can yield positive results across the browsers for everyone involved, then why are we still holding back? What is it that we are waiting for? Why not cast off these limitations thrown upon our fields and break out of these boxes? The next part of the discussion tries to figure out some of the contributing factors that could be responsible for keeping us restrained.

Fear Factor



The fail awaits, and so some of us opt to stay back. Image by [Ben Didier](#)

One contributing factor that has to be considered, is perhaps that we are being held back out of fear. This might be a fear of trying something new, now that we have gotten so comfortable waiting for that magic day of compatibility to come. This fear could also stem from not wanting to stand up to some particular clients and try to make them understand this truism of the Web and the concessions that need to be made — with regards to consistent presentation across the browsers. We get intimidated, so to speak, into playing along with these unrealistic expectations, rather than trusting that we can make them see the truth of the situation. Whatever the cause is that drives this factor, we need to face our fears and move on.

It's our responsibility of professionals to deliver high-quality work to our clients and advocate on and protect user's interests. It's our responsibility to confront clients when we have to, and we will have to do it at some point anyway, because 100% cross-browser compatibility is just not going to happen.

Comfortable Factor

A possible contributing factor that we should also look into is that some people in the community are just too comfortable with how we design today and are not willing to learn new technology. There are those of us who already tire of the extra work involved in the testing and coding to make everything work as it is, so we have little to no interest at all in an approach that seemingly calls for more thought and time. But really, if we start using new technologies today, we will have to master a learning curve first, but the advantages are certainly worth our efforts. We should see it as the challenge that will save us time and deliver better and cleaner code.

To some extent, today we are in the situation in which we were in the beginning of 2000s; at those times when the emergence and growing support of CSS in browsers made many developers question their approach to designing Web sites with tables. If the majority of designers passed on CSS back then and if the whole design community didn't push the Web standards forward, we probably still would be designing with tables.

Doubt Factor

Doubt is another thing we must consider when it comes to our being in hold mode, and this could be a major contributor to this issue. We begin to doubt ourselves and our ability to pull off this innovative, boundary

pushing-kind-of-work, or to master these new techniques and specs, so we sink into the comfort of playing the waiting game and playing it safe with our designs and code. We just accept the limitations and quietly work around them, railing on against the various vendors and the W3C. We should take the new technologies as the challenge to conquer; we've learned HTML and CSS 2.1 and we can learn HTML5 and CSS3, too.

Faith Factor



Faith can be a good thing, but in this case, it can hold you back. Image by [fotologic](#)

Undoubtedly, some of us are holding off on moving forward into these new areas because we are faithfully clinging to the belief that the cross-browser support push will eventually happen. There are those saying that we will be

better off as a community if we allowed the Web to evolve, and that this evolution should not be forced.

But this is not forcing evolution, it is just evolution. Just like with Darwin's theory, the Web evolves in stages, it does not happen for the entire population at once. It is a gradual change over time. And that is what we should be allowing to happen with the Web, gradually using and implementing features for Web community here and there. This way forward progress is happening, and nobody should be held back from these evolutionary steps until we all can take them.

“It's Too Early” Factor

Another possible contributor is the ever mocking “It's too early” factor. Some members of the online community faithfully fear that if they go ahead and accept this new way forward and begin designing or developing in accordance, then as soon as they begin completing projects, the support might be dropped and they would need to update the projects they already completed in the past. It's common to think that it's just too early to work with new standards until they are fully implemented in many browsers; because it's just not safe to assume that they will be implemented at all.

However, one needs to understand the difference between two groups of new features: the widely accepted ones (CSS3's media queries, border-radius, drop-shadows and HTML5 canvas are not going to disappear) and the experimental ones. The widely accepted features are safe to use and they will not disappear for certain; the experimental features can always be extracted in a separate stylesheet and be easily updated and maintained when necessary. It might be a good idea not to use experimental,

unsupported features in large corporate designs unless they are not affecting the critical design elements of the design.

Validation Factor

We cannot forget to mention that there are also many of us who are refusing to dabble in these new waters simply due to the fact that implementing some of these techniques or styles would cause a plethora of vendor-specific prefixes to appear in the stylesheet, thus impeding the validation we as professionals strive for.

Many of us would never put forth any project that does not fully validate with the W3C, and until these new specs are fully standardized and valid, we are unwilling to include them in their work. And because using CSS3 usually means using vendor-specific prefixes, we shouldn't be using CSS3. Right?



The image is a screenshot of a web browser showing a page from NetTuts+. The top navigation bar includes the 'net tuts+' logo, a search box, and links for Home, Tutorials, Articles, Tips, Freebies, Basic, Videos, Premium, and a SUBSCRIBE button. Below the navigation bar, the article 'But it Doesn't Validate' is displayed. The article is categorized under 'Articles | General' and was written by Jeffrey Way on Nov 22nd 2010 with 98 comments. The article's content begins with: 'I have a small pet peeve that I'm going to share with you. On the nights when I finish up a new CSS3 tutorial for NetTuts+ — typically while listening to my favorite Biebster songs — I click publish, and then wait to see how long it'll take before a reader leaves a comment containing the phrase, "But it doesn't validate."'

Jeffrey Way's article [But It Doesn't Validate](#)

Well, not quite. As Jeffrey Way perfectly explains in his article [But it Doesn't Validate](#), validation is not irrelevant, but the final score of the CSS validator might be. As Jeffrey says,

"This score serves no higher purpose than to provide you with feedback. It neither contributes to accessibility, nor points out best-practices. In fact, the validator can be misleading, as it signals errors that aren't errors, by any stretch of the imagination.

[...] Validation isn't a game, and, while it might be fun to test your skills to determine how high you can get your score, always keep in mind: it doesn't matter. And never, ever, ever compromise the use of the latest doctype, CSS3 techniques and selectors for the sake of validation."

— Jeffrey Way

Having our work validate 100% is not always the best for the project. If we make sure that our code is clean and accessible, and that it validates without the CSS3/HTML5-properties, then we should take our work to the next level, meanwhile sacrificing part of the validation test results. We should not let this factor keep us back. If we have a chance for true innovation, then we shouldn't allow ourselves to be restrained by unnecessary boundaries.

All in All...

Whatever the factors that keep us from daring into these new CSS3 styles or new HTML5 coding techniques, just for a tangible example, need to be gotten over. Plain and simple. We need to move on and start using CSS3 and HTML5 today. The community will become a much more exciting and innovative playground, which in turn will improve experiences for as well as

draw in more users to this dynamic new Web, which in turn will attract more clientele — effectively expanding the market. This is what could potentially be waiting on the other side of this fence that we are timidly facing — refusing to climb over it. Instead, waiting for a gate to be installed.

Only once we get past we get passed this limited way of looking at the situation, only then will we finally stop falling short of the full potential of ourselves and our field. Are there any areas that you would love to be venturing into, but you are not because of the lack of complete cross browser compatibility? Admittedly, I was a faith factor member of the community myself — how about you? And what CSS3 or HTML5 feature are you going to incorporate into your next design?

CSS Three — Connecting the Dots

Trent Walton

As a Web community, we've made a lot of exciting progress in regards to CSS3. We've put properties like `text-shadow` & `border-radius` to good use while stepping into using `background-clip` and visual effects like transitions and animations. We've also spent a great deal of time debating how and when to implement these properties. Just because a property isn't widely supported by browsers or fully documented at the moment, it doesn't mean that we shouldn't be working with it. In fact, I'd argue the opposite.

Best practices for CSS3 usage need to be hashed out in blog posts, during spare time, and outside of client projects. Coming up with creative and sensible ways to get the most out of CSS3 will require the kind of experimentation wherein developers gladly trade ten failures for a single success. Right now, there are tons of property combinations and uses out there waiting to be discovered. All we have to do is connect the dots. It's time to get your hands dirty and innovate!

Where do I start?

One of my favorite things to do is to scan a list of CSS properties and consider which ones might work well together. What would be possible if I was to connect `@font-face` to `text-shadow` and the `bg-clip:text` property? How could I string a `webkit-transition` and `opacity` together in a creative way? Here are a few results from experiments I've

done recently. While some may be more practical than others, the goal here is to spark creativity and encourage you to connect a few dots of your own.

Note: While Opera and Firefox may soon implement specs for many of the CSS3 properties found here, some of these experiments will currently only work in Webkit-browsers like Google Chrome or Safari.

Example #1: CSS3 Transitions

A safe place to start with CSS3 visual effects is transitioning a basic CSS property like `color`, `background-color`, or `border` on hover. To kick things off, let's take a link color CSS property and connect it to a .4 second transition.



Start with your link CSS, including the hover state:

```
1 | a { color: #e83119; }  
2 | a:hover { color: #0a99ae; }
```

Now, bring in the CSS3 to set and define which property you're transitioning, duration of transition and how that transition will proceed over time. In this case we're setting the color property to fade over .4 seconds with an `ease-out` timing effect, where the pace of the transition starts off quickly and slows as time runs out. To learn more about timing, check out the [Surfin' Safari Blog post on CSS animations](#). I prefer `ease-`

out most of the time simply because it yields a more immediate transition, giving users a more immediate cue that something is changing.

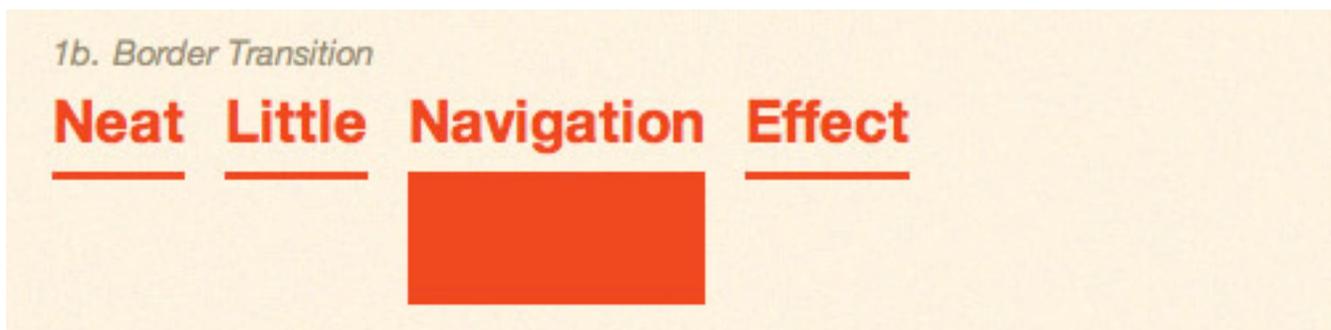
```
1 a {  
2 -webkit-transition-property: color;  
3 -webkit-transition-duration:.4s;  
4 -webkit-transition-timing:ease-out;  
5 }
```

You can also combine these into a single CSS property by declaring the property, duration, and timing function in that order:

```
1 a { -webkit-transition: color .4s ease-out; }
```

[View the live example here](#)

The final product should be a red text link that subtly transitions to blue when users hover with their mouse pointer. This basic transitioning technique can be connected to an infinite amount of properties. Next, let's let's create a menu bar hover effect where border-thickness is combined with a .3 second transition.



To start, we'll set a series of navigation links with a 3 pixel bottom border, and a 50 pixel border on hover:

```
1 border-nav a { border-bottom: 3px solid #e83119 }
```

```
2 | border-nav a:hover { border-bottom: 50px solid  
  | #e83119 }
```

To bring the transition into the mix, let's set a transition to gradually extend the border thickness over .3 seconds in a single line of CSS:

```
1 | border-nav a { -webkit-transition: border .3s ease-  
  | out; }
```

[View the live example here](#)

Examples

This is just one example of how to use these transitions to enhance links and navigation items. Here are a few other sites with similar creative techniques:

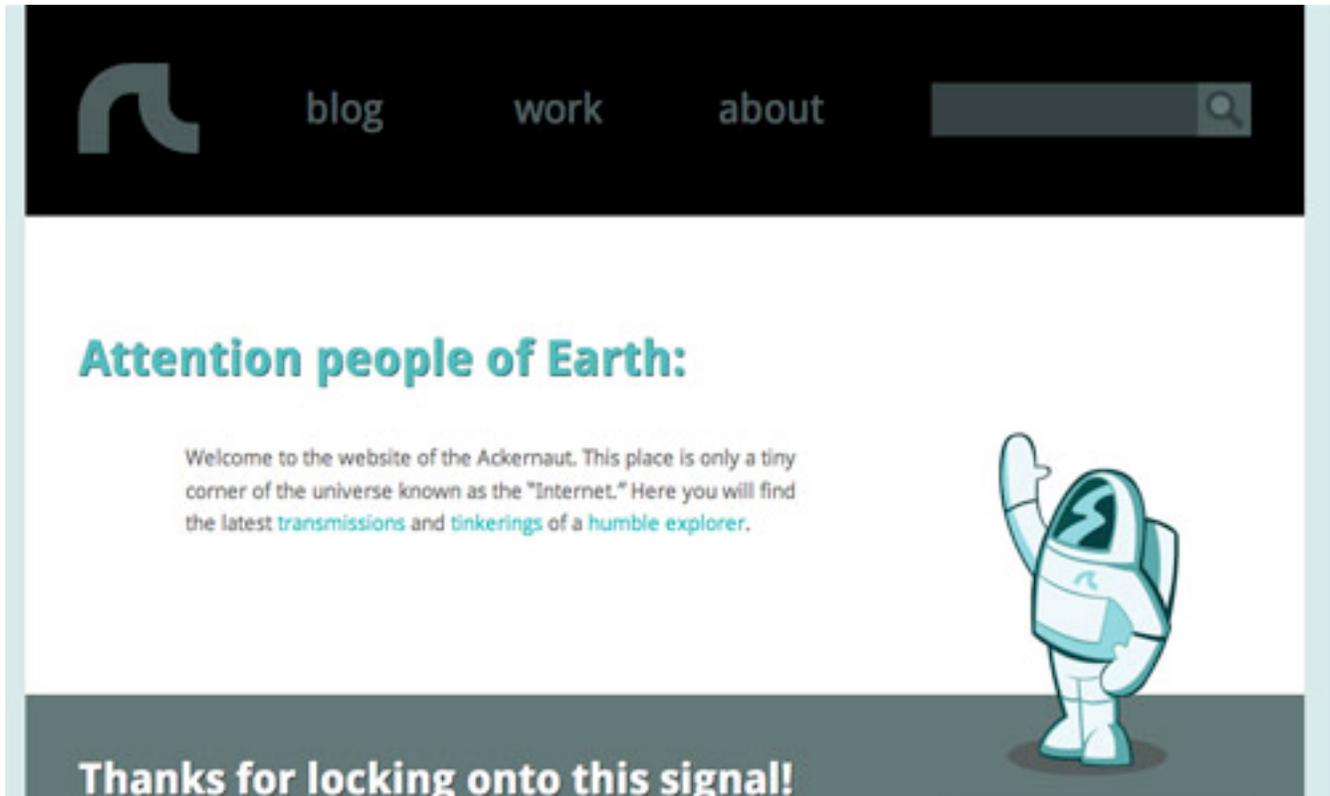
[Team Excellence](#)

The webkit transition on all navigation items, including the main navigation set at .2s provides a nice effect without making visitors wait too long for the hover state.



[Ackernaut](#)

Ackernaut has subtle transitions on all link hovers, and extends the property to fade the site header in/out.



[DesignSwap](#)

On DesignSwap, all text links have a .2 second transition on hover and the swapper profiles fade out to real details about the latest designs.



[Eric E. Anderson](#)

Eric E. Anderson has taken CSS3 implementation even further by implementing a transition on his main navigation for background color and color alongside `border-radius` and `box-shadow`.



Example #2: Background Clip

When connected to properties like `text-shadow` and `@font-face`, the `background-clip` property makes all things possible with type. To keep things simple, we'll start with taking a crosshatch image and masking it over some text. The code here is pretty simple. Start by wrapping some HTML in a div class called *bg-clip*:

```
1 <div class="bg-clip">
2 <h3>kablamo!</h3>
3 </div>
```

2a. `background-clip:text;`



Now to the CSS. First, set the image you will be masking the text with as the background-image. Then, set the `-webkit-text-fill-color` to transparent and define the `-webkit-background-clip` property for the text.

```
1 .bg-clip {  
2 background: url(../img/clipped_image.png) repeat;  
3 -webkit-background-clip: text;  
4 -webkit-text-fill-color: transparent;  
5 }
```

[View the live example here](#)

This opens the door for you to start adding texture or other graphic touches to your type without resorting to using actual image files. For even more CSS3 text experimentation, we can add the transform property to rotate the text (or any element for that matter) to any number of degrees. All it takes is a single line of CSS code:

```
1 -webkit-transform: rotate(-5deg);  
2 -moz-transform: rotate(-5deg);  
3 -o-transform: rotate (-5deg);
```

2b. Transform

The word "KABLAMO!" is rendered in a large, bold, red font with a fine grid or cross-hatch texture. The letters are slightly irregular and have a hand-drawn, stamped appearance. The background is a light beige or cream color.

Note: While `background-clip` isn't available in Firefox or Opera, the `transform` property is, so we'll set this for each browser.

[View the live example here](#)

Examples

This is a fairly simple implementation, but there are quite a few really interesting and innovative examples of this technique:

[Trent Walton](#)

An experiment of my own, combining `bg-clip` and `@font-face` to recreate a recent design.



[Neography](#)

An excellent example of what is possible when you throw `rotate`, `bg-clip` and `@font-face` properties together.



[Everyday Works](#)

One of the earliest innovative implementations of CSS text rotation I've seen.



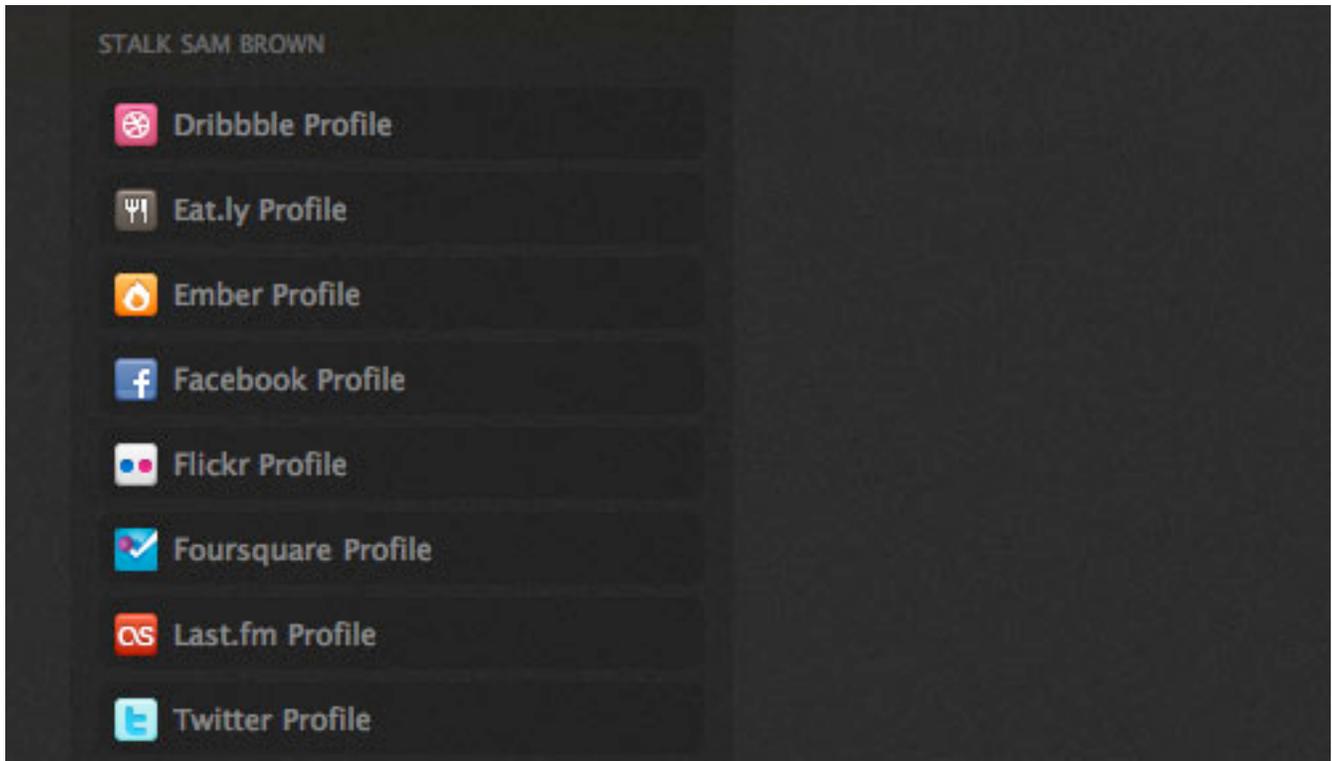
[Panic Blog](#)

The Panic blog randomly rotates divs / posts. Be sure to refresh to see subtle changes in the degree of rotation.



[Sam Brown](#)

Sam's got a really nice text-rotate hover effect on the "stalk" sidebar links.



Example #3: CSS Transforms, Box Shadow and RGBa

What used to take multiple `divs`, `pngs` and extra markup can now be accomplished with a few lines of CSS code. In this example we'll be combining the `transform` property from example 2 with `box-shadow` and `RGBa` color. To start things off, we'll create 4 image files, each showing a different version of the Smashing Magazine home page over time with a class for the shadow and a specific class to achieve a variety of rotations.



Here's the HTML:

```
1 <div class="boxes">
2 
3 
4 
5 
6 </div>
```

Let's set up the CSS for the RGBA Shadow:

```
1 | .shadowed {  
2 | border: 3px solid #fff;  
3 | -o-box-shadow: 0 3px 4px rgba(0,0,0,.5);  
4 | -moz-box-shadow: 0 3px 4px rgba(0,0,0,.5);  
5 | -webkit-box-shadow: 0 3px 4px rgba(0,0,0,.5);  
6 | box-shadow: 0 3px 4px rgba(0,0,0,.5);  
7 | }
```

Before moving forward, let's be sure we understand each property here. The `box-shadow` property works just like any drop shadow. The first two numbers define the shadow's offset for the X and Y coordinates. Here we've set the shadow to 0 for the X, and 3 for the Y. The final number is the shadow blur size, in this case it's 4px.

RGBA is defined in a similar manner. RGBA stands for red, green, blue, alpha. Here we've taken the RGB value for black as 0,0,0 and set it with a 50% alpha level at .5 in the CSS.

Now, let's finish off the effect by adding a little CSS Transform magic to rotate each screenshot:

```
1 | .smash1 { margin-bottom: -125px;  
2 | -o-transform: rotate(2.5deg);  
3 | -moz-transform: rotate(2.5deg);  
4 | -webkit-transform: rotate(2.5deg);  
5 | }
```

```
1 .smash2 {
2 -o-transform: rotate(-7deg);
3 -moz-transform: rotate(-7deg);
4 -webkit-transform: rotate(-7deg);
5 }
```

```
1 .smash3 {
2 -o-transform: rotate(2.5deg);
3 -moz-transform: rotate(2.5deg);
4 -webkit-transform: rotate(2.5deg);
5 }
```

```
1 .smash4 {
2 margin-top: -40px;
3 -o-transform: rotate(-2.5deg);
4 -moz-transform: rotate(-2.5deg);
5 -webkit-transform: rotate(-2.5deg);
6 }
```

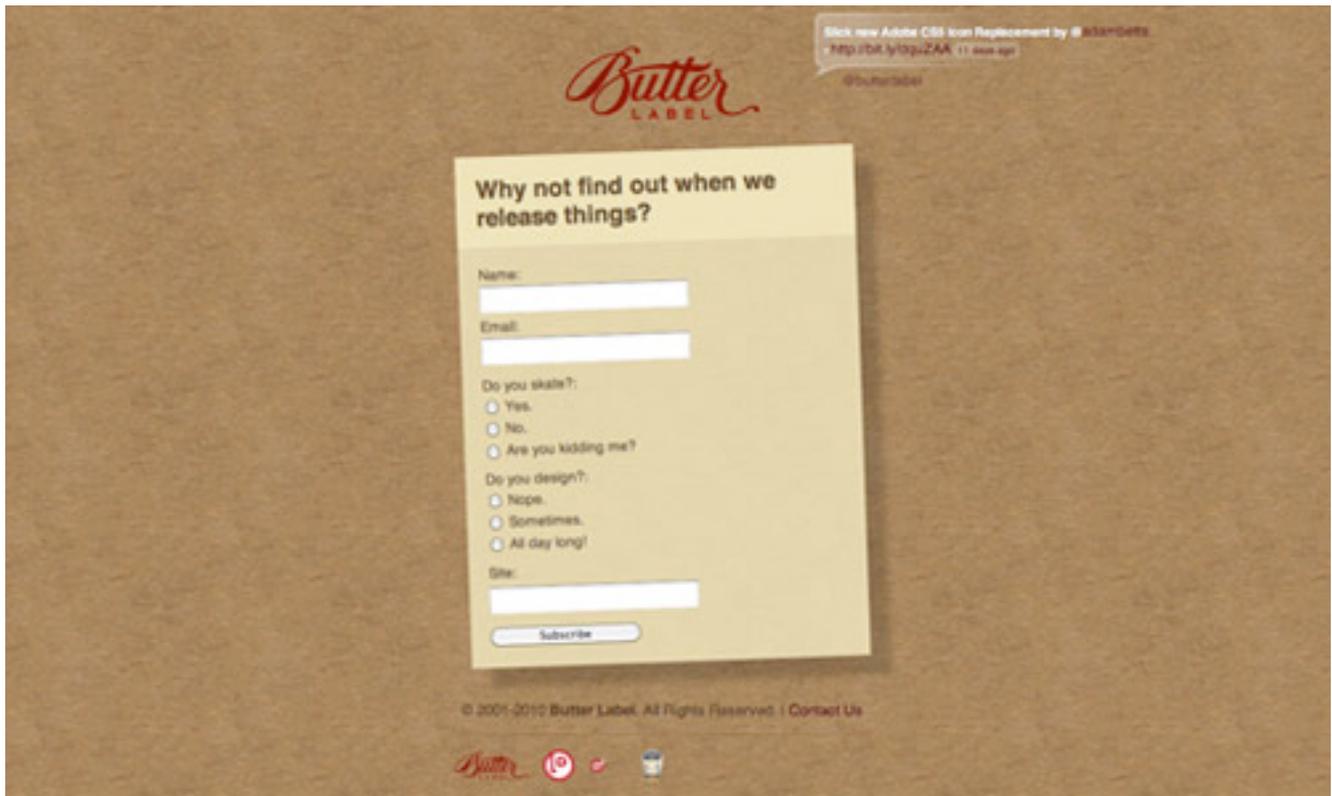
[View the live example here](#)

Examples

Here are a few additional sites with these properties implemented right now:

[Butter Label](#)

This site is jam packed with well-used CSS3. Notice the `transform` and `box-shadow` properties on the subscribe form.



[Hope 140](#)

Another site with plenty of CSS3 enhancements, Hope 140's End Malaria campaign site features a collage of photographs that all have the same shadow & transform properties outlined in our example.



[Simon Collison](#)

Simon Collison has implemented RGBa and box-shadow on each of the thumbnail links for his new website.



Example #4: CSS3 Animations

If you really want to push the envelope and truly experiment with the latest CSS3 properties, you've got to try creating a CSS3 keyframe animation. As a simple introduction, let's animate a circle .png image to track the outer edges of a rectangle. To begin, let's wrap circle.png in a div class:

```
1 <div class="circle_motion">
2 
3 </div>
```

4a. Keyframe Animation



The first step in the CSS will be to set the properties for `.circle_motion`, including giving it an animation name:

```
1 .circle_motion {  
2 -webkit-animation-name: track;  
3 -webkit-animation-duration: 8s;  
4 -webkit-animation-iteration-count: infinite;  
5 }
```

Now, all that remains is to declare properties for each percentage-based keyframe. To keep things simple here, I've just broken down the 8 second animation into 4 quarters:

```
1 @-webkit-keyframes track {  
2 0% {  
3 margin-top: 0px;  
4 }  
5 25% {  
6 margin-top: 150px;  
7 }  
8 50% {  
9 margin-top: 150px;  
10 margin-left: 300px;  
11 }
```

```
12 | 75% {  
13 | margin-top: 0px;  
14 | margin-left: 300px;  
15 | }  
16 | 100% {  
17 | margin-left: 0px;  
18 | }  
19 | }
```

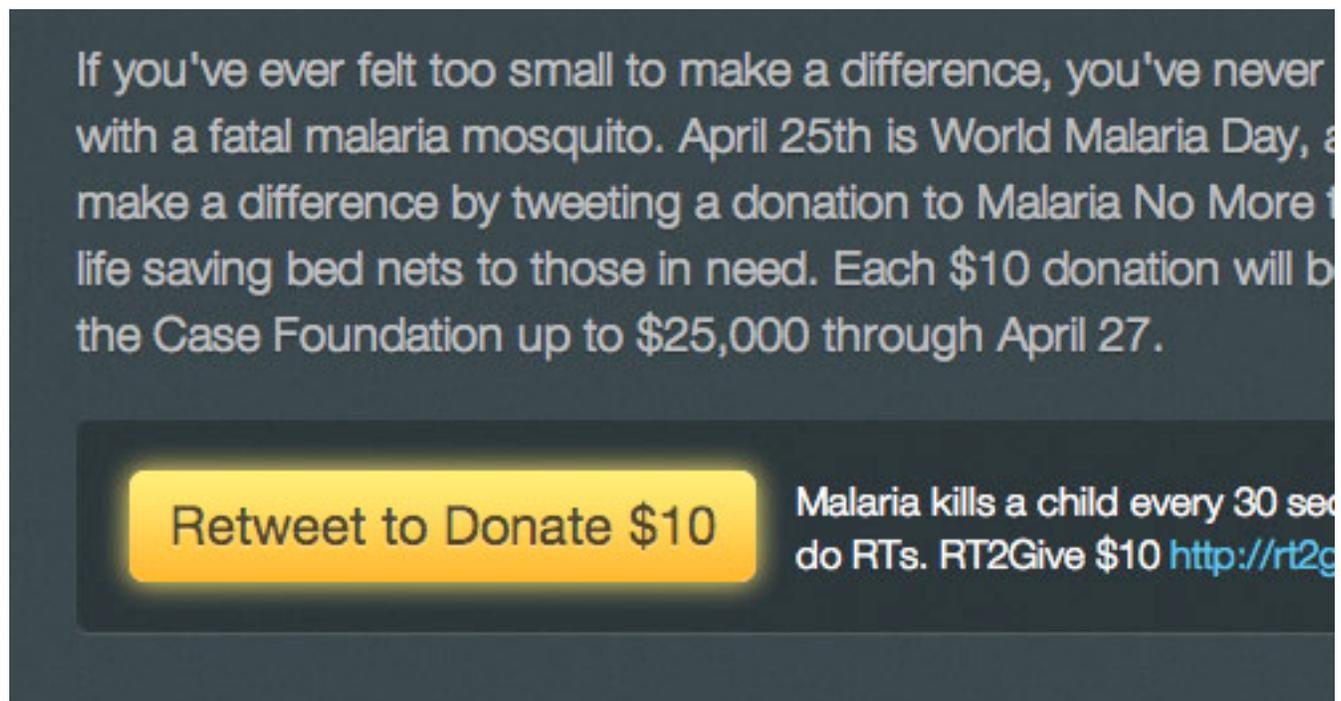
[View the live example here](#)

Examples

A few examples of CSS3 animations online now:

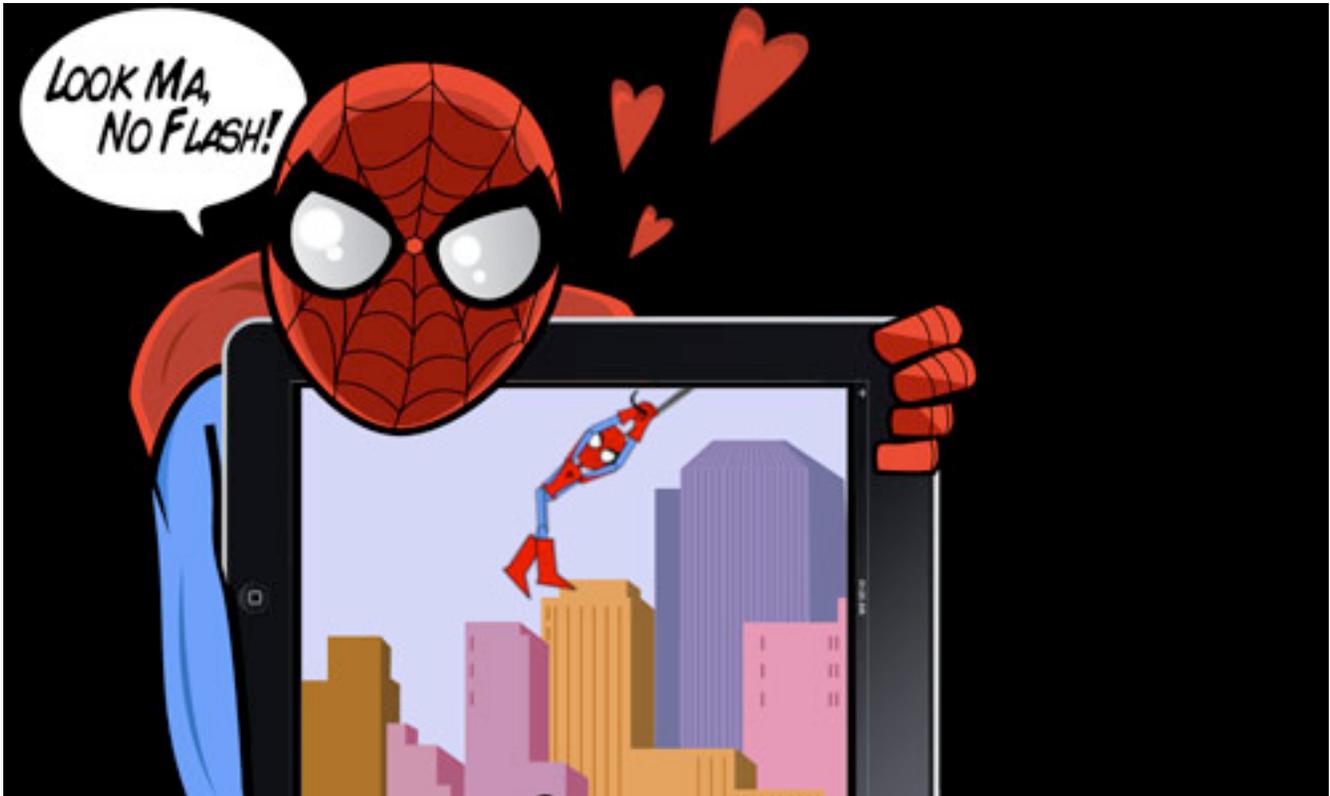
[Hope 140](#)

Hope 140 subtly animates their yellow “Retweet to Donate \$10” button’s box shadow.



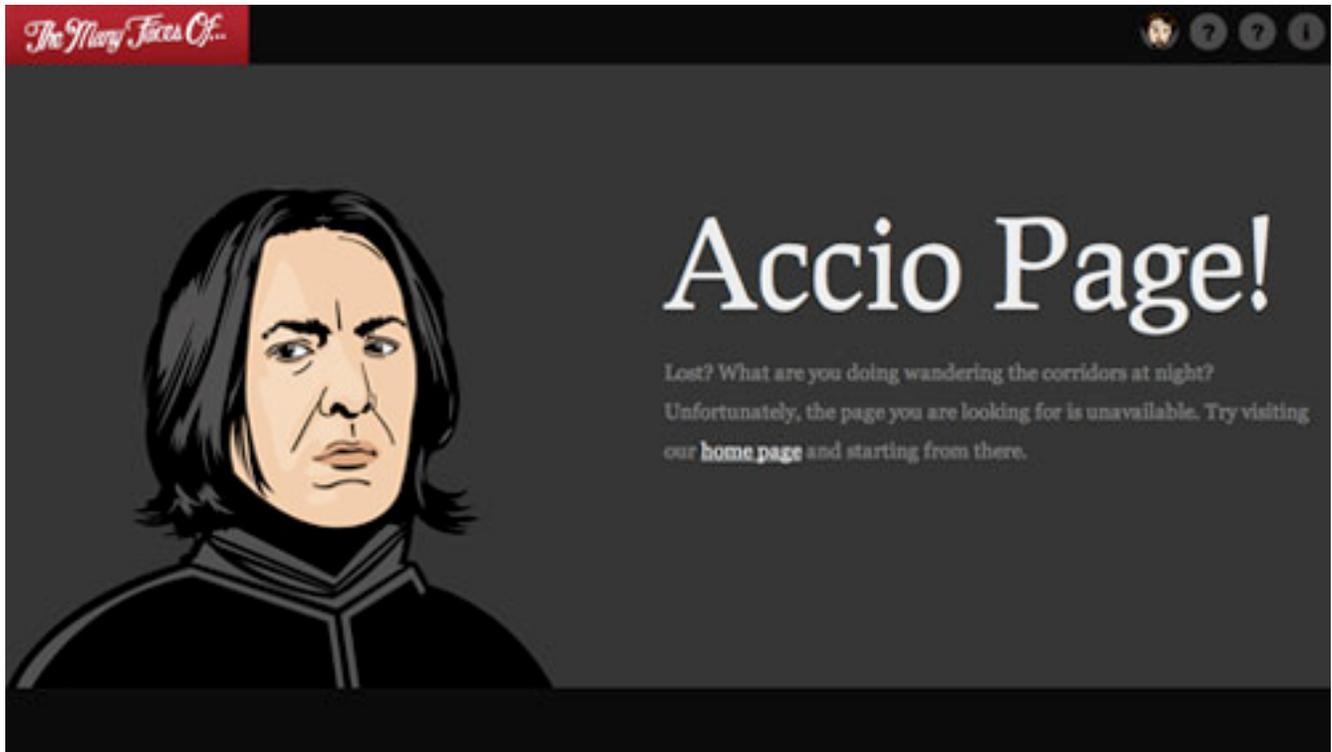
[Optimum7](#)

Anthony Calzadilla has recreated the Spider Man opening credits using CSS3 with JQuery and HTML5. You can also learn more about the process in his article [“Pure CSS3 Spiderman Cartoon w/ jQuery and HTML5 – Look Ma, No Flash!”](#).



[The Many Faces Of...](#)

The Many Faces Of... animates the background position of a `div` to create an effect where characters creep up from the bottom of the page.



OK, Dots connected! Now what?

Yes, all of this CSS3 stuff is insanely exciting. If you're like me, you'll want to start finding places to use it in the real world immediately. With each new experimental usage come even more concerns about implementation. Here are a few of my ever-evolving opinions about implementing these properties online for your consideration.

- CSS3 enhancements will never take the place of solid user-experience design.
- Motion and animation demands attention. Think about a friend waving to get your attention from across a crowded room or a flashing traffic light. Heavy-handed or even moderate uses of animation can significantly degrade user experience. If you are planning on implementing these techniques on a site with any sort of A to B conversion goals, be sure to consider the psychology of motion.
- Don't make people wait on animations. Especially when it comes to hover links, be sure there is an immediate state-change cue.
- Many of these effects can be used in a bonus or easter-egg type of application. Find places to go the extra mile.

Modern CSS Layouts: The Essential Characteristics

Zoe Mickley Gillenwater

Now is an exciting time to be creating CSS layouts. After years of what felt like the same old techniques for the same old browsers, we're finally seeing browsers implement CSS 3, HTML 5 and other technologies that give us cool new tools and tricks for our designs.

But all of this change can be stressful, too. How do you keep up with all of the new techniques and make sure your Web pages look great on the increasing number of browsers and devices out there? In part 1 of this article, you'll learn the five essential characteristics of successful modern CSS websites. In part 2 of this article, you'll learn about the techniques and tools that you need to achieve these characteristics.

We won't talk about design trends and styles that characterize modern CSS-based layouts. These styles are always changing. Instead, we'll focus on the broad underlying concepts that you need to know to create the most successful CSS layouts using the latest techniques. For instance, separating content and presentation is still a fundamental concept of CSS Web pages. But other characteristics of modern CSS Web pages are new or more important than ever. A modern CSS-based website is: progressively enhanced, adaptive to diverse users, modular, efficient and typographically rich.

Progressive Enhancement

Progressive enhancement means creating a solid page with appropriate markup for content and adding advanced styling (and perhaps scripting) to the page for browsers that can handle it. It results in web pages that are usable by all browsers but that do not look identical in all browsers. Users of newer, more advanced browsers get to see more cool visual effects and nice usability enhancements.

The idea of allowing a design to look different in different browsers is not new. CSS gurus have been preaching this for years because font availability and rendering, color tone, pixel calculations and other technical factors have always varied between browsers and platforms. Most Web designers avoid “pixel perfection” and have accepted the idea of their designs looking slightly different in different browsers. But progressive enhancement, which has grown in popularity over the past few years, takes it a step further. Designs that are progressively enhanced may look more than slightly different in different browsers; they might look *very* different.

For example, the [tweetCC website](#) has a number of CSS 3 properties that add attractive visual touches, like drop-shadows behind text, multiple columns of text and different-colored background “images” (without there having to be actually different images). These effects are seen to various extents in different browsers, with old browsers like IE 6 looking the “plainest.” However, even in IE 6, the text is perfectly readable, and the design is perfectly usable.

Search for a Twitter user or browse the 3,098 Twitter users who license tweets.

Does
license their tweets?

Find out

licensing your tweets matters

people don't have to ask.

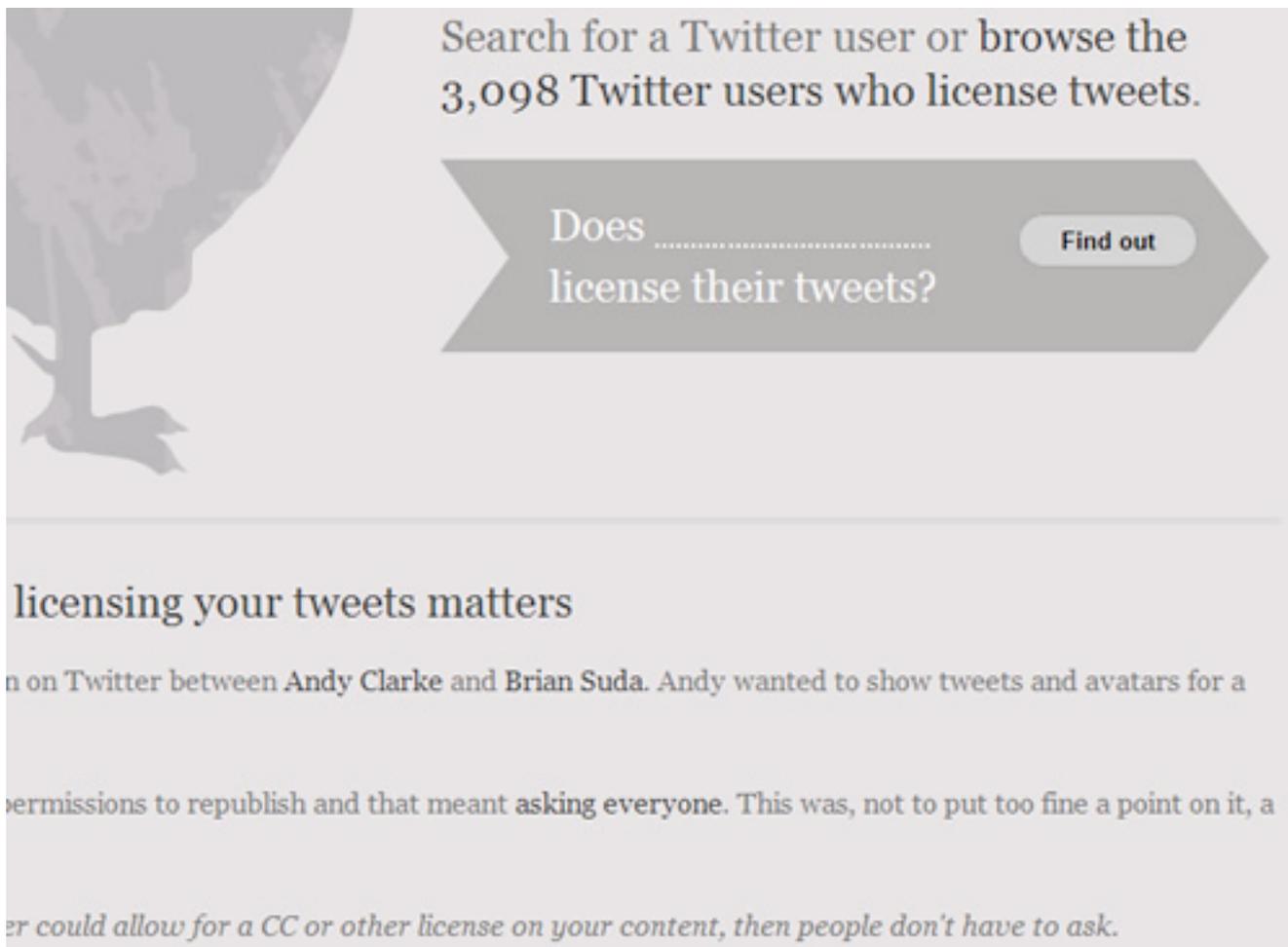
Twitter is clear that they make no intellectual claims over your tweets.

We claim no intellectual

But what about the people who want to reproduce your tweets? With no tweet license policy, republishing them without asking is a bit of a grey area.

login or password as this password anti-pattern practice teaches people how to be phished. Don't scatter your passwords around like chicken feed.

[tweetCC](#) in Safari.



[tweetCC](#) in IE 6.

In CSS 3-capable browsers like Safari (top), the [tweetCC](#) website shows a number of visual effects that you can't see in IE 6 (bottom).

These significant differences between browsers are perfectly okay, not only because that is the built-in nature of the Web, but because progressive enhancement brings the following benefits:

- **More robust pages**

Rather than use the graceful degradation method to create a fully functional page and then work backwards to make it function in less-

capable browsers, you focus first on creating a solid “base” page that works everywhere.

- **Happier users**

You start building the page making sure the basic functionality and styling is the same for everyone. People with old browsers, mobile devices and assistive technology are happy because the pages are clean and reliable and work well. People with the latest and greatest browsers are happy because they get a rich, polished experience.

- **Reduced development time**

You don't have to spend hours trying to get everything to look perfect and identical in all browsers. Nor do you have to spend much time reverse-engineering your pages to work in older browsers after you have completed the fully functional and styled versions (as is the case with the graceful degradation method).

- **Reduced maintenance time**

If a new browser or new technology comes out, you can add new features to what you already have, without altering and possibly breaking your existing features. You have only one base version of the page or code to update, rather than multiple versions (which is the case with graceful degradation).

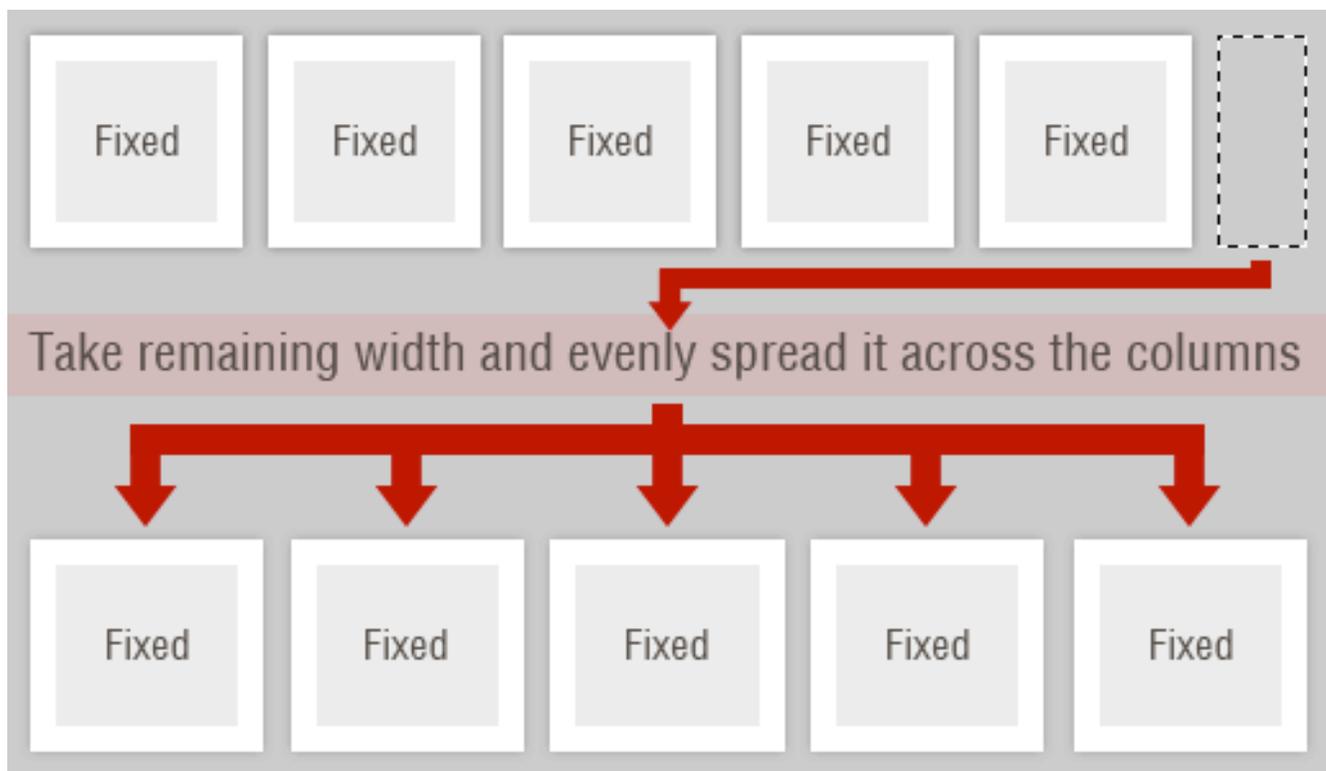
- **More fun**

It's just plain fun to be able to use cool and creative new techniques on your Web pages, and not have to wait years for old browsers to die off.

Adaptive to Diverse Users

Modern CSS-based Web pages have to accommodate the diverse range of browsers, devices, screen resolutions, font sizes, assistive technologies and

other factors that users bring to the table. This concept is also not new but is growing in importance as Web users become increasingly diverse. For instance, a few years ago, you could count on almost all of your users having [one of three screen resolutions](#). Now, users could be viewing your pages on 10-inch netbooks, 30-inch widescreen monitors or anything in between, not to mention tiny mobile devices.



In his article ["Smart columns with CSS and jQuery"](#) Soh Tanaka describes his techniques that adapts the layout depending on the current browser window size.

Creating Web designs that work for all users in all scenarios will never be possible. But the more users you can please, the better: for them, for your clients and for you. Successful CSS layouts now have to be more flexible and adaptable than ever before to the increasing variety of ways in which users browse the Web.

Consider factors such as these when creating CSS layouts:

- **Browser**

Is the design attractive and usable with the most current and popular browsers? Is it at least usable with old browsers?

- **Platform**

Does the design work on PC, Mac and Linux machines?

- **Device**

Does the design adapt to low-resolution mobile devices? How does it look on mobile devices that have full resolution (e.g. iPhones)?

- **Screen resolution**

Does the design stay together at multiple viewport (i.e. window) widths? Is it attractive and easy to read at different widths? If the design does adapt to different viewport widths, does it correct for extremely narrow or wide viewports (e.g. by using the `min-width` and `max-width` properties)?

- **Font sizes**

Does the design accommodate different default font sizes? Does the design hold together when the font size is changed on the fly? Is it attractive and easy to read at different font sizes?

- **Color**

Does the design make sense and is the content readable in black and white? Would it work if you are color blind or have poor vision or cannot detect color contrast?

- **JavaScript presence**

Does the page work without JavaScript?

- **Image presence**

Does the content make sense and is it readable without images (either background or foreground)?

- **Assistive technology/disability**

Does the page work well in screen readers? Does the page work well without a mouse?

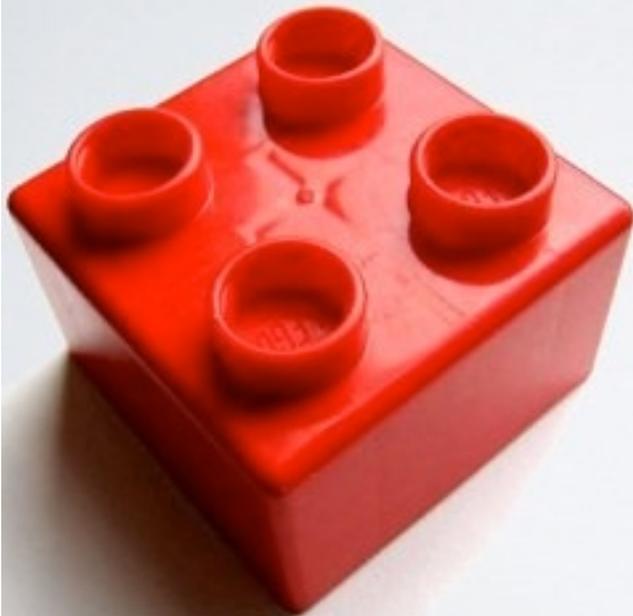
This is not a comprehensive list; and even so, you would not be able to accommodate every one of these variations in your design. But the more you can account for, the more user-friendly, robust and successful your website will be.

Modular

Modern websites are no longer collections of static pages. Pieces of content and design components are reused throughout a website and even shared between websites, as content management systems (CMS), RSS aggregation and user-generated content increase in popularity. Modern design components have to be able to adapt to all of the different places they will be used and the different types and amount of content they will contain.

OBJECT ORIENTED CSS

for high performance web applications and sites.



Nicole Sullivan

[Object Oriented CSS](#) is Nicole Sullivan's attempt to create a framework that would allow developers to write fast, maintainable, standards-based, modular front end code.

Modular CSS, in a broad sense, is CSS that can be broken down into chunks that work independently to create design components that can themselves be reused independently. This might mean separating your style into multiple sheets, such as *layout.css*, *type.css*, and *color.css*. Or it might mean creating a collection of universal CSS classes for form layout that you can apply to any form on your website, rather than have to style each form individually. CMS', frameworks, layout grids and other tools all help you create more modular Web pages.

Modular CSS offers these benefits (depending on which techniques and tools you use):

- **Smaller file sizes**

When all of the content across your website is styled with only a handful of CSS classes, rather than an array of CSS IDs that only work on particular pieces of content on particular pages, your style sheets will have many fewer redundant lines of code.

- **Reduced development time**

Using frameworks, standard classes and other modular CSS tools keeps you from having to re-invent the wheel every time you start a new website. By using your own or other developers' tried and true CSS classes, you spend less time testing and tweaking in different browsers.

- **Reduced maintenance time**

When your style sheets include broad, reusable classes that work anywhere on your website, you don't have to come up with new styles when you add new content. Also, when your CSS is lean and well organized, you spend less time tracking down problems in your style sheets when browser bugs pop up.

- **Easier maintenance for others**

In addition to making maintenance less time-consuming for you, well-organized CSS and smartly named classes also make maintenance easier for developers who weren't involved in the initial development of the style sheets. They'll be able to find what they need and use it more easily. CMS' and frameworks also allow people who are not as familiar with your website to update it easily, without screwing anything up.

- **More design flexibility**

Frameworks and layout grids make it easy, for instance, to switch between different types of layout on different pages or to [plug in different types of content](#) on a single page.

- **More consistent design**

By reusing the same classes and avoiding location-specific styling, you ensure that all elements of the same type look the same throughout the website. CMS' and frameworks provide even more insurance against design inconsistency.

Efficient

Modern CSS-based websites should be efficient in two ways:

- Efficient for you to develop
- Efficient for the server and browser to display to users

As Web developers, we can all agree that efficiency on the development side is a good thing. If you can save time while still producing high-quality work, then why wouldn't you adopt more efficient CSS development practices? But creating pages that perform efficiently for users is sometimes not given enough attention. Even though connection speeds are getting faster and faster, page load times are still very important to users. In fact, as connection speeds increase, users might expect all pages to load very quickly, so making sure your website can keep up is important. Shaving just a couple of seconds off the loading time can make a big difference.

We've already discussed how modular CSS reduces development and maintenance time and makes your workflow a lot faster and more efficient. A myriad of tools are out there to help you write CSS quickly, which we'll

cover in part 2 of this article. You can also streamline your CSS development process by using many of the new effects offered by CSS 3, which cut down on your time spent creating graphics and coding usability enhancements.

Some CSS 3 techniques also improve performance and speed. For instance, traditional rounded-corner techniques require multiple images and DIVs for just one box. Using CSS 3 to create rounded corners requires no images, thus reducing the number of HTTP calls to the server and making the page load faster. No images also reduces the number of bytes the user has to download and speeds up page loading. CSS 3 rounded-corners also do not require multiple nested DIVs, which reduces page file size and speeds up page loading again. Simply switching to CSS 3 for rounded corners can give your website a tremendous performance boost, especially if you have many boxes with rounded corners on each page.

Writing clean CSS that takes advantage of shorthand properties, grouped selectors and other efficient syntax is of course just as important as ever for improving performance. Many of the more advanced tricks for making CSS-based pages load faster are also not new but are increasing in usage and importance. For instance, the [CSS Sprites](#) technique, whereby a single file holds many small images that are each revealed using the CSS `background-position` property, was first described by [Dave Shea in 2004](#) but has been improved and added to a great deal since then. Many large enterprise websites now rely heavily on the technique to minimize HTTP requests. And it can improve efficiency for those of us working on smaller websites, too. CSS compression techniques are also increasingly common, and many automated tools make compressing and optimizing your CSS a breeze, as you'll also learn in part 2 of this article.

Rich Typography

Rich typography may seem out of place with the four concepts we have just covered. But we're not talking about any particular style of typography or fonts, but rather the broader concept of creating readable yet unique-looking text by applying tried and true typographic principles using the newest technologies. Typography is one of the most rapidly evolving areas of Web design right now. And boy, does it need to evolve! While Web designers have had few limits on what they could do graphically with their designs, their limits with typography have been glaring and frustrating.

Until recently, Web designers were limited to working with the fonts on their end users' machines. Image replacement tricks and clever technologies such as [sIFR](#) have opened new possibilities in the past few years, but none of these is terribly easy to work with. In the past year, we've finally made great strides in what is possible for type on the Web because of the growing support for CSS 3's `@font-face` property, as well as new easy-to-use technologies and services like [Cufón](#) and [Typekit](#).

The `@font-face` rule allows you to link to a font on your server, called a "Web font," just as you link to images. So you are no longer limited to working with the fonts that most people have installed on their machines. You can now take advantage of the beautiful, unique fonts that you have been dying to use.

How Are We 1

In 2002 we set out to build a company that focused on delivering great user experiences in the digital age. We couldn't rely on the legacy of past employers as a basis for our company. Instead, we challenged the conventional formula and created a new approach and process.

A few reasons we think clients and employees love

PARTNERS ON EVERY PROJECT

1 Our clients work with the people who pitch the business. This means we only

SMALL, AGILE, CREATIVE TEAMS

2 We were built by creative people for creative people. That's why 95% of our

DIRECT A

3 N
te
manag

Craigmod

The Potential of Web Typography:

@FONT-FACE AND FIREFOX 3.5

by [Ian Lynam](#) & [Craig Mod](#)

FIREFOX 3.5 IS OUT. AND THE MORE USERS DOWNLOAD more designers will be able to take advantage of the @font-face. How can @font-face be used with currently implemented to create engaging, nuanced and more mature typography? Let's

@font-face — what it is exactly?



@FONT-FACE IS A CSS RULE IMPLEMENTED IN Firefox's latest 3.5 browser release. It allows web designers to reference fonts not installed on end user

What we'd like



HERE ARE we'd love specifications:

Nicewebsite

ALL AB

THE EXLJBRIS EXPRESS

FREIGHTAGE

Museo and Museo Sans are available in several freights. Er, weights. Use these to your advantage by setting display text in light weights

ROLLING STOCK

Web layouts, like railroads, must oblige a hodgepodge of constituent aesthetics. Our job is crud mitigation. Helvetica can understudy

L
b
d
t
t/

The three [screenshots](#) above are all examples of what `@font-face` can do.

The main problem with `@font-face`, aside from the ever-present issue of [browser compatibility](#), is that most font licenses—even those of free fonts—do not allow you to serve the fonts over the Web. That’s where `@font-face` services such as [Typekit](#), [Fontdeck](#) and Kernest are stepping in. They work with type foundries to license select fonts for Web design on a “rental” basis. These subscription-based services let you rent fonts for your website, giving you a much wider range of fonts to work with, while avoiding licensing issues.



[For A Beautiful Web](#) uses the [Typekit font embedding service](#) for the website name, introductory text and headings.



[Ruler of the Interwebs](#) uses the [Kernest font embedding service](#) for the website name and headings.

We still have a long way to go, but the new possibilities make typography more important to Web design than ever before. To make your design truly stand out, use these modern typographic techniques, which we'll cover in even greater detail in Part 2.

Summary

We've looked at five characteristics of modern CSS websites:

- Progressively enhanced
- Adaptive to diverse users
- Modular
- Efficient
- Typographically rich

In part 2 we'll go over the techniques and tools that will help you implement these important characteristics on your CSS-based Web pages.

Modern CSS Layouts, Part 2: The Essential Techniques

Zoe Mickley Gillenwater

In part 1, you learned that modern, CSS-based Web sites should be progressively enhanced, adaptive to diverse users, modular, efficient and typographically rich. Now that you know *what* characterizes a modern CSS Web site, *how* do you build one? Here are dozens of essential techniques and tools to learn and use to achieve the characteristics of today's most successful CSS-based Web pages.

Just as in the previous article, we're not going to be talking about design trends and styles; these styles are always changing. Instead, we're focusing on the specific techniques that you need to know to create modern CSS-based Web pages of any style. For each technique or tool, we'll indicate which of the five characteristics it helps meet. To keep this shorter than an encyclopedia, we'll also just cover the basics of each technique, then point you to some useful, hand-picked resources to learn the full details.

CSS3

CSS3, the newest version of CSS that is now being partially supported by most browsers, is the primary thing you need to know in order to create modern CSS Web sites, of course. CSS is a styling language, so it's no surprise that most of what's new in CSS3 is all about visual effects. But CSS3 is about more than progressive enhancement and pretty typography. It can also aid usability by making content easier to read, as well as improve efficiency in development and page performance.

There are too many CSS3 techniques to cover in a single article, let alone an article that isn't just about CSS3! So, we'll go through the basics of the most important or supported CSS3 techniques and point you to some great resources to learn more in-depth.

CSS3 Visual Effects

Semi-transparent Color

Aids in: progressive enhancement, efficiency

[RGBA](#) allows you to specify a color by not only setting the values of red, green, and blue that it's comprised of, but also the level of opacity it should have. An alternative to RGBA is [HSLA](#), which works the same way, but allows you to set values of hue, saturation, and lightness, instead of values of red, green, and blue. The article [Color in Opera 10 — HSL, RGB and Alpha Transparency](#) explains how HSLA can be more intuitive to use than RGBA.



The [24 Ways Web site](#) makes extensive use of RGBA to layer semi-transparent boxes and text over each other.

RGBA or HSLA isn't just about making things look cool; it can also improve your Web site's efficiency. You don't have to take time to make alpha-transparent PNGs to use as backgrounds, since you can just use a color in the CSS, and the user agent doesn't have to download those images when loading the site.

Styling Backgrounds and Borders

Aids in: progressive enhancement, efficiency

CSS3 offers a whole host of new ways to style backgrounds and borders, often without having to use images or add extra `div`s. Most of these new techniques already have good browser support, and since they're mainly

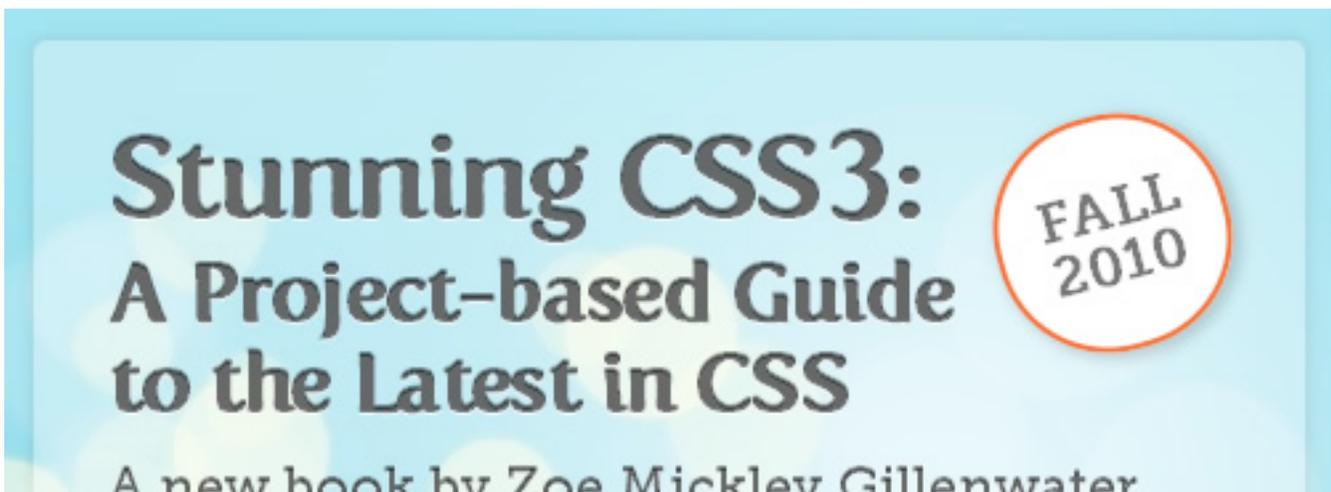
used for purely cosmetic changes, they're a good way to get some progressive enhancement goodness going in your sites right away.

Here are some of the new things CSS3 lets you do with backgrounds:

- **Multiple backgrounds on a single element:** You can now add [more than one background image](#) to an element by listing each image, separated by commas, in the `background-image` property. No more nesting extra `div`s just to have more elements to attach background images onto!
- **More control over where backgrounds are placed:** The new [background-clip](#) and [background-origin](#) properties let you control if backgrounds are displayed under borders, padding, or just content, as well as where the origin point for `background-position` should be.
- **Background sizing:** You can scale background images using the new [background-size property](#). While scaling won't look good on many background images, it could be really handy on abstract, grunge-type backgrounds, where tiling can be difficult and where some image distortion would be unnoticeable.
- **Gradients:** While just part of a CSS3 [draft spec](#), Safari, Chrome and Firefox support declaring multiple color and placement values in the `background-image` property to create gradients without images. This allows the gradients to scale with their container — unlike image gradients — and eliminates the need for page users to download yet another image while viewing your site.

CSS3 lets you do the following with borders:

- **Rounded corners:** Use the [border-radius-property](#) to get rounded corners on `divs`, buttons, and whatever else your heart desires — all without using images or JavaScript.
- **Images for borders:** With CSS 2.1, the only way to create a graphic border was to fake it with background images, often multiple ones pieced together on multiple `divs`. You can now add unique borders without having to use background images by adding the images to the borders directly, using the new [border-image property](#), which also allows you to control how the images scale and tile.



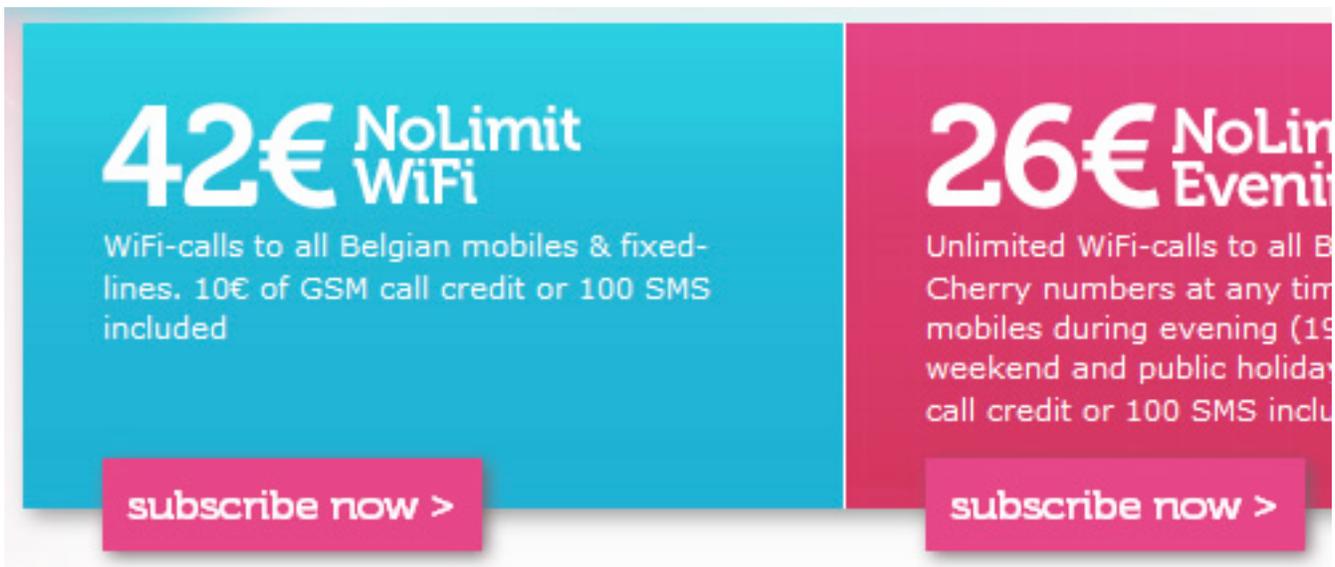
The `border-radius` property can be used to round corners and even create circles out of pure CSS, with no images needed. ([Stunning CSS3 Web site](#))

Drop Shadows

Aids in: progressive enhancement, adaptability, efficiency

Drop shadows can provide some visual polish to your design, and now they're possible to achieve without images, both on boxes and on text.

The [box-shadow property](#) has been temporarily removed from the CSS3 spec, but is supposed to be making its re-appearance soon. In the meantime, it's still possible to get image-free drop shadows on boxes in Firefox and Safari/Chrome using the `-moz-box-shadow` and `-webkit-box-shadow` properties, respectively, and in Opera 10.5 using the regular `box-shadow` property with no prefix. In the property, you set the shadow's horizontal and vertical offsets from the box, color, and can optionally set blur radius and/or spread radius.



The [Cherry Web site](#) uses drop shadows created with `box-shadow` on many boxes and buttons.

The [text-shadow property](#) adds drop shadows on — you guessed it — text. It's supported by all the major browsers except — you guessed it — Internet Explorer. This makes it the perfect progressive enhancement candidate — it's simply a visual effect, with no harm done if some users don't see it. Similarly to `box-shadow`, it takes a horizontal offset, vertical offset, blur radius and color.

Using `text-shadow` keeps you from resorting to Flash or images for your text. This can speed up the time it takes you to develop the site, as well as speed up your pages. Avoiding Flash and image text can also aid accessibility and usability; just make sure your text is still legible with the drop shadow behind it, so you don't inadvertently *hurt* usability instead!

Transforms

Aids in: progressive enhancement, adaptability, efficiency

CSS3 makes it possible to do things like rotate, scale, and skew the objects in your pages without resorting to images, Flash, or JavaScript. All of these effects are called "[transforms](#)." They're supported in Firefox, Safari, Chrome, and Opera 10.5.

You apply a transform using the `transform` property, naturally (though for now you'll need to use the browser-specific equivalents: `-moz-transform`, `-webkit-transform`, and `-o-transform`). You can also use the `transform-origin` property to specify the point of origin from which the transform takes place, such as the center or top right corner of the object.

In the `transform` property, you specify the type of transform (called "transform functions"), and then in parentheses write the measurements needed for that particular transform. For instance, a value of

`translate(10px, 20px)` would move the element 10 pixels to the right and 20 pixels down from its original location in the flow. Other supported transform functions are `scale`, `rotate`, and `skew`.



The [BeerCamp SXSW 2010 site](#) scales and rotates the sponsor logos on hover.

Animation and Transitions

Aids in: progressive enhancement, efficiency

Animation is now no longer the solely the domain of Flash or JavaScript — you can now create animation in pure HTML and CSS. Unfortunately, CSS3 animation and transitions do not have very good browser support, but as with most of the effects we've talked about so far, they're great for adding a little non-essential flair.

[CSS3 transitions](#) are essentially the simplest type of animation. They smoothly ease the change between one CSS value to another over a

specified duration of time. They're triggered by changing element states, such as hovering. They're supported by Safari, Chrome, and Opera 10.5.

To create a transition, all you have to do is specify which elements you want to apply the transition to and which CSS properties will transition, using the `transition-property` property. You'll also need to add a `transition-duration` value in seconds ("s" is the unit), since the default time a transition takes is 0 seconds. You can add them both in the `transition` shorthand property. You can also specify a delay or a timing function to more finely tune how the two values switch.

Beyond transitions, full-fledged [animations](#) with multiple keyframes are also possible with CSS3 (but currently only supported in Safari/Chrome). First, you give the animation a name and define what the animation will do at different points (keyframes, indicated with percentages) through its duration. Next, you apply this animation to an element using the `animation-name`, `animation-duration`, and `animation-iteration-count` properties. You could also set a delay and timing function, just like with transitions.

CSS3 Usability / Readability Enhancements

Most the CSS3 techniques we've gone over so far have been purely cosmetic effects that aid progressive enhancement. But CSS3 can also be used to improve the usability of your pages.

Creating Multiple Columns of Text

Aids in: progressive enhancement, adaptability

Some pieces of text are more readable in narrow, side-by-side columns, similar to traditional newspaper layout. You can tell the browser to arrange your text into [columns](#) by either defining a width for each column (the `column-width` property) or by defining a number of columns (the `column-count` property). Other new properties let you control gutters/gaps, rule lines, breaking between columns and spanning across columns. (For now, you need to use the browser-specific prefixes of `-moz` and `-webkit`.) This is another one of those techniques that can harm instead of aid usability if used improperly, as explained in "[CSS3 Multi-column layout considered harmful](#)," so use it judiciously.

Controlling Text Wrapping and Breaking

Aids in: adaptability

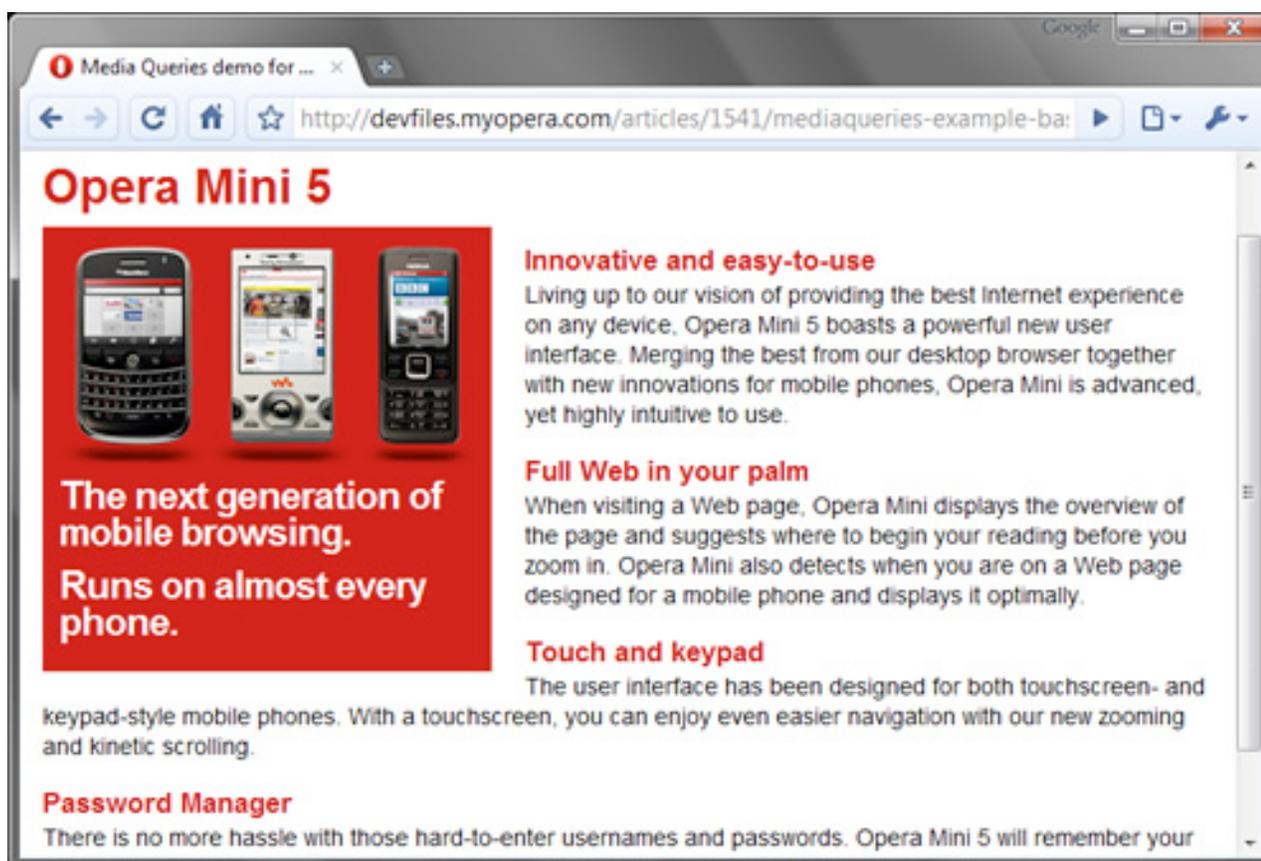
CSS3 gives you more control over how blocks of text and individual words break and wrap if they're too long to fit in their containers. Setting [word-wrap](#) to `break-word` will break a long word and wrap it onto a new line (particularly handy for long URLs in your text). The [text-wrap](#) property gives you a number of options for where breaks may and may not occur between words in your text. The CSS2 [white-space](#) property has now in CSS3 become a shorthand property for the new [white-space-collapse](#) and `text-wrap` properties, giving you more control over what spaces and line breaks are preserved from your markup to the rendered page. Another property worth mentioning, even though it's not currently in the CSS3 specification, is `text-overflow`, which allows the browser to add an

ellipsis character (...) to the end of a long string of text instead of letting it overflow.

Media Queries

Aids in: adaptability, efficiency

CSS2 let you apply different styles to different media types — screen, print, and so on. CSS3's [media queries](#) take this a step further by letting you customize styles based on the user's viewport width, display aspect ratio, whether or not his display shows color, and more. For instance, you could detect the user's viewport width and change a horizontal nav bar into a vertical menu on wide viewports, where there is room for an extra column. Or you could change the colors of your text and backgrounds on non-color displays.



Media Queries demo for ... x

http://devfiles.myopera.com/articles/1541/mediaqueries-example-ba:

Opera Mini 5

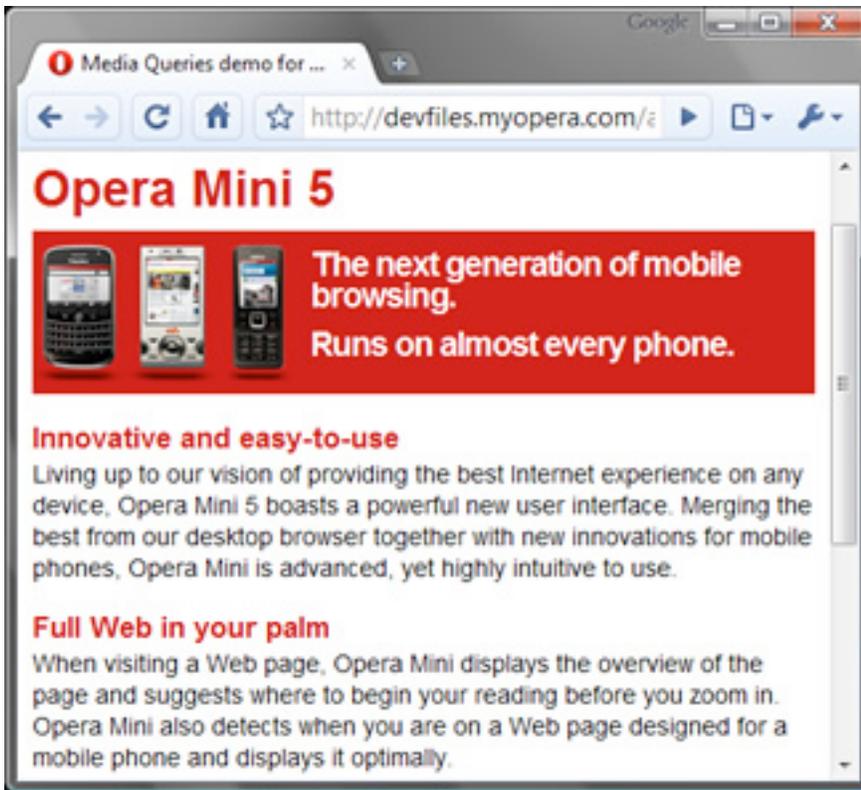
The next generation of mobile browsing.
Runs on almost every phone.

Innovative and easy-to-use
Living up to our vision of providing the best Internet experience on any device, Opera Mini 5 boasts a powerful new user interface. Merging the best from our desktop browser together with new innovations for mobile phones, Opera Mini is advanced, yet highly intuitive to use.

Full Web in your palm
When visiting a Web page, Opera Mini displays the overview of the page and suggests where to begin your reading before you zoom in. Opera Mini also detects when you are on a Web page designed for a mobile phone and displays it optimally.

Touch and keypad
The user interface has been designed for both touchscreen- and keypad-style mobile phones. With a touchscreen, you can enjoy even easier navigation with our new zooming and kinetic scrolling.

Password Manager
There is no more hassle with those hard-to-enter usernames and passwords. Opera Mini 5 will remember your



This [demo file from Opera](#) uses media queries to rearrange elements and resize text and images based on viewport size.

Media queries couldn't come at a better time — there is more variety in the devices and settings people use to browse the Web than ever before. You can now optimize your designs more precisely for these variations to provide a more usable and attractive design, but without having to write completely separate style sheets, use JavaScript redirects, and other less efficient development practices.

Improving Efficiency Through CSS3

Many of the visual effect properties of CSS3 that we've gone over have a great bonus in addition to making your design look great: they can improve efficiency, both in your development process and in the performance of the pages themselves.

Any CSS3 property that keeps you from having to create and add extra images is going to reduce the time it takes you to create new pages as well as re-skin existing ones. Less images also mean less stuff for the server to have to send out and less stuff for the users to download, both of which increase page loading speed.

CSS3 properties that keep you from having to add extra `div`s or extra classes can also reduce your development time as well as file size. We've already gone over some great techniques that help with this, but there are a few more worth mentioning.

The `box-sizing` Property

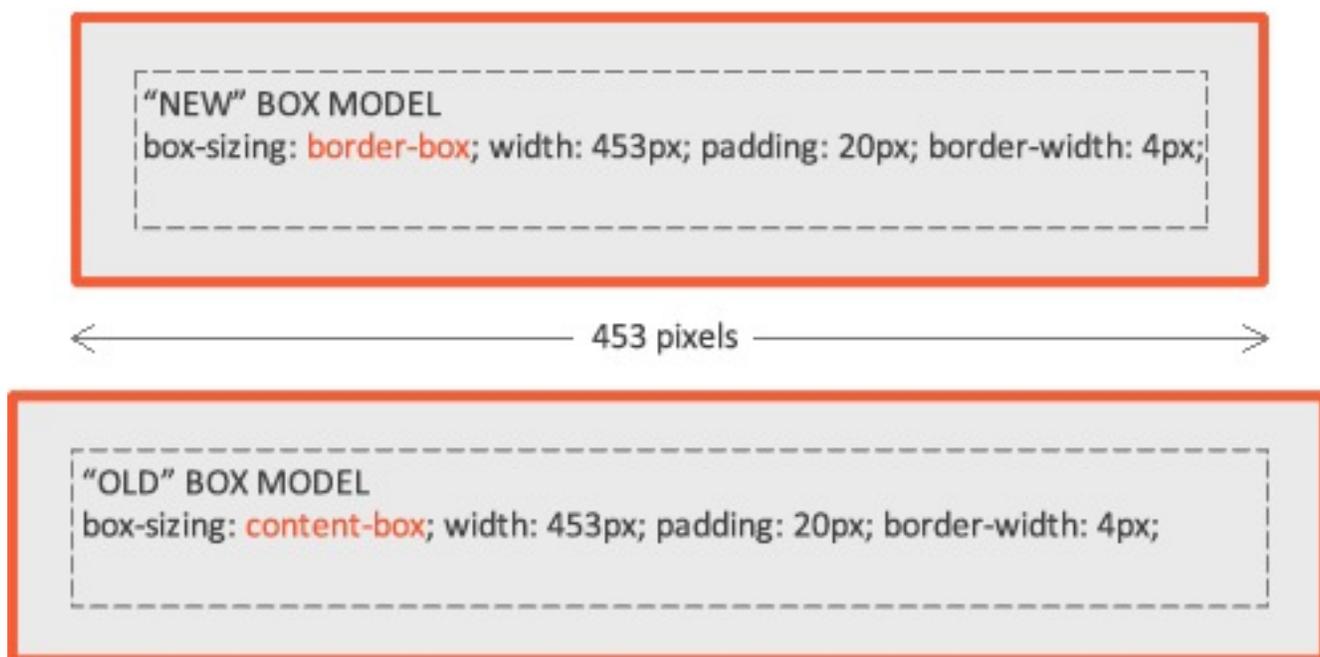
Aids in: efficiency

In addition to the `div`-conserving properties we've already talked about, the [`box-sizing`](#) property can also help limit your `div` use in certain situations.

In the traditional W3C box model of CSS 2.1, the value you declare for a width or height controls the width or height of the *content area* only, and then the padding and border are *added* onto it. (This is called the content-box model.) If you've worked with CSS for a while, you're probably used to the content-box box model and don't really think much about it. But, it can lead you to add extra `div`s from time to time. For instance, if you want to set a box's width and padding in different units of measurement from each other, like `ems` for the width and `pixels` for the padding, it's often easiest to nest another `div` and apply the padding to this instead, to make sure you know how much total space the box will take up. In small doses, nesting additional `div`s simply to add padding or borders is not a great sin. But in

complicated designs, the number of extra `div`s can really add up, which adds to both your development time and the file size of the HTML and CSS.

Setting the new `box-sizing` property to `border-box` instead of `content-box` solves this problem so you can get rid of all those extra `div`s. When a box is using the `border-box` box model, the browser will *subtract* the padding and border from the width of the box instead of adding it. You always know that the total space the box takes up equals the width value you've declared.



In the traditional box model (bottom image), padding and border are added onto the declared width. By setting `box-sizing` to `border-box` (top image), the padding and border are subtracted from the declared width.

The `box-sizing` property has good [browser support](#), with the exception of IE 6 and IE 7. Unlike the more decorative CSS3 properties, however, lack of support for `box-sizing` could cause your entire layout to fall apart. You'll have to determine how serious the problem would be in your

particular case, whether it's worth living with or hacking, or whether you should avoid using `box-sizing` for now.

CSS3 Pseudo-Classes and Attribute Selectors

Aids in: progressive enhancement, efficiency, modularity, rich typography

CSS has several really useful [selectors](#) that are only now coming into common use. Many of these are new in CSS3, but others have been around since CSS2, just not supported by all browsers (read: IE) until recently, and thus largely ignored. IE still doesn't support them all, but they can be used to add non-essential visual effects.

Taking advantage of these newer, more advanced selectors can improve your efficiency and make your pages more modular because they can reduce the need for lots of extra classes, `divs`, and `spans` to create the effects you want to see. Some selectors even make certain effects possible that you can't do with classes, such as styling the first line of a block of text differently. These types of visual effects can improve the typography of your site and aid progressive enhancement.

HTML5

Although this article is focused on modern CSS techniques, you can't have great CSS-based Web pages without great markup behind them. Although [HTML5](#) is still in development, and although debate continues about its strengths and weaknesses, some Web developers are already using it in their Web pages. While HTML 4.01 and XHTML 1.0 are still great choices for the markup of your pages, it's a good idea to start learning what HTML5 has to offer so you can work with it comfortably in the future and perhaps start taking advantage of some of its features now. So, here is a brief

overview of how HTML5 can help with our five modern CSS-based Web design characteristics (progressive enrichment, adaptive to diverse users, modular, efficient, typographically rich).

Note: Many of these techniques are not supported in enough browsers yet to make their benefits really tangible, so think of this section as, perhaps, “here’s how HTML5 *can* aid these five characteristics in the *future*.”

New Structural Markup

Aids: adaptability, modularity, efficiency

HTML5 introduces a number of new semantic elements that can add more structure to your markup to increase modularity. For instance, inside your main content `div` you can have several `article` elements, each a standalone chunk of content, and each can have its own `header`, `footer`, and heading hierarchy (`h1` through `h6`). You can further divide up an `article` element with `section` elements, again with their own `headers` and `footers`. Having clearer, more semantic markup makes it easier to shuffle independent chunks of content around your site if needed, or syndicate them through RSS on other sites and blogs.

In the future, as user agents build features to take advantage of HTML5, these new elements could also make pages more adaptable to different user scenarios. For instance, Web pages or browsers could generate table of contents based on the richer hierarchy provided by HTML5, to assist navigation within a page or across a site. Assistive technology like screen readers could use the elements to help users jump around the page to get straight to the important content without needing “skip nav” links.

Although many of these benefits won't be realized until some unforeseen time in the future, you can start adding these new elements now, so that as soon as tools pop up that can take full advantage of them, you'll be ready. Even if your browser doesn't recognize an element, you can still style it — that's standard browser behavior. Well, in every browser but IE. Luckily, you can easily trick IE into styling these elements using a very simple piece of JavaScript, handily provided by [Remy Sharp](#).

Of course, you usually can't depend on all your users having JavaScript enabled, so the very safest and most conservative option is to not use these new structural elements just yet, but use `div`s with corresponding `class` names as if they were these new elements. For instance, where you would use an `article` element, use a `div` with a `class` name of "article." You can still use the HTML5 doctype — HTML5 pages work fine in IE, as long as you don't use the new elements. You can then later convert to the new HTML5 elements easily if desired, and in the meantime, you can take advantage of the more detailed [HTML5 validators](#). Also, using these standardized class names can make updating the styles easier for both you and others in your team, and having consistent naming conventions across sites makes it easier for users with special needs to set up user style sheets that can style certain elements in a needed way.

Reducing JavaScript and Plug-in Dependence

Aids in: adaptability, efficiency

A number of the new elements and features in HTML5 make effects possible with pure markup that used to be possible only with JavaScript or various third-party plug-ins, like Flash or Java. By removing the need for JavaScript and plug-ins, you can make your pages work on a wider variety

of devices and for a wider variety of users. You may also make your development process quicker and more efficient, since you don't have to take the time to find the right script or plug-in and get it all set up. Finally, these techniques may be able to boost the speed of your pages, since extra files don't have to be downloaded by the users. (On the other hand, some may decrease performance, if the built-in browser version is slower than a third-party version. We'll have to wait and see how browsers handle each option now and in the future.)

Some of the features that reduce JavaScript and plug-in dependence are:

- **New form elements and attributes.** HTML5 offers a bunch of new `input` types, such as `email`, `url`, and `date`, that come with built-in client-side validation without the need for JavaScript. There are also many new form attributes that can accomplish what JavaScript used to be required for, like `placeholder` to add suggestive placeholder text to a field or `autofocus` to make the browser jump to a field. The new `input` types degrade to regular inputs in browsers that don't support them, and the new attributes are just ignored, so it doesn't hurt unsupported browsers to start using them now. Of course, you'll have to put in fallback JavaScript for unsupported browsers, negating the "no JavaScript" benefits for the time being. (Or, depend on server-side validation—which you always ought to have in place as a backup behind client-side validation anyway—to catch the submissions from unsupported browsers.) Still, they offer a nice usability boost for users with the most up to date browsers, so they're good for progressive enhancement.

- **The canvas element.** The canvas element creates a blank area of the screen that you can create drawings on with JavaScript. So, it *does* require the use of JavaScript, *but* it removes the need for Flash or Java plug-ins. It's supported in every major browser but IE, but you can make it work in IE easily using the [ExplorerCanvas](#) script.
- **The video and audio elements.** HTML5 can embed [video](#) and [audio](#) files directly, just as easily as you would add an image to a page, without the need for any additional plug-ins.

Date and Time (datetime)

: UTC

Week	Mon	Tue	Wed	Thu	Fri	Sat	Sun
13	29	30	31	<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>
14	<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>	<u>9</u>	<u>10</u>	<u>11</u>
15	<u>12</u>	<u>13</u>	<u>14</u>	<u>15</u>	<u>16</u>	<u>17</u>	<u>18</u>
16	<u>19</u>	<u>20</u>	<u>21</u>	<u>22</u>	<u>23</u>	<u>24</u>	<u>25</u>
17	<u>26</u>	<u>27</u>	<u>28</u>	<u>29</u>	<u>30</u>	1	2
18	3	4	5	6	7	8	9

Today None

Some of the new input types in HTML5 will bring up widgets, such as the calendar date picker seen with the `datetime` input type in Opera, without needing any JavaScript. ([HTML5 input types test page](#))

IE Filtering

Aids in: progressive enhancement

IE 6 doesn't seem to be going away anytime soon, so if you want to really make sure your pages are progressively enhanced, you're going to have to learn how to handle it. Beyond ignoring the problem or blocking IE 6 altogether, there are a number of stances you can take:

- **Use [conditional comments](#) to fix IE's bugs:** You can create separate style sheets for each version of IE you're having problems with and make sure only that version sees its sheet. The IE sheets contain only a few rules with hacks and workarounds that the browser needs.
- **Hide all main styles from IE and feed it very minimal styles only:** This is another conditional comment method, but instead of fixing the bugs, it takes the approach of hiding all the complex CSS from IE 6 to begin with, and only feeding it very simple CSS to style text and the like. Andy Clarke calls this [Universal Internet Explorer 6 CSS](#).
- **Use JavaScript to "fix" IE:** There are a number of scripts out there that can make IE 6 emulate CSS3, alpha-transparent PNGs, and other things that IE 6 doesn't support. Some of the most popular are [ie7-js](#), [Modernizr](#), and [ie-css3.js](#).

Flexible Layouts

Aids in: adaptability

One of the main ways you can make your sites adaptable to your users' preferences is to create flexible instead of fixed-width layouts. We've already gone over how media queries can make your pages more

adaptable to different viewport widths, but creating liquid, elastic, or resolution-dependent layouts can be used instead of or in conjunction with media queries to further optimize the design for as large a segment of your users as possible.

- **Liquid layouts:** Monitor sizes and screen resolutions cover a much larger range than they used to, and mobile devices like the iPhone and iPad let the user switch between portrait and landscape mode, changing their viewport width on the fly. Liquid layouts, also called fluid, change in width based on the user's viewport (e.g., window) width so that the entire design always fits on the screen without horizontal scrollbars appearing. The `min-width` and `max-width` properties and/or media queries can and should be used to keep the design from getting too stretched out or too squished at extreme dimensions.
- **Elastic layouts:** If you want to optimize for a particular number of text characters per line, you can use an elastic layout, which changes in width based on the user's text size. Again, you can use `min-` and `max-width` and/or media queries to limit the degree of elasticity.
- **Resolution-dependent layouts:** This type of layout, also called adaptive layout, is similar to media queries, but uses JavaScript to switch between different style sheets and rearrange boxes to accommodate different viewport widths.

Layout Grids

Aids in: modularity, efficiency

Designing on a grid of (usually invisible) consistent horizontal and vertical lines is not new — it goes back for centuries — but its application to Web

design has gained in popularity in recent years. And for good reason: a layout grid can create visual rhythm to guide the user's eye, make the design look more clean and ordered, and enforce design consistency.

Grids can also make your designs more modular and your development more efficient because they create a known, consistent structure into which you can easily drop new elements and rearrange existing ones without as much thought and time as it would take in a non-grid layout. For instance, all of your elements must be as wide as your grid's column measurement, or some multiple of it, so you can easily move an element to another spot on the page or to another page and be assured that it will fit and look consistent with the rest of the design. At worst, you'll need to adjust the other elements' widths around it to a different multiple of the column measurements to get the new element to fit, but even this is not too work-intensive, as there is only a handful of pre-determined widths that any element can have.



All of the content of [The New York Times site](#) falls into a grid of five columns, plus a thin column on the left for navigation.

Efficient CSS Development Practices

Aids in: modularity, efficiency

Layout grids and many of the CSS3 techniques we've gone over have the side benefit of making your CSS more modular and helping you write and maintain CSS more efficiently. There are also a few CSS development practices that you can use with *any* of the techniques we've already covered in order to reduce the time it takes you to write the CSS for those techniques in the first place, as well as save you time reusing components in your pages.

CSS Frameworks

A CSS framework is a library of styles that act as building blocks to create the standard pieces you might need in your site. While CSS frameworks differ greatly in depth and breadth, most popular, publicly-distributed frameworks contain some sort of layout grid, as well as standard styles for text, navigation, forms, images, and more. It's a good idea to create your own CSS framework, perhaps based on one of the most popular ones; it can be as simple as standardizing the IDs and classes you tend to use on every project and creating a starter style sheet for yourself.

Good CSS frameworks provide you with a solid starting point for your designs, cutting down your time spent developing, testing, tweaking, and updating. They can also reduce the time others (your team members or those who inherit your sites) spend modifying your CSS, as everyone is working from a standard set of conventions. Frameworks can make your designs more modular by giving you a standard set of classes that can be reused from page to page easily, breaking the styles down into separate sheets that can be applied independently to pages on an as-needed basis, or allowing you to plug in various types of content without needing to invent new classes for it.

But, frameworks have their share of problems too. For instance, publicly-distributed (as opposed to your own private) frameworks tend to have large file sizes, as they need to work for any type of site with any type of content; if they're separated into multiple sheets, they can further damage page speed since every HTTP request takes time. We won't get into the full list of pros and cons here, but there are ways to work around many of them, so check out the following articles for the details.

Object-oriented CSS (OOCSS)

Nicole Sullivan coined the term [object-oriented CSS \(OOCSS\)](#) for her method of creating self-contained chunks of HTML (modules) that can be reused anywhere in the page or site and that any class can be applied to. Some of the main principles of OOCSS are:

- using primarily classes instead of IDs
- creating default classes with multiple, more specific classes added on to elements
- avoiding dependent selectors and class names that are location-specific
- leaving dimensions off module styles so the modules can be moved anywhere and fit
- styling containers separately from content

OOCSS aims to make your CSS development more efficient, as well as to make the CSS itself more modular and less redundant, which reduces file sizes and loading speed.

CSS Generation

When it comes to writing CSS quickly, what could be quicker than having some piece of software write it for you? Now, please don't think that I'm advocating not learning CSS and having a tool write a complete style sheet for you. That is a bad, bad idea. But, there are some quality tools out there that can give you a *headstart* with your CSS, just to shave a *little* time off the front of your CSS development process. Most good CSS generators are

focused on creating styles for one particular area of your design, such as the layout structure or type styles, not the whole style sheet.

There are far too many tools to link to individually here, so remember when you're finding your own tools to carefully review the CSS it outputs. If it's invalid, bloated, or just plain ugly, don't use the tool!

CSS Performance

Aids in: efficiency

Your efficiently *created* CSS-based Web sites also need to *perform* as efficiently as possible for your users. Many of the CSS3 techniques we've covered can reduce file sizes and HTTP requests to increase the speed of your pages. There are some additional CSS techniques you can use to boost performance.

CSS Compression

Writing clean CSS that takes advantage of shorthand properties, grouped selectors, and other efficient syntax is nothing new, but it remains very important for improving performance. There are also tricks some CSS developers employ to further reduce CSS file sizes, such as writing each rule on one line to reduce all the line breaks. Although you can do some of this manually, there are a number of tools that can optimize and compress your CSS for you.

CSS Sprites

CSS Sprites is a CSS technique named by [Dave Shea](#) of combining many (or all) of your site's images into one big master image and then using

`background-position` to shift the image around to show only a single image at a time. This greatly improves your pages' performance because it greatly reduces the number of HTTP requests to your server. This is not a new technique, but it's becoming increasingly important in modern CSS-based Web sites as page performance becomes more and more important.



The [Apple site](#) uses CSS sprites for various states of its navigation bar.

Font Embedding and Replacement

Aids in: progressive enhancement, rich typography

Until recently, Web designers were limited to working with the fonts on their end users' machines. We now have a number of techniques and technologies that make unique but still readable and accessible text possible.

The @font-face Rule

The [@font-face rule](#), part of CSS3, allows you to link to a font on your server, called a "web font," just as you can link to images, and displays text on your site in this font. You can now make use of your beautiful, unique fonts instead of just the fonts that most people already have installed on

their machines. Fortunately, `@font-face` has good browser support. But alas, it's not as simple as that. Different browsers support different types of fonts, different platforms and browsers anti-alias very differently, you can get a flash of unstyled text before the font loads, your font may not allow `@font-face` embedding in its license, and on and on it goes.

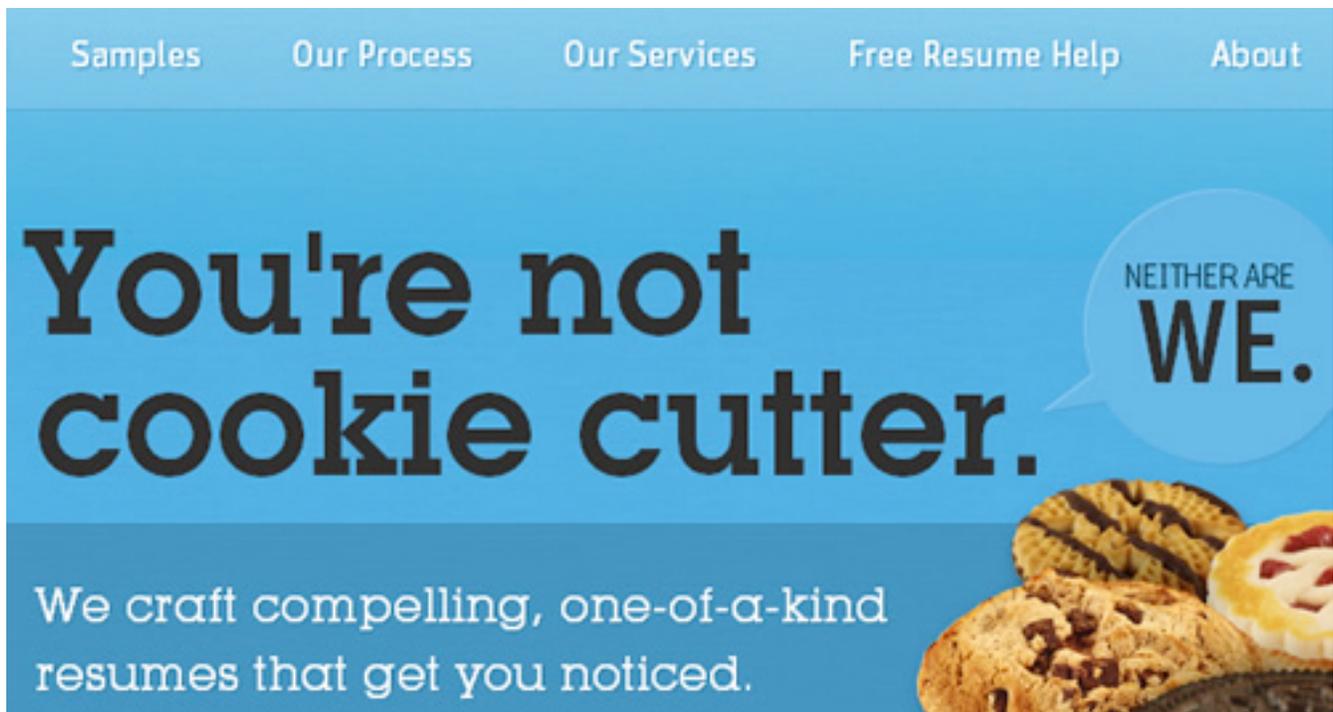
April 4, 2010

HTML5 Hearts iPad

The iPad is increasing the mind-share of HTML to the businesses, and tech-minds at breakneck speed that will make most enduring web-standardists jealous.

I've been watching HTML develop over the last 10 years – at a time when Jeffery Zeldman spent years developing the standards community, cajoling the business community into understanding the importance of standards.

[Sam Howat's site](#) uses `@font-face` to get attractive non-standard fonts into the headings and intro blocks of text.



Blue Sky Resumes uses `@font-face` extensively in headings, feature copy, and the main nav bar of the site.

Other Font Embedding and Replacement Techniques

If the pure CSS solution of `@font-face` is making your head spin, you can use a font embedding service or font replacement technique.

- **Font embedding services:** There are a number of third-party font embedding services available that make use of `@font-face`, such as [Typekit](#), but make implementation easier by helping you work around the browser differences. They also all get around the legal issue of font embedding by providing you with a set of fonts that are licensed for this type of use and impossible or difficult for end users to steal. Most of these services are not free, but some have free options that give you access to a limited set of fonts.

-
- **Font replacement techniques:** These free techniques, such as [sIFR](#) and [Cufón](#), do not make use of `@font-face`, but instead use scripting and/or Flash to display fonts that are not on the user's machine. None of them directly address the licensing issue, but none of them link directly to ready-to-use fonts, so copyright legality is not clear-cut.

Conclusion

You're now equipped with the basic knowledge and a slew of links to create modern CSS-based Web pages that are progressively enriched, adaptive to diverse users, modular, efficient, and typographically rich. Go out and create great, modern work!

How to Use CSS3 Pseudo-Classes

Richard Shepherd

CSS3 is a wonderful thing, but it's easy to be bamboozled by the transforms and animations (many of which are vendor-specific) and forget about the nuts-and-bolts selectors that have also been added to the specification. A number of powerful new pseudo-selectors (16 are listed in the [latest W3C spec](#)) enable us to select elements based on a range of new criteria.

Before we look at these new CSS3 pseudo-classes, let's briefly delve into the dusty past of the Web and chart the journey of these often misunderstood selectors.

A Brief History Of Pseudo-Classes

When the [CSS1](#) spec was completed back in 1996, a few pseudo-selectors were included, many of which you probably use almost every day. For example:

- `:link`
- `:visited`
- `:hover`
- `:active`

Each of these states can be applied to an element, usually `<a>`, after which comes the name of the pseudo-class. It's amazing to think that these pseudo-classes arrived on the scene before [HTML4](#) was published by the W3C a year later in December 1997.

CSS2 Arrives

Hot on the heels of CSS1 was [CSS2](#), whose recommended spec was published just two years later in May 1998. Along with exciting things like positioning were new pseudo-classes: `:first-child` and `:lang()`.

`:lang`

There are a couple of ways to indicate the language of a document, and if you're using [HTML5](#), it'll likely be by putting `<html lang="en">` just after the doc type (specifying your local language, of course). You can now use `:lang(en)` to style elements on a page, which is useful when the language changes dynamically.

`:first-child`

You may have already used `:first-child` in your documents. It is often used to add or remove a top border on the first element in a list. Strange, then, that it wasn't accompanied by `:last-child`; we had to wait until CSS3 was proposed before it could meet its brother.

Why Use Pseudo-Classes?

What makes pseudo-classes so useful is that they allow you to style content dynamically. In the `<a>` example above, we are able to describe how links are styled when the user interacts with them. As we'll see, the new pseudo-classes allow us to dynamically style content based on its position in the document or its state.

Sixteen new pseudo-classes have been introduced as part of the W3C's [CSS Proposed Recommendation](#), and they are broken down into four groups: structural pseudo-classes, pseudo-classes for the states of UI elements, a target pseudo-class and a negation pseudo-class.

Let's now run through the 16 new pseudo-selectors one at a time and see how each is used. I'll use the same notation for naming classes that the W3C uses, where E is the element, n is a number and s is a selector.

Sample Code

For many of these new selectors, I'll also refer to some sample code so that you can see what effect the CSS has. We'll take a regular form and make it suitable for an iPhone using our new CSS3 pseudo-classes.

Note that we could arguably use ID and class selectors for much of this form, but it's a great opportunity to take our new pseudo-classes out for a spin and demonstrate how you might use them in a real-world example.

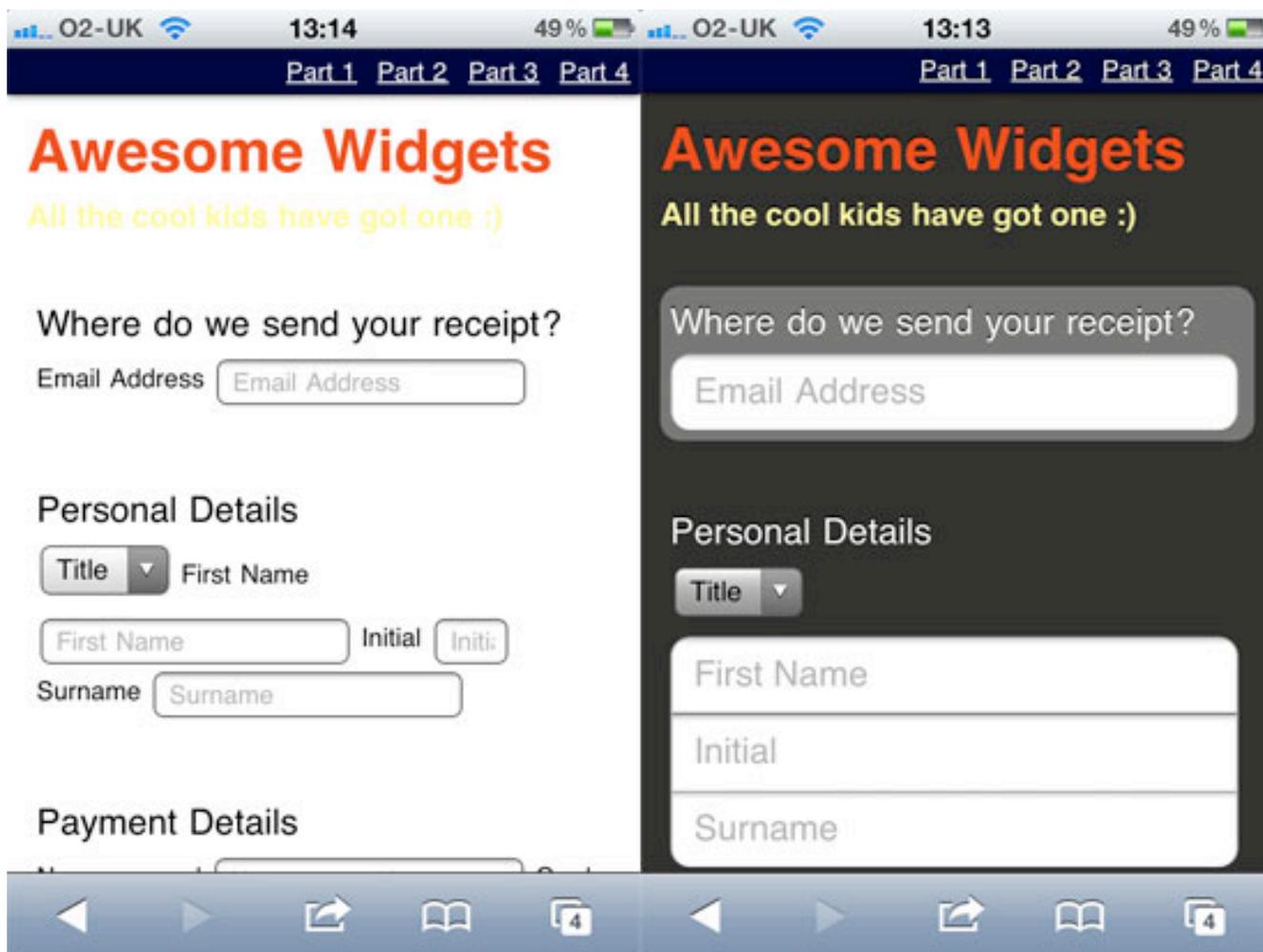
Here's the HTML:

```
1 <form>
2 <hgroup>
3 <h1>Awesome Widgets</h1>
4 <h2>All the cool kids have got one :)</h2>
5 </hgroup>
6 <fieldset id="email">
7 <legend>Where do we send your receipt?</legend>
8 <label for="email">Email Address</label>
9 <input type="email" name="email" placeholder="Email
  Address" />
10 </fieldset>
11
12 <fieldset id="details">
13 <legend>Personal Details</legend>
14 <select name="title" id="field_title">
15 <option value="" selected="selected">Title</option>
```

```
16 <option value="Mr">Mr</option>
17 <option value="Mrs">Mrs</option>
18 <option value="Miss">Miss</option>
19 </select>
20
21 <label for="firstname">First Name</label>
22 <input name="firstname" placeholder="First Name" />
23
24 <label for="initial">Initial</label>
25 <input name="initial" placeholder="Initial" size="3" /
  >
26
27 <label for="surname">Surname</label>
28 <input name="surname" placeholder="Surname" />
29 </fieldset>
30
31 <fieldset id="payment">
32 <legend>Payment Details</legend>
33
34 <label for="cardname">Name on card</label>
35 <input name="cardname" placeholder="Name on card" />
36
37 <label for="cardnumber">Card number</label>
38 <input name="cardnumber" placeholder="Card number" />
39
40 <select name="cardType" id="field_cardType">
41 <option value="" selected="selected">Select Card
  Type</option>
42 <option value="1">Visa</option>
43 <option value="2">American Express</option>
44 <option value="3">MasterCard</option>
```

```
45 </select>
46
47 <label for="cardExpiryMonth">Expiry Date</label>
48 <select id="field_cardExpiryMonth"
    name="cardExpiryMonth">
49 <option selected="selected" value="mm">MM</option>
50 <option value="01">01</option>
51 <option value="02">02</option>
52 <option value="03">03</option>
53 <option value="04">04</option>
54 <option value="05">05</option>
55 <option value="06">06</option>
56 <option value="07">07</option>
57 <option value="08">08</option>
58 <option value="09">09</option>
59 <option value="10">10</option>
60 <option value="11">11</option>
61 <option value="12">12</option>
62 </select> /
63 <select id="field_cardExpiryYear"
    name="cardExpiryYear">
64 <option value="yyyy">YYYY</option>
65 <option value="2011">11</option>
66 <option value="2012">12</option>
67 <option value="2013">13</option>
68 <option value="2014">14</option>
69 <option value="2015">15</option>
70 <option value="2016">16</option>
71 <option value="2017">17</option>
72 <option value="2018">18</option>
73 <option value="2019">19</option>
```

```
74 </select>
75
76 <label for="securitycode">Security code</label>
77 <input name="securitycode" type="number"
  placeholder="Security code" size="3" />
78
79 <p>Would you like Insurance?</p>
80 <input type="radio" name="Insurance"
  id="insuranceYes" />
81 <label for="insuranceYes">Yes Please!</label>
82 <input type="radio" name="Insurance"
  id="insuranceNo" />
83 <label for="insuranceNo">No thanks</label>
84
85 </fieldset>
86
87 <fieldset id="submit">
88 <button type="submit" name="Submit" disabled>Here I
  come!</button>
89 </fieldset>
90 </form>
```



Our form, before and after.

1. Structural Pseudo-Classes

According to the W3C, structural pseudo-classes do the following:

... permit selection based on extra information that lies in the document tree but cannot be represented by other simple selectors or combinators.

What this means is that we have selectors that have been turbo-charged to dynamically select content based on its position in the document. So let's start at the beginning of the document, with `:root`.

E:root

The `:root` pseudo-class selects the root element on the page. Ninety-nine times out of a hundred, this will be the `<html>` element. For example:

```
1 | :root { background-color: #fcfcfc; }
```

It's worth noting that you could style the `<html>` element instead, which is perhaps a little more descriptive:

```
1 | html { background-color: #fcfcfc; }
```

iPhone Form Example

Let's move over to our sample code and give the document some basic text and background styles:

```
1 | :root {  
2 | color: #fff;  
3 | text-shadow: 0 -1px 0 rgba(0,0,0,0.8);  
4 | background: url(.../images/background.png) no-repeat  
   | #282826; }
```

E:nth-child(n)

The `:nth-child()` selector might require a bit of experimentation to fully understand. The easiest implementation is to use the keywords `odd` or `even`, which are useful when displaying data that consists of rows or columns. For example, we could use the following:

```
1 | ul li:nth-child(odd) {  
2 | background-color: #666;  
3 | color: #fff; }
```

This would highlight every other row in an unordered list. You might find this technique extremely handy when using tables. For example:

```
1 | table tr:nth-child(even) { ... }
```

The `:nth-child` selector can be much more specific and flexible, though. You could select only the third element from a list, like so:

```
1 | li:nth-child(3) { ... }
```

Note that `n` does not start at zero, as it might in an array. The first element is `:nth-child(1)`, the second is `:nth-child(2)` and so on.

We can also use some simple algebra to make things even more exciting. Consider the following:

```
1 | li:nth-child(2n) { ... }
```

Whenever we use `n` in this way, it stands for all positive integers (until the document runs out of elements to select!). In this instance, it would select the following list items:

- Nothing (2×0)
- 2nd element (2×1)
- 4th element (2×2)
- 6th element (2×3)
- 8th element (2×4)

This actually gives us the same thing as `nth-child(even)`. So, let's mix things up a bit:

```
1 | li:nth-child(5n) { ... }
```

This gives us:

- Nothing (5×0)
- 5th element (5×1)
- 10th element (5×2)
- 15th element (5×3)
- 20th element (5×4)
- etc.

Perhaps this would be useful for long lists or tables, perhaps not. We can also add and subtract numbers in this equation:

```
1 | li:nth-child(4n + 1) { ... }
```

This gives us:

- 1st element ($(4 \times 0) + 1$)
- 5th element ($(4 \times 1) + 1$)
- 9th element ($(4 \times 2) + 1$)
- 13th element ($(4 \times 3) + 1$)
- 17th element ($(4 \times 4) + 1$)
- etc.

[SitePoint points out](#) an interesting quirk here. If you set n as negative, you'll be able to select the first x number of items like so:

```
1 | li:nth-child(-n + x) { ... }
```

Let's say you want to select the first five items in a list. Here's the CSS:

```
1 | li:nth-child(-n + 5) { ... }
```

This gives us:

- 5th element (-0 + 5)
- 4th element (-1 + 5)
- 3rd element (-2 + 5)
- 2nd element (-3 + 5)
- 1st element (-4 + 5)
- Nothing (-5 + 5)
- Nothing (-6 + 5)
- etc.

If you're listing data in order of popularity, then highlighting, say, the top 10 entries might be useful.

[WebDesign & Such](#) has created a [demo of zebra striping](#), which is a perfect example of how you might use `nth-child` in practice.



This is an example of using CSS3 to apply zebra striping to a Table. It doesn't work in Internet Explorer, but that browser sucks anyway. [Click here for the tutorial.](#)

Text

Zebra striping a table with CSS3.

If none of your tables need styling, then you could do what [Webvisionary Awards](#) has done and use `:nth-child` to style alternating sections of its website. Here's the CSS:

```
1 | section > section:nth-child(even) {  
2 | background: rgba (255, 255, 255, .1)  
3 | url ("../images/hr-damaged2.png") 0 bottom no-repeat;  
4 | }
```

The effect is subtle on the website, but it adds a layer of detail that would be missed in older browsers.



The :nth-child selectors in action on Webvisionary Awards.

iPhone Form Example

We could use `:nth-child` in a few places in our iPhone form example, but let's focus on one. We want to hide the labels for the first three fieldsets from view and use the placeholder text instead. Here's the CSS:

```
1 | form:nth-child(-n+3) label { display: none; }
```

Here, we're looking for the first three children of the `<form>` element (which are all fieldsets in our code) and then selecting the label. We then hide these labels with `display: none;`.

E:nth-last-child(n)

Not content with confusing us all with the `:nth-child()` pseudo-class, the clever folks over at the W3C have also given us `:nth-last-child(n)`. It operates much like `:nth-child()` except in reverse, counting from the last item in the selection.

```
1 |li:nth-last-child(1) { ... }
```

The above will select the last element in a list, whereas the following will select the penultimate element:

```
1 |li:nth-last-child(2) { ... }
```

Of course, you could create other rules, like this one:

```
1 |li:nth-last-child(2n+1) { ... }
```

But you would more likely want to use the following to select the last five elements of a list (based on the logic discussed above):

```
1 |li:nth-last-child(-n+5) { ... }
```

If this still doesn't make much sense, [Lea Verou](#) has created a useful [CSS3 structural pseudo-class selector tester](#), which is definitely worth checking out.

CSS3 structural pseudo-class selector tester

Helps you understand how the `nth-child`, `nth-last-child`, `nth-of-type` and `nth-last-of-type` CSS3 selectors work. Uses the native browser algorithm, so you're out of luck if you're on IE (but if you're on IE, you have more serious issues to sort out anyway)

* \updownarrow :nth-child \updownarrow (3n+2)

dt (1)
dt (2)
dd (3)
dd (4)
dt (5)
dd (6)
dt (7)
dd (8)
dd (9)

CSS3 structural pseudo-class selector tester.

iPhone Form Example

We can use `:nth-last-child` in our example to add rounded corners to our input for the “Card number.” Here’s our CSS, which is overly specific but gives you an idea of how we can chain pseudo-selectors together:

```
1 fieldset:nth-last-child(2) input:nth-last-of-type(3) {  
2 border-radius: 10px; }
```

We first grab the penultimate fieldset and select the input that is third from last (in this case, our “Card number” input). We then add a `border-radius`.

:nth-of-type(n)

Now we'll get even more specific and apply styles only to particular *types* of element. For example, let's say you wanted to style the first paragraph in an article with a larger font. Here's the CSS:

```
1 | article p:nth-of-type(1) { font-size: 1.5em; }
```

Perhaps you want to align every other image in an article to the right, and the others to the left. We can use keywords to control this:

```
1 | article img:nth-of-type(odd) { float: right; }
2 | article img:nth-of-type(even) { float: left; }
```

As with `:nth-child()` and `:nth-last-child()`, you can use algebraic expressions:

```
1 | article p:nth-of-type(2n+2) { ... }
2 | article p:nth-of-type(-n+1) { ... }
```

It's worth remembering that if you need to get this specific about targeting elements, then using descriptive class names instead might be more useful.

Simon Foster has created a [beautiful infographic about his 45 RPM record collection](#), and he uses `:nth-of-type` to style some of the data. Here's a snippet from the CSS, which assigns a different background to each genre type:

```
1 | ul#genre li:nth-of-type(1) {
2 |   width: 32.9%;
3 |   background: url(images/orangenoise.jpg);
4 | }
5 | ul#genre li:nth-of-type(2) {
```

```
6 width:15.2%;
7 background:url(images/bluenoise.jpg);
8 }
9 ul#genre li:nth-of-type(3) {
10 width:13.1%;
11 background:url(images/greennoise.jpg);
12 }
```

And here's what it looks like on his website:



The :nth-of-type selectors on "For the Record."

iPhone Form Example

Let's say we want every second input element to have rounded corners on the bottom. We can achieve this with CSS:

```
1 input:nth-of-type(even) {  
2 border-bottom-left-radius: 10px;  
3 border-bottom-right-radius: 10px; }
```

In our example, we want to apply this only to the fieldset for payment, because the fieldset for personal details has three text inputs. We'll also get a bit tricky and make sure that we *don't* select any of the radio inputs.

Here's the final CSS:

```
1 #payment input:nth-of-type(even):not([type=radio]) {  
2 border-bottom-left-radius: 10px;  
3 border-bottom-right-radius: 10px;  
4 border-bottom: 1px solid #999;  
5 margin-bottom: 10px; }
```

We'll explain `:not` later in this article.

:nth-last-of-type(n)

Hopefully, by now you see where this is going: `:nth-last-of-type()` starts at the end of the selected elements and works backwards.

To select the last paragraph in an article, you would use this:

```
1 |article p:nth-last-of-type(1) { ... }
```

You might want to choose this selector instead of `:last-child` if your articles don't always end with paragraphs.

:first-of-type and :last-of-type

If `:nth-of-type()` and `:nth-last-of-type()` are too specific for your purposes, then you could use a couple of simplified selectors. For example, instead of this...

```
1 | article p:nth-of-type(1) {  
2 | font-size: 1.5em; }
```

... we could just use this:

```
1 | article p:first-of-type {  
2 | font-size: 1.5em; }
```

As you'd expect, `:last-of-type` works in exactly the same way but from the last element selected.

iPhone Form Example

We can use both `:first-of-type` and `:last-of-type` in our iPhone example, particularly when styling the rounded corners. Here's the CSS:

```
1 | fieldset input:first-of-type:not([type=radio]) {  
2 | border-top-left-radius: 10px;  
3 | border-top-right-radius: 10px; }  
4 |  
5 | fieldset input:last-of-type:not([type=radio]) {  
6 | border-bottom-left-radius: 10px;  
7 | border-bottom-right-radius: 10px; }
```

The first line of CSS adds a top rounded border to all `:first-of-type` inputs in a fieldset that *aren't* radio buttons. The second line adds the bottom rounded border to the last input element in a fieldset.

:only-of-type

There's one more `type` selector to look at: `:only-of-type()`. This is useful for selecting elements that are the only one of their kind in their parent element.

For example, consider the difference between this CSS selector...

```
1 | p {  
2 | font-size: 18px; }
```

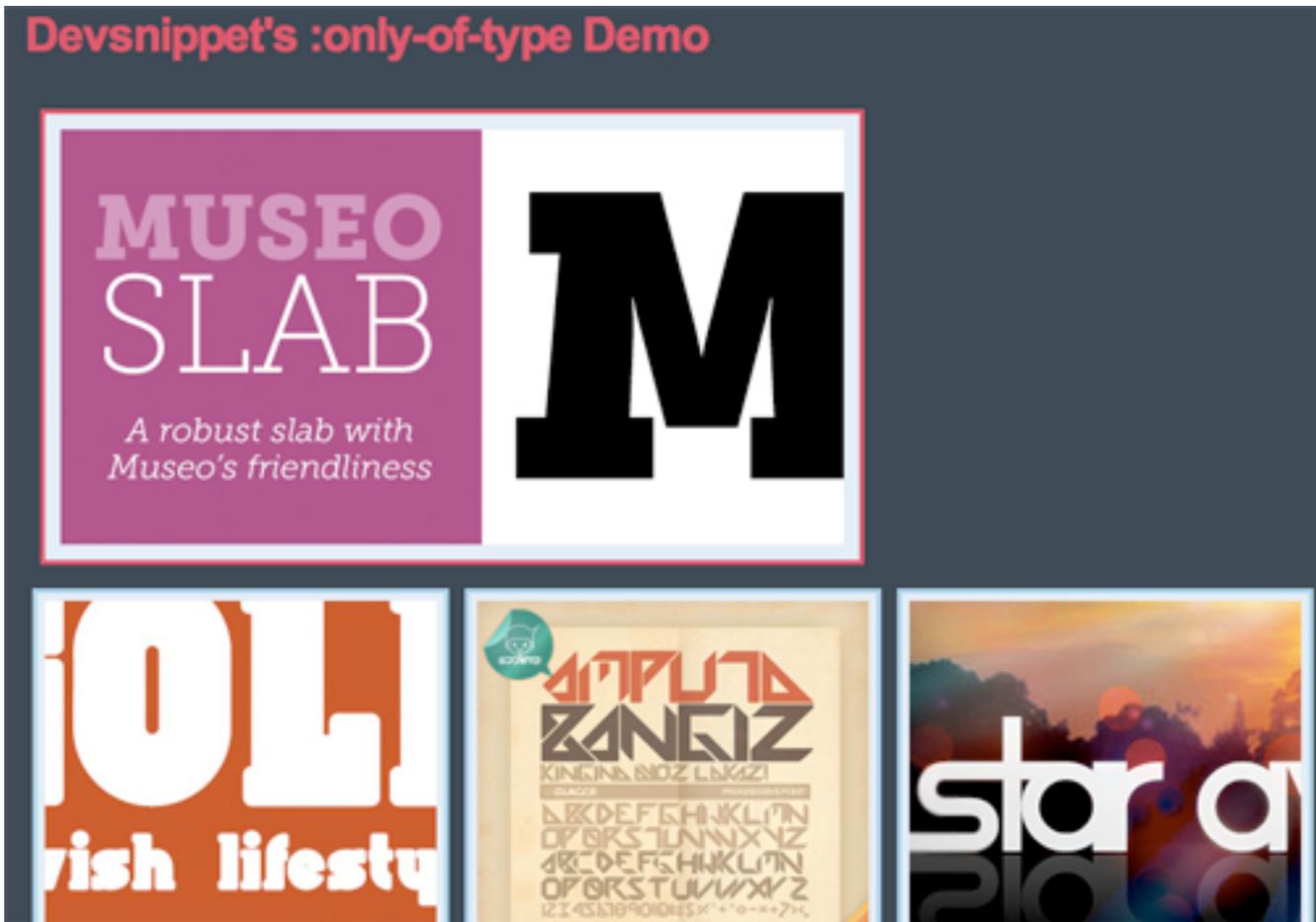
... and this one:

```
1 | p:only-of-type {  
2 | font-size: 18px; }
```

The first selector will style every paragraph element on the page. The second element will grab a paragraph that is the *only* paragraph in its parent.

This could be handy when you are styling content or data that has been dynamically outputted from a database and the query returns only one result.

[Devsnippet](#) has created a demo in which [single images are styled differently](#) from multiple images.



Devsnipet's demo for `:only-of-type`.

iPhone Form Example

In the case of our iPhone example, we can make sure that all inputs that are the only children of a fieldset have rounded corners on both the top and bottom. The CSS would be:

```
1 | fieldset input:only-of-type {  
2 | border-radius: 10px; }
```

:last-child

It's a little strange that `:first-child` was part of the CSS2 spec but that its partner in crime, `:last-child`, didn't appear until CSS3. It takes no expressions or keywords here; it simply selects the last child of its parent element. For example:

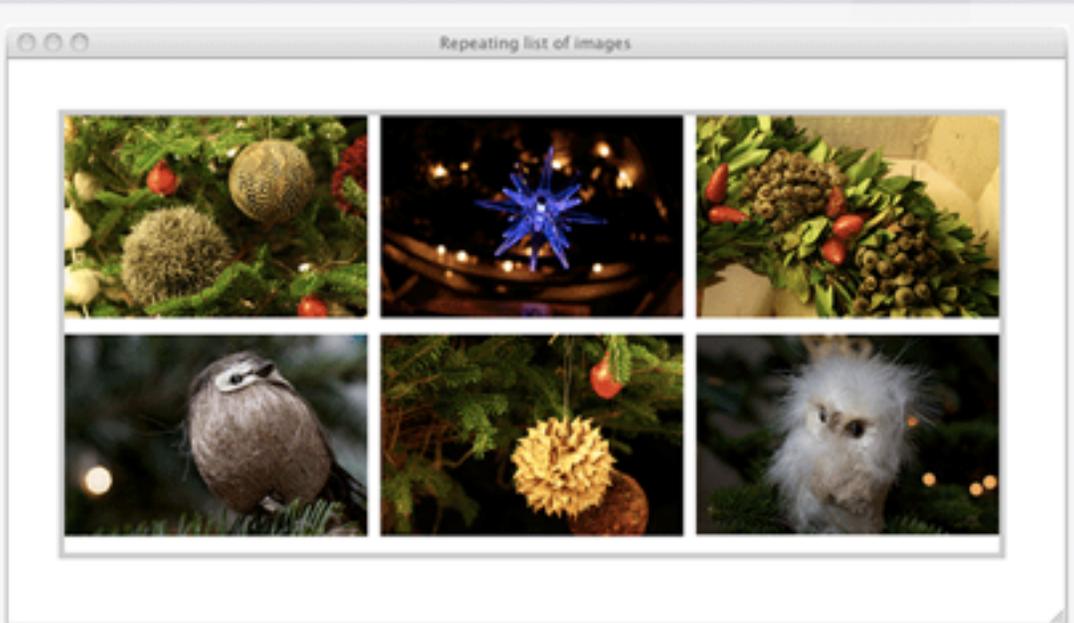
```
1 li {  
2 border-bottom: 1px solid #ccc; }  
3  
4 li:last-child {  
5 border-bottom: none; }
```

This is a useful way to remove bottom borders from lists. You'll see this technique quite often in WordPress widgets.

Rachel Andrew takes a more detailed look at `:last-child` and other CSS pseudo-selectors in her 24 Ways article "[Cleaner Code With CSS3 Selectors](#)." Rachel shows us how to use this selector to create a well-formatted image gallery without additional classes.

```
ul.gallery li:nth-child(3n) {  
    margin-right: 0;  
}
```

[View Example 6](#)



The CSS for :last-child in action, courtesy of Rachel Andrew.

:only-child

If an element is the only child of its parent, then you can select it with `:only-child`. Unlike with `:only-of-type`, it doesn't matter what type of element it is. For example:

```
1 |li:only-child { ... }
```

We could use this to select list elements that are the only list elements in their `` or `` parent.

:empty

Finally, in structural pseudo-classes, we have `:empty`. Not surprisingly, this selects only elements that have no children and no content. Again, this might be useful when dealing with dynamic content outputted from a database.

```
1 | #results:empty {  
2 | background-color: #fcc; }
```

You might use the above to draw the user's attention to an empty search results section.

2. The Target Pseudo-Class

:target

This is one of my favourite pseudo-classes, because it allows us to style elements on the page based on the URL. If the URL has an identifier (that follows an `#`), then the `:target` pseudo-class will style the element that shares the ID with the identifier. Take a URL that looks like this:

```
http://www.example.com/css3-pseudo-selectors#summary
```

The section with the id `summary` can now be styled like so:

```
1 | :target {  
2 | background-color: #fcc; }
```

This is a great way to style elements on pages that have been linked to from external content. You could also use it with internal anchors to highlight content that users have skipped to.

Perhaps the most impressive use of `:target` I've seen is [Corey Mwamba's Scrolling Site of Green](#). Corey uses some creative CSS3 and the `:target` pseudo-class to create [animated tabbed navigation](#). The demo contains some clever use of CSS3, illustrating how pseudo-classes are often best used in combination with other CSS selectors.



Corey's Scrolling Site of Green.

There's also an interesting example over at [Web Designer Notebook](#). In it, `:target` and Webkit animations are used to highlight blocks of text in target divs. Chris Coyier also creates a `:target`-based tabbing system at [CSS-Tricks](#).

iPhone Form Example

As you'll see on [my demo page](#), I've added a navigation bar at the top that skips down to different sections of the form. We can highlight any section the user jumps to with the following CSS:

```
1 :target {
2 background-color: rgba(255,255,255,0.3);
3
4 -webkit-border-radius:
5 10px;}
```

3. The UI Element States Pseudo-Classes

:enabled and :disabled

Together with `:checked`, `:enabled` and `:disabled` make up the three pseudo-classes for UI element states. That is, they allow you to style elements (usually form elements) based on their state. A state could be set by the user (as with `:checked`) or by the developer (as with `:enabled` and `:disabled`). For example, we could use the following:

```
1 input:enabled {
2 background-color: #dfd; }
3
4 input:disabled {
5 background-color: #fdd; }
```

This is a great way to give feedback on what users can and cannot fill in. You'll often see this dynamic feature enhanced with JavaScript.

iPhone Form Example

To illustrate `:disabled` in practice, I have disabled the form's "Submit" button in the HTML and added this line of CSS:

```
1 | :disabled {  
2 | color: #600; }
```

The button text is now red!

:checked

The third pseudo-class here is `:checked`, which deals with the state of an element such as a checkbox or radio button. Again, this is very useful for giving feedback on what users have selected. For example:

```
1 | input[type=radio]:checked {  
2 | font-weight: bold; }
```

iPhone Form Example

As a flourish, we can use CSS to highlight the text next to each radio button once the button has been pressed:

```
1 | input:checked + label {  
2 | text-shadow: 0 0 6px #fff; }
```

We first select any input that has been checked, and then we look for the very next `` element that contains our text. Highlighting the text with a simple `text-shadow` is an effective way to provide user feedback.

4. Negation Pseudo-Class

:not

This is another of my favorites, because it selects everything *except* the element you specify. For example:

```
1 | :not(footer) { ... }
```

This selects everything on the page that is not a footer element. When used with form inputs, they allow us to get a little sneakier:

```
1 | input:not([type=submit]) { ... }  
2 | input:not(disabled) { ... }
```

The first line selects every form input that's not a "Submit" button, which is useful for styling forms. The second selects all input elements that are not enabled; again useful for giving feedback on how to fill in a form.

iPhone User Example

You've already seen the `:not` selector in action. It's particularly powerful when chained with other CSS3 pseudo-selectors. Let's take a closer look at one example:

```
1 | fieldset input:not([type=radio]) {  
2 |   margin: 0;  
3 |   width: 290px;  
4 |   font-size: 18px;  
5 |   border-radius: 0;  
6 |   border-bottom: 0;  
7 |   border-color: #999;  
8 |   padding: 8px 10px; }
```

Here we are selecting all inputs inside fieldset elements that are *not* radio buttons. This is incredibly useful when styling forms because you will often want to style text inputs different from select boxes, radio buttons and "Submit" buttons.

What's Old Is New Again

Let's go back to the beginning of our story and the humble `a:link`. HTML5 arrived on the scene recently and brought with it an [exciting change](#) to the `<a>` element that gives the CSS3 pseudo-selector an additive effect.

An `<a>` element can now be wrapped around block-level elements, turning whole sections of your page into links (as long as those sections don't contain other interactive elements). Whereas JavaScript was once popular for making entire `<div>` elements clickable, you can now do so by wrapping sections in `<a>` tags, like so:

```
1 <a href="http://www.smashing-magazine.com ">
2 <div id="advert">
3 <hgroup>
4 <h1>Jackson's Widgets</h1>
5 <h2>The finest widgets in Kentucky</h2>
6 </hgroup>
7 <p>Buy Jackson's Widgets today,
8 and be sure of a trouble-free life for you,
9 your widget and your machinery.
10 Trusted and sold since 1896.</p>
11 </div>
12 </a>
```

The implication for CSS pseudo-selectors is that you can now style a `<div>` based on whether it is being hovered over (`a:hover`) or is active (`a:active`), like so:

```
1 a:hover #advert {
2 background-color: #f7f7f7; }
```

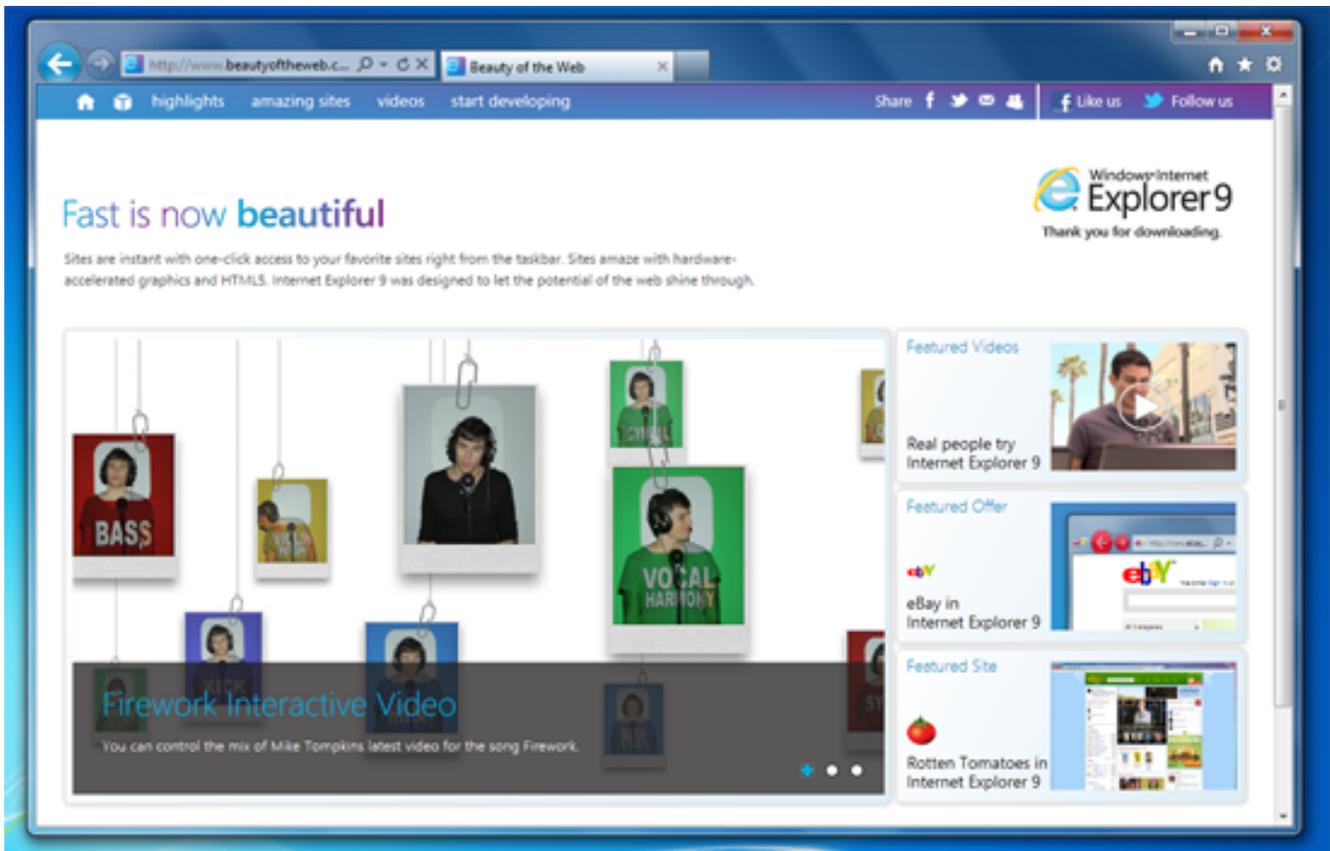
Anything that decreases JavaScript and increases semantic code has to be good!

Cross-Browser Compatibility

You had to ask, didn't you! Unbelievably, Internet Explorer 8 (and earlier) doesn't support any of these selectors, whereas the latest versions of Chrome, Opera, Safari and Firefox all do. Before your blood boils, consider the following solutions.

Internet Explorer 9

Unless you've been living under a rock for the last week, you'll have heard that Microsoft unleashed its latest browser on an unsuspecting public. The good thing is, it's actually quite good. While I don't expect people who are reading this article to change their browsing habits, it's worth remembering that the majority of the world uses IE; and thanks to Windows Update and a global marketing campaign, we can hope to see IE9 as the dominant Windows browser in the near future. That's good for Web designers, and it's good for pseudo-selectors. But what about IE8 and its ancestors?



Internet Explorer 9 is here.

JavaScript

Our old friend JavaScript comes to the rescue. I particularly like [Selectivizr](#) by Keith Clark. Keith has put together a lovely script that, in combination with your JavaScript library of choice, adds CSS3 pseudo-class selector functionality for earlier versions of IE. Be warned that some libraries fare better than others: if you're using MooTools with Selectivizr, then all the pseudo-classes will be available, but if you're relying on jQuery to do the heavy lifting, then a number of the selectors won't work at all.

:select[ivizr]

CSS3 selectors for IE

selectivizr is a JavaScript utility that emulates CSS3 pseudo-classes and attribute selectors in Internet Explorer 6-8. Simply include the script in your pages and selectivizr will do the rest.

DOWNLOAD
v1.0.1 - (4k .ZIP archive)



Enhancing IE's selector engine

Selectivizr adds support for 19 CSS3 pseudo-classes, 2 pseudo-elements and



JavaScript-knowledge: none

Selectivizr works automatically so you don't need any JavaScript knowledge to



Works with existing tools

Selectivizr requires a JavaScript library to work. If your website already uses one

Selectivizr.

Keith recently released a [jQuery plug-in](#) that extends jQuery to include support for the following CSS3 pseudo-class selectors:

- :first-of-type
- :last-of-type
- :only-of-type
- :nth-of-type
- :nth-last-of-type

It's also worth looking at the ubiquitous *ie7.js* script (and its successors) by [Dean Edwards](#). This script solves a number of IE-related problems, including CSS3 pseudo-selectors.

So, Should We Start Using CSS3 Pseudo-Selectors Today?

I guess the answer to that question depends on how you view JavaScript. It's true that pseudo-selectors can be completely replaced with classes and IDs; but it's also true that, when styling complex layouts, pseudo-selectors are both incredibly useful and the natural next step for your CSS. If you find that they improve the readability of your CSS and reduce the need for (non-semantic) classes in your HTML, then it I'd definitely recommend embracing them today.

You could use two selectors and fall back on a class name, but that would just duplicate work. It also means that you wouldn't need the pseudo-classes in the first place. But if you did choose to go down this path, the code might look something like this:

```
1 | li:nth-of-type(3),  
2 | li.third { ... }
```

This method is not as flexible as using pseudo-classes because you have to keep updating the HTML and CSS when the page content changes.

If a lot of your users don't have JavaScript enabled, that puts you in a bit of a bind. Many Web designers argue that functionality (i.e. JavaScript) is different from layout (i.e. CSS), and so you should not rely on JavaScript to make pseudo-selectors work in IE8 and earlier.

While I agree with the principle, in practice I believe that providing the best possible experience to 99% of your users is better than accounting for the remaining 1% (or however big your non-JavaScript base may be).

Follow your website's analytics, and be prepared to make decisions that improve your skills as a Web designer and, more importantly, provide the best experience possible to the majority of users.

Final Thoughts

It's hard not to be depressed by IE8's complete lack of support for pseudo-classes. Arguably, having the browser calculate and recalculate page styles in this fashion will have implications for rendering speed; but because all other major browsers now support these selectors, it's frustrating that most of our users can't benefit from them without a JavaScript hack.

But as Professor Farnsworth says, "Good news everyone!" Breaking on the horizon is the dawn of Internet Explorer 9, and Microsoft has made sure that its new browser [supports each and every one](#) of the selectors discussed in this article.

CSS3 pseudo-selectors won't likely take up large chunks of your style sheets. They are specific yet dynamic and are more likely, at least initially, to add finishing touches to a page than to set an overall style. Perhaps you want to drop the bottom border in the last item of a list, or give visual feedback to users as they fill in a form. This is all possible with CSS3, and as usage becomes more mainstream, I expect these will become a regular part of the Web designer's toolbox.

Taming Advanced CSS Selectors

Inayaili de Leon

CSS is one of the most powerful tools that is available to Web designers (if not the most powerful). With it we can completely transform the look of a website in just a couple of minutes, and without even having to touch the markup. But despite the fact that we are all well aware of its usefulness, CSS selectors are still not used to their full potential and we sometimes have the tendency to litter our HTML with excessive and unnecessary classes and ids, divs and spans.

The best way to avoid these plagues spreading in your markup and keep it clean and semantic, is by using more complex CSS selectors, ones that can target specific elements without the need of a class or an id, and by doing that keep our code and our stylesheets flexible.

CSS Specificity

Before delving into the realms of advanced CSS selectors, it's important to understand how CSS specificity works, so that we know how to properly use our selectors and to avoid us spending hours debugging for a CSS issue that could be easily fixed if we had only paid attention to the specificity.

When we are writing our CSS we have to keep in mind that some selectors will rank higher than others in the cascade, the latest selector that we wrote will not always override the previous ones that we wrote for the same elements.

So how do you calculate the specificity of a particular selector? It's fairly straightforward if you take into account that specificity will be represented as four numbers separated by commas, like: 1, 1, 1, 1 or 0, 2, 0, 1

1. The first digit (a) is always zero, unless there is a style attribute applied to that element within the markup itself
2. The second digit (b) is the sum of the number of IDs in that selector
3. The third digit (c) is the sum of other attribute selectors and pseudo-classes in that selector. Classes (`.example`) and attribute selectors (eg. `li[id=red]`) are included here.
4. The fourth digit (d) counts the elements (like `table`, `p`, `div`, etc.) and pseudo-elements (like `:first-line`)
5. The universal selector (`*`) has a specificity of zero
6. If two selectors have the same specificity, the one that comes last on the stylesheet will be applied

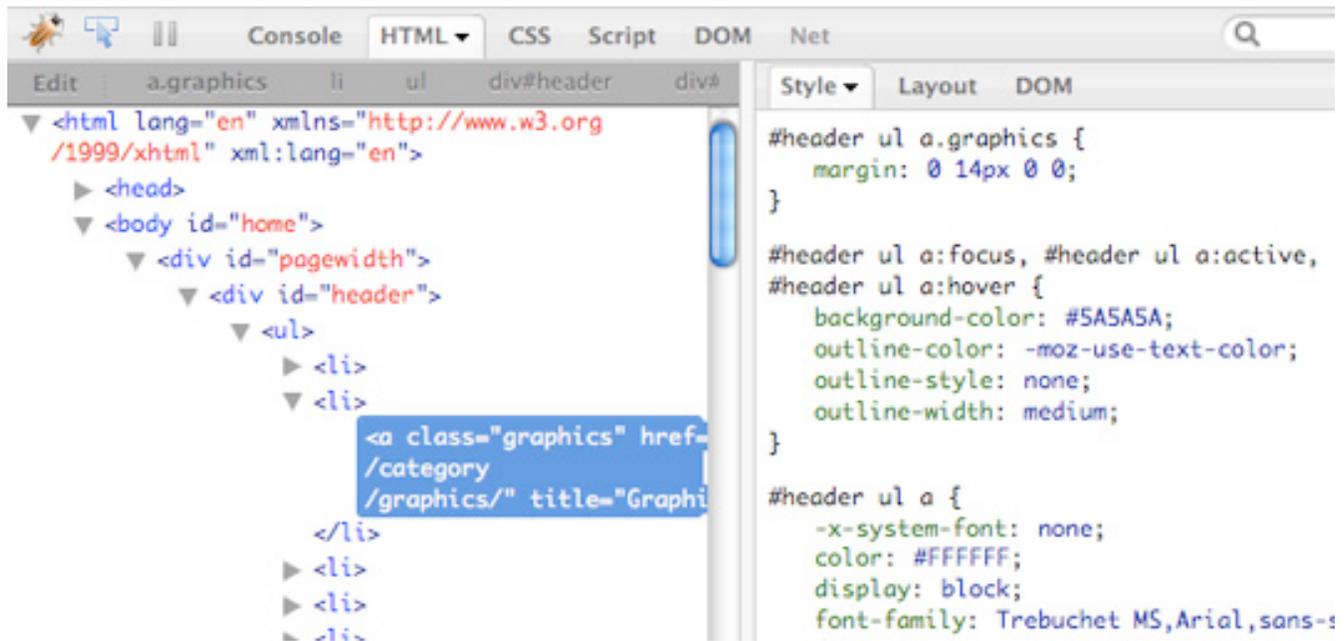
Let's take a look at a few examples, to make it easier to understand:

- `#sidebar h2` — 0, 1, 0, 1
- `h2.title` — 0, 0, 1, 1
- `h2 + p` — 0, 0, 0, 2
- `#sidebar p:first-line` — 0, 1, 0, 2

From the following selectors, the first one is the one who will be applied to the element, because it has the higher specificity:

- #sidebar p#first { color: red; } — 0, 2, 0, 1
- #sidebar p:first-line { color: blue; } — 0, 1, 0, 2

It's important to have at least a basic understanding of how specificity works, but tools like Firebug are useful to let us know which selector is being applied to a particular element by listing all the CSS selectors in order of their specificity when you are inspecting an element.



Firebug lets you easily see which selector is being applied to an element.

1. Attribute selectors

Attribute selectors let you target an element based on its attributes. You can specify the element's attribute only, so all the elements that *have* that attribute — whatever the value — within the HTML will be targeted, or be more specific and target elements that have particular values on their attributes — and this is where attribute selectors show their power.

There are 6 different types of attribute selectors:

- `[att=value]`
The attribute has to have the exact value specified.
- `[att~=value]`
The attribute's value needs to be a whitespace separated list of words (for example, `class="title featured home"`), and one of the words is exactly the specified value.
- `[att|=value]`
The attribute's value is exactly "value" or starts with the word "value" and is immediately followed by "-", so it would be "value-".
- `[att^=value]`
The attribute's value starts with the specified value.
- `[att$=value]`
The attribute's value ends with the specified value.
- `[att*=value]`
The attribute's value contains the specified value.

For example, if you want to change the background color of all the `div` elements that are posts on your blog, you can use the an attribute selector that targets every `div` whose `class` attribute starts with "post-":

```
1 | div[class*="post"] {  
2 |   background-color: #333;  
3 | }
```

This will match all the `div` elements whose `class` attribute contains the words "posts", in any position.

Another useful usage of attribute selectors is to target different types of `input` elements. For example, if you want your text inputs to have a different width from the others, you can use a simple attribute selector:

```
1 input[type="text"] {  
2   width: 200px;  
3 }
```

This will target all the `input` elements whose `type` attribute is exactly "text".

Now let's say you want to add a different icon next to each different type of file your website is linking to, so your website's visitors know when they'll get an image, a PDF file, a Word document, etc. This can be done by using an attribute selector:

```
1 a[href$=".jpg"] {  
2   background: url(jpeg.gif) no-repeat left 50%;  
3   padding: 2px 0 2px 20px;  
4 }  
5  
6 a[href$=".pdf"] {  
7   background: url(pdf.gif) no-repeat left 50%;  
8   padding: 2px 0 2px 20px;  
9 }  
10  
11 a[href$=".doc"] {  
12   background: url(word.gif) no-repeat left 50%;  
13   padding: 2px 0 2px 20px;  
14 }
```

In this example, we've used an attribute selector that will target all the links (a) whose `href` attribute ends (\$) with `.jpg`, `.pdf` or `.doc`.

Notes on browser support

Apart from Internet Explorer 6, all major browsers support attribute selectors. This means that when you are using attribute selectors on your stylesheets, you should make sure that IE6 users will still be provided with a usable site. Take our third example: adding an icon to your links adds another level of usability to your site, but the site will still be usable if the links don't show any icons.

2. Child selector

The child selector is represented by the sign `>`. It allows you to target elements that are direct children of a particular element.

For example, if you want to match all the `h2` elements that are a direct child of your sidebar `div`, but not the `h2` elements that may be also within the `div`, but that are grandchildren (or later descendants) of your element, you can use this selector:

```
1 | div#sidebar > h2 {  
2 |   font-size: 20px;  
3 | }
```

You can also use both child and descendant selectors combined. For example, if you want to target only the `blockquote` elements that are within `divs` that are direct grandchildren of the `body` element (you may want to match blockquotes inside the main content `div`, but not if they are outside it):

```
1 | body > div > div blockquote {
2 |   margin-left: 30px;
3 | }
```

Notes on browser support

Like the attribute selectors, the child selector is not supported by Internet Explorer 6. If the effect you are trying to achieve by using it is crucial for the website's usability or overall aesthetics, you can consider using a class selector with it, or on a IE-only stylesheet, but that would detract from the purpose of using child selectors.

3. Sibling combinators

There are two types of sibling combinators: adjacent sibling combinators and general sibling combinators.

Adjacent sibling combinator

This selector uses the plus sign, "+", to combine two sequences of simple selectors. The elements in the selector have the same parent, and the second one must come immediately after the first.

The adjacent sibling combinator can be very useful, for example, when dealing with text. Lets say you want to add a top margin to all the h2 tags that follow a paragraph (you don't need to add a top margin if the heading comes after an h1 tag or if it's the first element on that page):

```
1 | p + h2 {
2 |   margin-top: 10px;
3 | }
```

You can be even more specific and say that you only want this rule applied if the elements are within a particular `div`:

```
1 | div.post p + h2 {  
2 |   margin-top: 10px;  
3 | }
```

Or you can add another level of complexity: say you want the first line of the paragraphs of every page to be in small caps.

```
1 | .post h1 + p:first-line {  
2 |   font-variant: small-caps;  
3 | }
```

Because you know that the first paragraph of every post immediately follows an `h1` tag, you can refer to the `h1` on your selector.

General sibling combinator

The general sibling combinator works pretty much the same as the adjacent sibling combinator, but with the difference that the second selector doesn't have to immediately follow the first one.

So if you need to target all the `p` tags that are within a particular `div` and that follow the `h1` tag (you may want those `p` tags to be larger than the ones that come before the title of your post), you can use this selector:

```
1 | .post h1 ~ p {  
2 |   font-size: 13px;  
3 | }
```

Notes on browser support

Internet Explorer 6 doesn't understand sibling combinators, but, as for the other cases, if your audience includes a small percentage of IE6 users, and if the website's layout isn't broken or severely affected by its lack of support, this is a much easier way of achieving lots of cool effects without the need of cluttering your HTML with useless classes and ids.

4. Pseudo-classes

Dynamic pseudo-classes

These are called dynamic pseudo-classes because they actually do not exist within the HTML: they are only present **when the user is or has interacted** with the website.

There are two types of dynamic pseudo-classes: **link** and **user action** ones. The link are `:link` and `:visited`, while the user action ones are `:hover`, `:active` and `:focus`.

From all the CSS selectors mentioned in this post, these will probably be the ones that are most commonly used.

The **:link** pseudo-class applies to links that haven't been visited by the user, while the **:visited** pseudo-class applies to links that have been visited, so they are mutually exclusive.

The **:hover** pseudo-class applies when the user moves the cursor over the element, without having to activate or click on it. The **:active** pseudo-class applies when the user actually clicks on the element. And finally the **:focus** pseudo-class applies when that element is on focus — the most common application is on form elements.

You can use more than one user action dynamic pseudo-class in your stylesheets, so you can have, for example, a different background color for an input field depending on whether the user's cursor is only hovering over it or hovering over it while in focus:

```
1 input:focus {
2   background: #D2D2D2;
3   border: 1px solid #5E5E5E;
4 }
5
6 input:focus:hover {
7   background: #C7C7C7;
8 }
```

Notes on browser support

The dynamic pseudo-classes are supported by all modern browsers, even IE6. But bear in mind that IE6 only allows the `:hover` pseudo-class to be applied to link elements (a) and only IE8 accepts the `:active` state on elements other than links.

:first-child

The `:first-child` pseudo-class allows you to target an element that is the first child of another element. For example, if you want to add a top margin to the first `li` element of your unordered lists, you can have this:

```
1 ul > li:first-child {
2   margin-top: 10px;
3 }
```

Let's take another example: you want all your h2 tags in your sidebar to have a top margin, to separate them from whatever comes before them, but the first one doesn't need a margin. You can use the following code:

```
1 #sidebar > h2 {
2   margin-top: 10px;
3 }
4
5 #sidebar > h2:first-child {
6   margin-top: 0;
7 }
```

Notes on browser support

IE6 doesn't support the `:first-child` pseudo-class. Depending on the design that the pseudo-class is being applied to, it may not be a major cause for concern. For example, if you are using the `:first-child` selector to remove top or bottom margins from headings or paragraphs, your layout will probably not break in IE6, it will only look slightly different. But if you are using the `:first-child` selector to remove left and right margins from, for example, a floated sequence of divs, that may cause more disruption to your designs.

The language pseudo-class

The language pseudo-class, `:lang()`, allows you to match an element based on its language.

For example, lets say you want a specific link on your site to have a different background color, depending on that page's language:

```
1 :lang(en) > a#flag {
2   background-image: url(english.gif);
3 }
4
5 :lang(fr) > a#flag {
6   background-image: url(french.gif);
7 }
```

The selectors will match that particular link if the page's language is either equal to "en" or "fr" or if it starts with "en" or "fr" and is immediately followed by an "-".

Notes on browser support

Not surprisingly, the only version of Internet Explorer that supports this selector is 8. All other major browsers support the language pseudo-selector.

5. CSS 3 Pseudo-classes

:target

When you're using links with fragment identifiers (for example, <http://www.smashingmagazine.com/2009/08/02/bauhaus-ninety-years-of-inspiration/#comments>, where "#comments" is the fragment identifier), you can style the target by using the :target pseudo-class.

For example, let's imagine you have a long page with lots of text and h2 headings, and there is an index of those headings at the top of the page. It will be much easier for the user if, when clicking on a particular link within the index, that heading would become highlighted in some way, when the page scrolls down. Easy:

```
1 h2:target {
2   background: #F2EBD6;
3 }
```

Notes on browser support

This time, Internet Explorer is really annoying and has no support at all for the `:target` pseudo-class. Another glitch is that Opera doesn't support this selector when using the back and forward buttons. Other than that, it has support from the other major browsers.

The UI element states pseudo-classes

Some HTML elements have an enable or disabled state (for example, input fields) and checked or unchecked states (radio buttons and checkboxes). These states can be targeted by the `:enabled`, `:disabled` or `:checked` pseudo-classes, respectively.

So you can say that any `input` that is disabled should have a light grey background and dotted border:

```
1 input:disabled {
2   border: 1px dotted #999;
3   background: #F2F2F2;
4 }
```

You can also say that all checkboxes that are checked should have a left margin (to be easily seen within a long list of checkboxes):

```
1 input[type="checkbox"]:checked {
2   margin-left: 15px;
3 }
```

Notes on browser support

All major browsers, except our usual suspect, Internet Explorer, support the UI element states pseudo-classes. If you consider that you are only adding an extra level of detail and improved usability to your visitors, this can still be an option.

6. CSS 3 structural pseudo-classes

:nth-child

The `:nth-child()` pseudo-class allows you to target one or more specific children of a parent element.

You can target a single child, by defining its value as an integer:

```
1 | ul li:nth-child(3) {  
2 |   color: red;  
3 | }
```

This will turn the text on the third `li` item within the `ul` element red. Bear in mind that if a different element is inside the `ul` (not a `li`), it will also be counted as its child.

You can target a parent's children using expressions. For example, the following expression will match every third `li` element starting from the fourth:

```
1 | ul li:nth-child(3n+4) {  
2 |   color: yellow;  
3 | }
```

In the previous case, the first yellow `li` element will be the fourth. If you just want to start counting from the first `li` element, you can use a simpler expression:

```
1 | ul li:nth-child(3n) {
2 |   color: yellow;
3 | }
```

In this case, the first yellow `li` element will be the third, and every other third after it. Now imagine you want to target only the first four `li` elements within the list:

```
1 | ul li:nth-child(-n+4) {
2 |   color: green;
3 | }
```

The value of `:nth-child` can also be defined as “even” or “odd”, which are the same as using “`2n`” (every second child) or “`2n+1`” (every second child starting from the first), respectively.

:nth-last-child

The `:nth-last-child` pseudo-class works basically as the `:nth-child` pseudo-class, but it starts counting the elements from the last one.

Using one of the examples above:

```
1 | ul li:nth-child(-n+4) {
2 |   color: green;
3 | }
```

Instead of matching the first four `li` elements in the list, this selector will match the *last* four elements.

You can also use the values “even” or “odd”, with the difference that in this case they will count the children starting from the last one:

```
1 | ul li:nth-last-child(odd) {  
2 |   color: grey;  
3 | }
```

:nth-of-type

The `:nth-of-type` pseudo-class works just like the `:nth-child`, with the difference that it only counts children that match the element in the selector.

This can be very useful if we want to target elements that may contain different elements within them. For example, let’s imagine we want to turn every second paragraph in a block of text blue, but we want to ignore other elements such as images or quotations:

```
1 | p:nth-of-type(even) {  
2 |   color: blue;  
3 | }
```

You can use the same values as you would use for the `:nth-child` pseudo-class.

:nth-last-of-type

You guessed it! The `:nth-last-of-type` pseudo-class can be used exactly like the aforementioned `:nth-last-child`, but this time, it will only target the elements that match our selector:

```
1 | ul li:nth-last-of-type(-n+4) {
2 |   color: green;
3 | }
```

We can be even more clever, and combine more than one of these pseudo-classes together on a massive selector. Let's say all images within a post `div` to be floated left, except for the first and last one (let's imagine these would full width, so they shouldn't be floated):

```
1 | .post img:nth-of-type(n+2):nth-last-of-type(n+2) {
2 |   float: left;
3 | }
```

So in the first part of this selector, we are targeting every image starting from the second one. In the second part, we are targeting every image except for the last one. Because the selectors aren't mutually exclusive, we can use them both on one selector thus excluding both the first and last element at once!

:last-child

The `:last-child` pseudo-class works just as the `:first-child` pseudo-class, but instead targets the last child of a parent element.

Let's imagine you don't want the last paragraph within your post `div` to have a bottom margin:

```
1 | .post > p:last-child {
2 |   margin-bottom: 0;
3 | }
```

This selector will target the last paragraph that is a direct *and* the last child of an element with the class of “post”.

:first-of-type and :last-of-type

The `:first-of-type` pseudo-class is used to target an element that is the first of its type within its parent.

For example, you can target the first paragraph that is a direct child of a particular `div`, and capitalize its first line:

```
1 | .post > p:first-of-type:first-line {  
2 |   font-variant: small-caps;  
3 | }
```

With this selector you make sure that you are targeting only paragraphs that are direct children of the “post” `div`, and that are the first to match our `p` element.

The `:last-of-type` pseudo-class works exactly the same, but targets the *last* child of its type instead.

:only-child

The `:only-child` pseudo-class represents an element that is the only child of its parent.

Let’s say you have several boxes (“news”) with paragraphs of text inside them. When you have more than one paragraph, you want the text to be smaller than when you have only one:

```
1 div.news > p {
2   font-size: 1.2em;
3 }
4
5 div.news > p:only-child {
6   font-size: 1.5em;
7 }
```

In the first selector, we are defining the overall size of the `p` elements that are direct children of a “news” `div`. On the second one, we are overriding the previous font-size by saying, if the `p` element is the only child of the “news” `div`, its font size should be bigger.

:only-of-type

The `:only-of-type` pseudo-class represents an element that is the only child of its parent with the same element.

How can this be useful? Imagine you have a sequence of posts, each one represented by a `div` with the class of “post”. Some of them have more than one image, but others have only one image. You want the image within the later ones to be aligned to the center, while the images on posts with more than one image to be floated. That would be quite easy to accomplish with this selector:

```
1 .post > img {
2   float: left;
3 }
4
5 .post > img:only-of-type {
6   float: none;
7   margin: auto;
8 }
```

:empty

The `:empty` pseudo-class represents an element that has no content within it.

It can be useful in a number of ways. For example, if you have multiple boxes in your “sidebar” `div`, but don’t want the empty ones to appear on the page:

```
1 #sidebar .box:empty {
2   display: none;
3 }
```

Beware that even if there is a single space in the “box” `div`, it will not be treated as empty by the CSS, and therefore will not match the selector.

Notes on browser support

Internet Explorer (up until version 8) has no support for structural pseudo-classes. Firefox, Safari and Opera support these pseudo-classes on their latest releases. This means that if what is being accomplished with these selectors is fundamental for the website’s usability and accessibility, or if the larger part of the website’s audience is using IE and you don’t want to deprive them of some design details, it would be wise to keep using regular

classes and simpler selectors to cater for those browsers. If not, you can just go crazy!

7. The negation pseudo-class

The negation pseudo-class, `:not()`, lets you target elements that do not match the selector that is represented by its argument.

For example, this can be useful if you need to style all the `input` elements within a form, but you don't want your input elements with the type `submit` to be styled — you want them to be styled in a different way —, to look more like buttons:

```
1 input:not([type="submit"]) {  
2   width: 200px;  
3   padding: 3px;  
4   border: 1px solid #000000;  
5 }
```

Another example: you want all the paragraphs within your `post div` to have a larger font-size, except for the one that indicates the time and date:

```
1 .post p:not(.date) {  
2   font-size: 13px;  
3 }
```

Can you image the number of possibilities this selector brings with it, and the amount of useless selectors you could strip out off your CSS files were it widely supported?

Notes on browser support

Internet Explorer is our usual party pooper here: no support at all, not even on IE8. This probably means that this selector will still have to wait a while before some developers lose the fear of adding them to their stylesheets.

8. Pseudo-elements

Pseudo-elements allow you to access elements that don't actually exist in the HTML, like the first line of a text block or its first letter.

Pseudo-elements exist in CSS 2.1, but the CSS 3 specifications state that they should be used with the double colon "::", to distinguish them from pseudo-classes. In CSS 2.1, they are used with only one colon, ":". Browsers should be able to accept both formats, except in the case of pseudo-elements that may be introduced only in CSS 3.

::first-line

The `::first-line` pseudo-element will match the first line of a block, inline-block, table-caption or table-cell level element.

This is particularly useful to add subtle typographical details to your text blocks, like, for example, transforming the first line of an article into small caps:

```
1 | h1 + p::first-line {  
2 |   font-variant: small-caps;  
3 | }
```

If you've been paying attention, you'll know that this means the paragraph that comes *immediately after* an `h1` tag ("`+`") should have its first line in small caps.

You could also refer to the first line of a particular `div`, without having to refer to the actual paragraph tag:

```
1 |div.post p::first-line { font-variant: small-caps; }
```

Or go one step farther and target specifically the *first* paragraph within a particular `div`:

```
1 |div.post > p:first-child::first-line {  
2 |   font-variant: small-caps;  
3 | }
```

Here, the "`>`" symbol indicates that you are targeting a direct child the `post` `div`, so if the paragraph were to be inside a second `div`, it wouldn't match this selector.

::first-letter

The `::first-letter` pseudo-element will match the first letter of a block, unless it's preceded by some other content, like an image, on the same line.

Like the `::first-line` pseudo-element, `::first-letter` is commonly used to add typographical details to text elements, like drop caps or initials.

Here is how you could use the `::first-letter` pseudo-element to create a drop cap:

```
1 p {  
2   font-size: 12px;  
3 }  
4  
5 p::first-letter {  
6   font-size: 24px;  
7   float: left;  
8 }
```

Bear in mind that if you use both `::first-line` and `::first-letter` in the same element, the `::first-letter` properties will override the same properties inherited from `::first-line`.

This element can sometimes produce unexpected results, if you're not aware of the W3C specs: it's actually the CSS selector with the longest spec! So it's a good idea to read them carefully if you're planning on using it (as it is for all the other selectors).

::before and ::after

The `::before` and `::after` pseudo-elements are used to insert content before or after an element's content, purely via CSS.

These elements will inherit many of the properties of the elements that they are being attached to.

Imagine you want to add the words "Graphic number x:" before the descriptions of graphs and charts on your page. You could achieve this without having to write the words "Graphic number", or the number itself yourself:

```
1 .post {
2   counter-reset: image;
3 }
4
5 p.description::before {
6   content: "Figure number " counter(image) ": ";
7   counter-increment: image;
8 }
```

What just happened here?

First, we tell the HTML to create the “image” counter. We could have added this property to the body of the page, for example. Also, we can call this counter whatever name we want to, as long as we always reference it by the same name: try it for yourself!

Then we say that we want to add, before every paragraph with the class “description”, this piece of content: “Figure number ” — notice that only what we wrote between quotes will be created on the page, so we need to add the spaces as well!

After that, we have `counter(image)`: this will pick up the property we’ve already defined in the `.post` selector. It will by default start with the number one (1).

The next property is there so that the counter knows that for each `p.description`, it needs to increment the image counter by 1 (`counter-increment: image`).

It’s not as complicated as it looks, and it can be quite useful.

The `::before` and `::after` pseudo-elements are often only used with the `content` property, to add small sentences or typographical elements,

but here it's shown how we can use it in a more powerful way in conjunction with the `counter-reset` and `counter-increment` properties.

Fun fact: the `::first-line` and `::first-letter` pseudo-elements will match the content added by the `::before` pseudo-element, if present.

Notes on browser support

These pseudo-elements are supported by IE8 (not IE7 or 6), if the single colon format is used (for example, `:first-letter`, not `::first-letter`). All the other major browsers support these selectors.

Conclusion

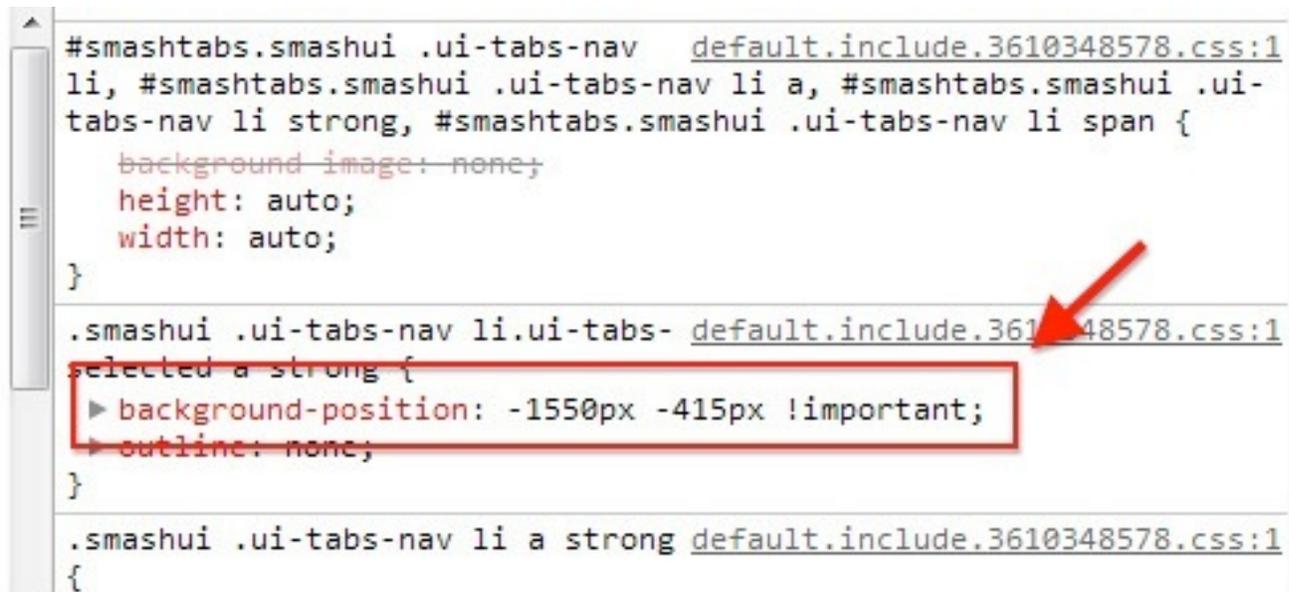
Enough with the boring talk, now it's time for *you* to grab the information from this article and go try it for yourself: start by creating an experimental page and testing all of these selectors. Come back here when in doubt and make sure to always refer to the W3C specs, but don't just sit there thinking that because these selectors aren't yet widely supported you might as well ignore them.

If you're a bit more adventurous, or if you're not afraid of letting go of the past filled with useless and non-semantic classes and ids, why not sneak one or two of these powerful CSS selectors into your next project? We promise you'll never look back.

!important CSS Declarations: How and When to Use Them

Louis Lazaris

When the [CSS1 specification](#) was drafted in the mid to late 90s, it introduced `!important` declarations that would help developers and users easily override normal specificity when making changes to their stylesheets. For the most part, `!important` declarations have remained the same, with only one change in CSS2.1 and nothing new added or altered in the CSS3 spec in connection with this unique declaration.



```
#smashtabs.smashui .ui-tabs-nav default.include.3610348578.css:1
li, #smashtabs.smashui .ui-tabs-nav li a, #smashtabs.smashui .ui-
tabs-nav li strong, #smashtabs.smashui .ui-tabs-nav li span {
  background-image: none;
  height: auto;
  width: auto;
}

.smashui .ui-tabs-nav li.ui-tabs- default.include.3610348578.css:1
selected a strong {
  background-position: -1550px -415px !important;
  outline: none;
}

.smashui .ui-tabs-nav li a strong default.include.3610348578.css:1
{
```

Let's take a look at what exactly these kinds of declarations are all about, and when, if ever, you should use them.

A Brief Primer on the Cascade

Before we get into `!important` declarations and exactly how they work, let's give this discussion a bit of context. In the past, Smashing Magazine has covered [CSS specificity](#) in-depth, so please take a look at that article if you want a detailed discussion on the CSS cascade and how specificity ties in.

Below is a basic outline of how any given CSS-styled document will decide how much weight to give to different styles it encounters. This is a general summary of [the cascade](#) as discussed in the spec:

- Find all declarations that apply to the element and property
- Apply the styling to the element based on importance and origin using the following order, with the first item in the list having the least weight:
 - Declarations from the user agent
 - Declarations from the user
 - Declarations from the author
 - Declarations from the author with `!important` added
 - Declarations from the user with `!important` added
- Apply styling based on specificity, with the more specific selector “winning” over more general ones
- Apply styling based on the order in which they appear in the stylesheet (i.e., in the event of a tie, last one “wins”)

With that basic outline, you can probably already see how `!important` declarations weigh in, and what role they play in the cascade. Let's look at `!important` in more detail.

Syntax and Description

An `!important` declaration provides a way for a stylesheet author to give a CSS value more weight than it naturally has. It should be noted here that the phrase “!important declaration” is a reference to an entire CSS declaration, including property and value, with `!important` added (thanks to [Brad Czerniak](#) for pointing out this discrepancy). Here is a simple code example that clearly illustrates how `!important` affects the natural way that styles are applied:

```
1 #example {
2   font-size: 14px !important;
3 }
4
5 #container #example {
6   font-size: 10px;
7 }
```

In the above code sample, the element with the id of “example” will have text sized at 14px, due to the addition of `!important`.

Without the use of `!important`, there are two reasons why the second declaration block should naturally have more weight than the first: The second block is later in the stylesheet (i.e. it's listed second). Also, the second block has more specificity (`#container` followed by `#example` instead of just `#example`). But with the inclusion of `!important`, the first `font-size` rule now has more weight.

Some things to note about `!important` declarations:

- When `!important` was first introduced [in CSS1](#), an author rule with an `!important` declaration held more weight than a user rule with an `!important` declaration; to improve accessibility, this [was reversed in CSS2](#)
- If `!important` is used on a shorthand property, this adds “importance” to all the sub-properties that the shorthand property represents
- The `!important` keyword (or statement) must be placed at the end of the line, immediately before the semicolon, otherwise it will have no effect (although a space before the semicolon won’t break it)
- If for some particular reason you have to write the same property twice in the same declaration block, then add `!important` to the end of the first one, the first one will have more weight in every browser except IE6 (this works as an IE6-only hack, but doesn’t invalidate your CSS)
- In IE6 and IE7, if you use a different word in place of `!important` (like `!hotdog`), the CSS rule will still be given extra weight, while other browsers will ignore it

When Should `!important` Be Used?

As with any technique, there are pros and cons depending on the circumstances. So when should it be used, if ever? Here’s my subjective overview of potential valid uses.

Never

`!important` declarations should not be used unless they are absolutely necessary after all other avenues have been exhausted. If you use `!important` out of laziness, to avoid proper debugging, or to rush a project to completion, then you're abusing it, and you (or those that inherit your projects) will suffer the consequences.

If you include it even sparingly in your stylesheets, you will soon find that certain parts of your stylesheet will be harder to maintain. As discussed above, CSS property importance happens naturally through the cascade and specificity. When you use `!important`, you're disrupting the natural flow of your rules, giving more weight to rules that are undeserving of such weight.

If you never use `!important`, then that's a sign that you understand CSS and give proper forethought to your code before writing it.

That being said, the old adage "never say never" would certainly apply here. So below are some legitimate uses for `!important`.

To Aid or Test Accessibility

As mentioned, user stylesheets can include `!important` declarations, allowing users with special needs to give weight to specific CSS rules that will aid their ability to read and access content.

A special needs user can add `!important` to typographic properties like `font-size` to make text larger, or to color-related rules in order to increase the contrast of Web pages.

In the screen grab below, Smashing Magazine's home page is shown with a user-defined stylesheet overriding the normal text size, which can be done using Firefox's Developer Toolbar:



In this case, the text size was adjustable without using `!important`, because a user-defined stylesheet will override an author stylesheet regardless of specificity. If, however, the text size for body copy was set in the author stylesheet using an `!important` declaration, the user

stylesheet could not override the text-size setting, even with a more specific selector. The inclusion of `!important` resolves this problem and keeps the adjustability of text size within the user's power, even if the author has abused `!important`.

To Temporarily Fix an Urgent Problem

There will be times when something bugs out in your CSS on a live client site, and you need to apply a fix very quickly. In most cases, you should be able to use Firebug or another developer tool to track down the CSS code that needs to be fixed. But if the problem is occurring on IE6 or another browser that doesn't have access to debugging tools, you may need to do a quick fix using `!important`.

After you move the temporary fix to production (thus making the client happy), you can work on fixing the issue locally using a more maintainable method that doesn't muck up the cascade. When you've figured out a better solution, you can add it to the project and remove `!important` — and the client will be none the wiser.

To Override Styles Within Firebug or Another Developer Tool

Inspecting an element in Firebug or Chrome's developer tools allows you to edit styles on the fly, to test things out, debug, and so on — without affecting the real stylesheet. Take a look at the screen grab below, showing some of Smashing Magazine's styles in Chrome's developer tools:

```
}  
  
#smashtabs.smashui .ui-tabs-nav default.include.3610348578.css:1  
li, #smashtabs.smashui .ui-tabs-nav li a, #smashtabs.smashui .ui-  
tabs-nav li strong, #smashtabs.smashui .ui-tabs-nav li span {  
  background-image: none;  
  height: auto;  
  width: auto;  
}  
  
.smashui .ui-tabs-nav li.ui-tabs- default.include.3610348578.css:1  
selected a strong {  
  background-position: 1550px 415px;  
  outline: none;  
}  
  
.smashui .ui-tabs-nav li a strong default.include.3610348578.css:1  
{
```

The highlighted background style rule has a line through it, indicating that this rule has been overridden by a later rule. In order to reapply this rule, you could find the later rule and disable it. You could alternatively edit the selector to make it more specific, but this would give the entire declaration block more specificity, which might not be desired.

`!important` could be added to a single line to give weight back to the overridden rule, thus allowing you to test or debug a CSS issue without making major changes to your actual stylesheet until you resolve the issue.

Here's the same style rule with `!important` added. You'll notice the line-through is now gone, because this rule now has more weight than the rule that was previously overriding it:

```
#smashtabs.smashui .ui-tabs-nav default.include.3610348578.css:1
li, #smashtabs.smashui .ui-tabs-nav li a, #smashtabs.smashui .ui-
tabs-nav li strong, #smashtabs.smashui .ui-tabs-nav li span {
  background-image: none;
  height: auto;
  width: auto;
}

.smashui .ui-tabs-nav li.ui-tabs- default.include.3610348578.css:1
selected a strong {
  background-position: -1550px -415px !important;
  outline: none;
}

.smashui .ui-tabs-nav li a strong default.include.3610348578.css:1
{
```

To Override Inline Styles in User-Generated Content

One frustrating aspect of CSS development is when user-generated content includes inline styles, as would occur with some WYSIWYG editors in CMSs. In the CSS cascade, inline styles will override regular styles, so any undesirable element styling that occurs through generated content will be difficult, if not impossible, to change using customary CSS rules. You can circumvent this problem using an `!important` declaration, because a CSS rule with `!important` in an author stylesheet will override inline CSS.

For Print Stylesheets

Although this wouldn't be necessary in all cases, and might be discouraged in some cases for the same reasons mentioned earlier, you could add `!important` declarations to your print-only stylesheets to help override specific styles without having to repeat selector specificity.

For Uniquely-Designed Blog Posts

If you've dabbled in [uniquely-designed blog posts](#) (many designers [take issue](#) with using "art direction" for this technique, and rightly so), as showcased on [Heart Directed](#), you'll know that such an undertaking requires each separately-designed article to have its own stylesheet, or else you need to use inline styles. You can give an individual page its own styles using the code presented [in this post](#) on the Digging Into WordPress blog.

The use of `!important` could come in handy in such an instance, allowing you to easily override the default styles in order to create a unique experience for a single blog post or page on your site, without having to worry about natural CSS specificity.

Conclusion

`!important` declarations are best reserved for special needs and users who want to make Web content more accessible by easily overriding default user agent or author stylesheets. So you should do your best to give your CSS proper forethought and avoid using `!important` wherever possible. Even in many of the uses described above, the inclusion of `!important` is not always necessary.

Nonetheless, `!important` is valid CSS. You might inherit a project wherein the previous developers used it, or you might have to patch something up quickly — so it could come in handy. It's certainly beneficial to understand it better and be prepared to use it should the need arise.

Do you ever use `!important` in your stylesheets? When do you do so? Are there any other circumstances you can think of that would require its use?

An Introduction to CSS3 Keyframe Animations

Louis Lazaris

By now you've probably heard at least something about animation in CSS3 using keyframe-based syntax. The [CSS3 animations module](#) in the specification has been around for a couple of years now, and it has the potential to become a big part of Web design.

Using CSS3 keyframe animations, developers can create smooth, maintainable animations that perform relatively well and that don't require reams of scripting. It's just another way that CSS3 is helping to solve a real-world problem in an elegant manner. If you haven't yet started learning the syntax for CSS3 animations, here's your chance to prepare for when this part of the CSS3 spec moves past the [working draft](#) stage.

In this article, we'll cover all the important parts of the syntax, and we'll fill you in on browser support so that you'll know when to start using it.

A Simple Animated Landscape Scene

For the purpose of this article, I've created a simple animated landscape scene to introduce the various aspects of the syntax. You can [view the demo page](#) to get an idea of what I'll be describing. The page includes a sidebar that displays the CSS code used for the various elements (sun, moon, sky, ground and cloud). Have a quick look, and then follow along as I describe the different parts of the CSS3 animations module.

(NOTE: Safari has a bug that prevents the animation from finishing correctly. This bug seems to be fixed in Safari using a WebKit nightly build, so future versions of Safari should look the same as Chrome. See more under the heading “The Animation’s Fill Mode”)

I’ll describe the CSS related to only one of the elements: the animated sun. That should suffice to give you a good understanding of keyframe-based animations. For the other elements in the demo, you can examine the code on the demo page using the tabs.

The Keyframe @ Rule

The first unusual thing you’ll notice about any CSS3 animation code is the `keyframes @` rule. According to the spec, this specialized CSS @ rule is followed by an identifier (chosen by the developer) that is referred to in another part of the CSS.

The @ rule and its identifier are then followed by a number of rule sets (i.e. style rules with declaration blocks, as in normal CSS code). This chunk of rule sets is delimited by curly braces, which nest the rule sets inside the @ rule, much as you would find with [other @ rules](#).

Here’s the @ rule we’ll be using:

```
1 | @-webkit-keyframes sunrise {  
2 |   /* rule sets go here ... */  
3 | }
```

The word `sunrise` is an identifier of our choosing that we’ll use to refer to this animation.

Notice that I'm using the `-webkit-` prefix for all of the code examples here and in the demo. I'll discuss browser support at the end of this article, but for now it's enough to know that the only stable browsers that support these types of animations are WebKit-based ones.

The Keyframe Selectors

Let's add some rule sets inside the `@` rule:

```
1 @-webkit-keyframes sunrise {
2   0% {
3     bottom: 0;
4     left: 340px;
5     background: #f00;
6   }
7
8   33% {
9     bottom: 340px;
10    left: 340px;
11    background: #ffd630;
12  }
13
14   66% {
15     bottom: 340px;
16     left: 40px;
17     background: #ffd630;
18   }
19
20   100% {
21     bottom: 0;
```

```
22 |   left: 40px;  
23 |   background: #f00;  
24 | }  
25 | }
```

With the addition of those new rule sets, we've introduced the keyframe selector. In the code example above, the keyframe selectors are 0%, 33%, 66%, and 100%. The 0% and 100% selectors could be replaced by the keywords "from" and "to," respectively, and you would get the same results.

Each of the four rule sets in this example represents a different snapshot of the animated element, with the styles defining the element's appearance at that point in the animation. The points that are not defined (for example, from 34% to 65%) comprise the transitional period between the defined styles.

Although the spec is still in development, some rules have been defined that user agents should follow. For example, the order of the keyframes doesn't really matter. The keyframes will play in the order specified by the percentage values, and not necessarily the order in which they appear. Thus, if you place the "to" keyframe before the "from" keyframe, the animation would still play the same way. Also, if a "to" or "from" (or its percentage-based equivalent) is not declared, the browser will automatically construct it. So, the rule sets inside the @ rule are not governed by the cascade that you find in customary CSS rule sets.

The Keyframes That Animate the Sun

For the purpose of animating the sun in this demo, I've set four keyframes. As mentioned, the code above includes comments that describe the changes.

In the first keyframe, the sun is red (as if it were just rising or setting), and it is positioned below ground (i.e. not visible). Naturally, the element itself must be positioned relatively or absolutely in order for the `left` and `bottom` values to have any effect. I've also used [z-index](#) to stack the elements (to make sure, for example, that the ground is above the sun). Take note that the only styles shown in the keyframes are the styles that are animated. The other styles (such as `z-index` and `position`, which aren't animated) are declared elsewhere in the style sheet and thus aren't shown here.

```
1 0% {
2   bottom: 0; /* sun at bottom */
3   left: 340px; /* sun at right */
4   background: #f00; /* sun is red */
5 }
```

About one third of the way into the animation (33%), the sun is on the same horizontal plane but has risen and changed to a yellow-orange (to represent full daylight):

```
1 33% {
2   bottom: 340px; /* sun rises */
3   left: 340px;
4   background: #ffd630; /* changes color */
5 }
```

Then, at about two thirds into the animation (66%), the sun has moved to the left about 300 pixels but stays on the same vertical plane. Notice something else in the 66% keyframe: I've repeated the same color from the 33% keyframe, to keep the sun from changing back to red too early.

```
1 66% {
2   bottom: 340px;
3   left: 40px; /* sun moves left across sky */
4   background: #ffd630; /* maintains its color */
5 }
```

Finally, the sun gradually animates to its final state (the full red) as it disappears below the ground.

```
1 100% {
2   bottom: 0; /* sun sets */
3   left: 40px;
4   background: #f00; /* back to red */
5 }
```

Associating The Animation Name With An Element

Here's the next chunk of code we'll add in our example. It associates the animation name (in this case, the word `sunrise`) with a specific element in our HTML:

```
1 #sun.animate {
2   -webkit-animation-name: sunrise;
3 }
```

Here we're introducing the `animation-name` property. The value of this property must match an identifier in an existing `@keyframes` rule, otherwise no animation will occur. In some circumstances, you can use JavaScript to set its value to `none` (the only keyword that has been reserved for this property) to prevent an animation from occurring.

The object we've targeted is an element with an id of `sun` and a class of `animate`. The reason I've doubled up the id and class like this is so that I can use scripting to add the class name `animate`. In the demo, I've started the page statically; then, with the click of a button, all of the elements with a particular class name will have another class appended called `animate`. This will trigger all of the animations at the same time and will allow the animation to be controlled by the user.

Of course, that's just one way to do it. As is the case with anything in CSS or JavaScript, there are other ways to accomplish the same thing.

The Animation's Duration And Timing Function

Let's add two more lines to our CSS:

```
1 #sun.animate {  
2   -webkit-animation-name: sunrise;  
3   -webkit-animation-duration: 10s;  
4   -webkit-animation-timing-function: ease;  
5 }
```

You can specify the duration of the animation using the `animation-duration` property. The duration represents the time taken to complete a single iteration of the animation. You can express this value in seconds (for example, `4s`), milliseconds (`2000ms`), and seconds in decimal notation (`3.3s`).

The specification doesn't seem to specify all of the available units of time measurement. However, it seems unlikely that anyone would need anything longer than seconds; and even then, you could express duration in minutes,

hours or days simply by calculating those units into seconds or milliseconds.

The `animation-timing-function` property, when declared for the entire animation, allows you to define how an animation progresses over a single iteration of the animation. The values for `animation-timing-function` are `ease`, `linear`, `ease-out`, `step-start` and many more, [as outlined in the spec](#).

For our example, I've chosen `ease`, which is the default. So in this case, we can leave out the property and the animation will look the same.

Additionally, you can apply a specific timing function to each keyframe:

```
1 @-webkit-keyframes sunrise {
2   0% {
3     background: #f00;
4     left: 340px;
5     bottom: 0;
6     -webkit-animation-timing-function: ease;
7   }
8
9   33% {
10    bottom: 340px;
11    left: 340px;
12    background: #ffd630;
13    -webkit-animation-timing-function: linear;
14  }
15
16  66% {
17    left: 40px;
18    bottom: 340px;
```

```
19 background: #ffd630;
20 -webkit-animation-timing-function: steps(4);
21 }
22
23 100% {
24 bottom: 0;
25 left: 40px;
26 background: #f00;
27 -webkit-animation-timing-function: linear;
28 }
29 }
```

A separate timing function defines each of the keyframes above. One of them is the `steps` value, which jerks the animation forward a predetermined number of steps. The final keyframe (100% or `to`) also has its own timing function, but because it is for the final state of a forward-playing animation, the timing function applies only if the animation is played in reverse. In our example, we won't define a specific timing function for each keyframe, but this should suffice to show that it is possible.

The Animation's Iteration Count And Direction

Let's now add two more lines to our CSS:

```
1 #sun.animate {
2   -webkit-animation-name: sunrise;
3   -webkit-animation-duration: 10s;
4   -webkit-animation-timing-function: ease;
5   -webkit-animation-iteration-count: 1;
6   -webkit-animation-direction: normal;
7 }
```

This introduces two more properties: one that tells the animation how many times to play, and an indicator that tells the animation to animate back to the start position.

The `animation-iteration-count` property is set to `1`, meaning that the animation will play only once. This property accepts an integer value or `infinite`.

In addition, the `animation-direction` property is set to `normal` (the default), which means that the animation will play in the same direction (from start to finish) each time it runs. In our example, the animation is set to run only once, so the property is unnecessary. The other value we could specify here is `alternate`, which makes the animation play in reverse on every other iteration. Naturally, for the `alternate` value to take effect, the iteration count needs to have a value of 2 or higher.

The Animation's Delay And Play State

Let's add another two lines of code:

```
1 #sun.animate {
2   -webkit-animation-name: sunrise;
3   -webkit-animation-duration: 10s;
4   -webkit-animation-timing-function: ease;
5   -webkit-animation-iteration-count: 1;
6   -webkit-animation-direction: normal;
7   -webkit-animation-delay: 5s;
8   -webkit-animation-play-state: running;
9 }
```

First, we introduce the `animation-delay` property, which does exactly what you would think: it allows you to delay the animation by a set amount of time. Interestingly, this property can have a negative value, which moves the starting point partway through the animation according to negative value.

The `animation-play-state` property, which [might be removed](#) from the spec, accepts one of two possible values: `running` and `paused`. This value has limited practical use. The default is `running`, and the value `paused` simply makes the animation start off in a paused state, until it is manually played. You can't specify a `paused` state in the CSS for an individual keyframe; the real benefit of this property becomes apparent when you use JavaScript to change it in response to user input or something else.

The Animation's Fill Mode

We'll add one more line to our code, the property to define the "fill mode":

```
1 #sun.animate {
2   -webkit-animation-name: sunrise;
3   -webkit-animation-duration: 10s;
4   -webkit-animation-timing-function: ease;
5   -webkit-animation-iteration-count: 1;
6   -webkit-animation-direction: normal;
7   -webkit-animation-delay: 5s;
8   -webkit-animation-play-state: running;
9   -webkit-animation-fill-mode: forwards;
10 }
```

The `animation-fill-mode` property allows you to define the styles of the animated element before and/or after the animation executes. A value of `backwards` causes the styles in the first keyframe to be applied before the animation runs. A value of `forwards` causes the styles in the last keyframe to be applied after the animation runs. A value of `both` does both.

UPDATE: It seems that the `animation-fill-mode` property has been removed from the spec, or else was never there to begin with, so this property may not end up in the spec. Also, Chrome and Safari respond differently when it is used. Safari will only apply a value of “forwards” if there are exactly two keyframes defined. It always seems to use the 2nd keyframe as the “forwards” state, which is not how Chrome does it; Chrome uses the final keyframe, which seems to be correct behaviour. Additionally, I’ve confirmed that the most up to date WebKit nightly does not have this bug, so future versions of Safari will render this correctly.

Shorthand

Finally, the specification describes [shorthand notation](#) for animations, which combines six of the properties described above. This includes everything except `animation-play-state` and `animation-fill-mode`.

Some Notes On The Demo Page And Browser Support

As mentioned, the code in this article is for animating only a single element in the demo: the sun. To see the full code, visit the [demo page](#). You can view all of the source together or use the tabs in the sidebar to view the code for individual elements in the animation.

The demo does not use any images, and the animation does not rely on JavaScript. The sun, moon and cloud are all created using CSS3's `border-radius`, and the only scripting on the page is for the tabs on the right and for the button that lets users start and reset the animation.

If you view the page in anything but a WebKit browser, it won't work. Firefox does not currently support keyframe-based animation, but [support is expected for Firefox 5](#). So, to make the source code as future-proof as possible, I've included all of the `-moz` prefixes along with the standard syntax.

Here are the browsers that support CSS3 keyframe animations:

- Chrome 2+
- Safari 4+
- Firefox 5+
- iOS Safari 3.2+
- Android 2.1+

Although no official announcement has been made, support in Opera is expected. There's no word yet on support in IE.

CSS Specificity and Inheritance

Inayaili de Leon

CSS's barrier to entry is extremely low, mainly due to the nature of its syntax. Being clear and easy to understand, the syntax makes sense even to the inexperienced Web designer. It's so simple, in fact, that you could style a simple CSS-based website within a few hours of learning it.

But this apparent simplicity is deceitful. If after a few hours of work, your perfectly crafted website looks great in Safari, all hell might break loose if you haven't taken the necessary measures to make it work in Internet Explorer. In a panic, you add hacks and filters where only a few tweaks or a different approach might do. Knowing how to deal with these issues comes with experience, with trial and error and with failing massively and then learning the correct way.

Understanding a few often overlooked concepts is also important. The concepts may be hard to grasp and look boring at first, but understanding them and knowing how to take advantage of them is important.

Two of these concepts are specificity and inheritance. Not very common words among Web designers, are they? Talking about `border-radius` and `text-shadow` is a lot more fun; but specificity and inheritance are fundamental concepts that any person who wants to be good at CSS should understand. They will help you create clean, maintainable and flexible style sheets. Let's look at what they mean and how they work.

The notion of a "cascade" is at the heart of CSS (just look at its name). It ultimately determines which properties will modify a given element. The cascade is tied to three main concepts: importance, specificity and source

order. The cascade follows these three steps to determine which properties to assign to an element. By the end of this process, the cascade has assigned a weight to each rule, and this weight determines which rule takes precedence, when more than one applies.

1. Importance

Style sheets can have a few different sources:

1. **User agent**

For example, the browser's default style sheet.

2. **User**

Such as the user's browser options.

3. **Author**

This is the CSS provided by the page (whether inline, embedded or external)

By default, this is the order in which the different sources are processed, so the author's rules will override those of the user and user agent, and so on.

There is also the `!important` declaration to consider in the cascade. This declaration is used to balance the relative priority of user and author style sheets. While author style sheets take precedence over user ones, if a user rule has `!important` applied to it, it will override even an author rule that also has `!important` applied to it.

Knowing this, let's look at the final order, in ascending order of importance:

-
1. User agent declarations
 2. User declarations
 3. Author declarations
 4. Author `!important` declarations
 5. User `!important` declarations

This flexibility in priority is key because it allows users to override styles that could hamper the accessibility of a website. (A user might want a larger font or a different color, for example.)

2. Specificity

Every CSS rule has a particular weight (as mentioned in the introduction), meaning it could be more or less important than the others or equally important. This weight defines which properties will be applied to an element when there are conflicting rules.

Upon assessing a rule's importance, the cascade attributes a specificity to it; if one rule is more specific than another, it overrides it.

If two rules share the same weight, source and specificity, the later one is applied.

2.1 How to Calculate Specificity?

There are several ways to calculate a selector's specificity.

The quickest way is to do the following. Add 1 for each element and pseudo-element (for example, `:before` and `:after`); add 10 for each

attribute (for example, [type="text"]), class and pseudo-class (for example, :link or :hover); add 100 for each ID; and add 1000 for an inline style.

Let's calculate the specificity of the following selectors using this method:

- **p.note**
1 class + 1 element = 11
- **#sidebar p[lang="en"]**
1 ID + 1 attribute + 1 element = 111
- **body #main .post ul li:last-child**
1 ID + 1 class + 1 pseudo-class + 3 elements = 123

A similar method, described in the W3C's specifications, is to start with a=0, b=0, c=0 and d=0 and replace the numbers accordingly:

- a = 1 if the style is inline
- b = the number of IDs
- c = the number of attribute selectors, classes and pseudo-classes
- d = the number of element names and pseudo-elements

Let's calculate the specificity of another set of selectors:

- **<p style="color:#000000;">**
a=1, b=0, c=0, d=0 → 1000
- **footer nav li:last-child**
a=0, b=0, c=1, d=3 → 0013

- `#sidebar input:not([type="submit"])`

a=0, b=1, c=1, d=1 → 0111

(Note that the negation pseudo-class doesn't count, but the selector inside it does.)

If you'd rather learn this in a more fun way, Andy Clarke drew a clever [analogy between specificity and Star Wars](#) back in 2005, which certainly made it easier for Star Wars fans to understand specificity. Another good explanation is "[CSS Specificity for Poker Players](#)," though slightly more complicated.



Andy Clarke's CSS Specificity Wars chart.

Remember that non-CSS presentational markup is attributed with a specificity of 0, which would apply, for example, to the `font` tag.

Getting back to the `!important` declaration, keep in mind that using it on a shorthand property is the same as declaring all of its sub-properties as `!important` (even if that would revert them to the default values).

If you are using imported style sheets (`@import`) in your CSS, you have to declare them before all other rules. Thus, they would be considered as coming before all the other rules in the CSS file.

Finally, if two selectors turn out to have the same specificity, the last one will override the previous one(s).

2.2 Making Specificity Work For You

If not carefully considered, specificity can come back to haunt you and lead you to unwittingly transform your style sheets into a complex hierarchy of unnecessarily complicated rules.

You can follow a few guidelines to avoid major issues:

- When starting work on the CSS, use generic selectors, and add specificity as you go along
- Using advanced selectors doesn't mean using unnecessarily complicated ones
- Rely more on specificity than on the order of selectors, so that your style sheets are easier to edit and maintain (especially by others)

A good rule of thumb can be found in Jim Jeffers' article, "[The Art and Zen of Writing CSS](#)":

Refactoring CSS selectors to be less specific is exponentially more difficult than simply adding specific rules as situations arise.

3. Inheritance

A succinct and clear explanation of inheritance is in the [CSS3 Cascading and Inheritance module](#) specifications (still in “Working draft” mode):

Inheritance is a way of propagating property values from parent elements to their children.

Some CSS properties are inherited by the children of elements by default. For example, if you set the `body` tag of a page to a specific font, that font will be inherited by other elements, such as headings and paragraphs, without you having to specifically write as much. This is the magic of inheritance at work.

The CSS specification determines whether each property is inherited by default or not. Not all properties are inherited, but you can force ones to be by using the `inherit` value.

3.1 Object-Oriented Programming Inheritance

Though beyond the scope of this article, CSS inheritance shouldn't be confused with object-oriented programming (OOP) inheritance. Here is the definition of [OOP inheritance from Wikipedia](#), and it makes clear that we are *not* talking about the same thing:

In object-oriented programming (OOP), inheritance is a way to form new classes [...] using classes that have already been defined. Inheritance is employed to help reuse existing code with little or no modification. The new classes [...] inherit attributes and behavior of the pre-existing classes.

...

3.2 How Inheritance Works

When an element inherits a value from its parent, it is inheriting its computed value. What does this mean? Every CSS property goes through a four-step process when its value is being determined. Here's an excerpt from the [W3C specification](#):

The final value of a property is the result of a four-step calculation: the value is determined through specification (the "specified value"), then resolved into a value that is used for inheritance (the "computed value"), then converted into an absolute value if necessary (the "used value"), and finally transformed according to the limitations of the local environment (the "actual value").

In other words:

1. **Specified value**

The user agent determines whether the value of the property comes from a style sheet, is inherited or should take its initial value.

2. **Computed value**

The specified value is resolved to a computed value and exists even when a property doesn't apply. The document doesn't have to be laid out for the computed value to be determined.

3. **Used value**

The used value takes the computed value and resolves any dependencies that can only be calculated after the document has been laid out (like percentages).

4. **Actual value**

This is the value used for the final rendering, after any approximations have been applied (for example, converting a decimal to an integer).

If you look at any CSS property's specification, you will see that it defines its initial (or default) value, the elements it applies to, its inheritance status and its computed value (among others). For example, the `background-color` specification states the following:

Name: background-color

Value: <color>

Initial: transparent

Applies to: all elements

Inherited: no

Percentages: N/A

Media: visual

Computed value: the computed color(s)

Confusing? It can be. So, what do we need to understand from all this? And why is it relevant to inheritance?

Let's go back to the first sentence of this section, which should make more sense now. *When an element inherits a value from its parent, it inherits its computed value.* Because the computed value exists even if it isn't specified in the style sheet, a property can be inherited even then: the initial value will be used. So, you can make use of inheritance even if the parent doesn't have a specified property.

3.3 Using Inheritance

The most important thing to know about inheritance is that it's there and how it works. If you ignore the jargon, inheritance is actually very straightforward.

Imagine you had to specify the `font-size` or `font-family` of every element, instead of simply adding it to the `body` element? That would be cumbersome, which is why inheritance is so helpful.

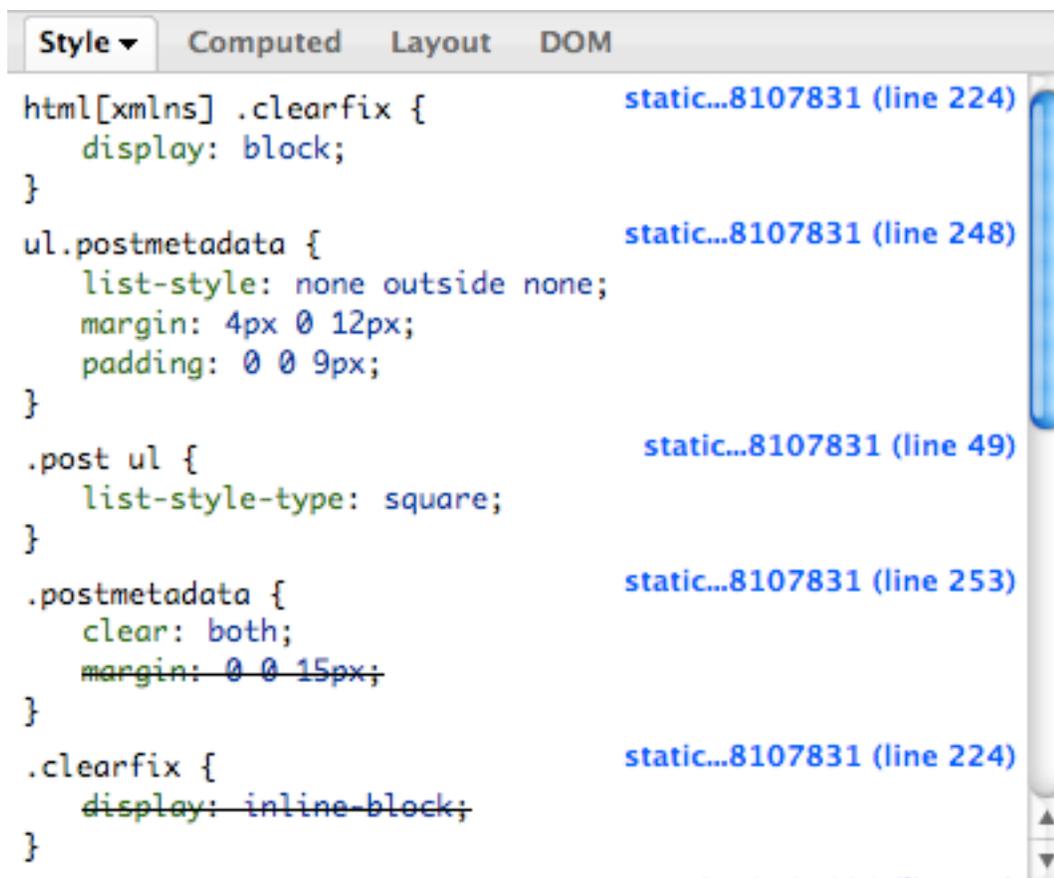
Don't break it by using the universal selector (`*`) with properties that inherit by default. Bobby Jack wrote [an interesting post about this](#) on his Five-Minute Argument blog. You don't have to remember all of the properties that inherit, but you will in time.

Rarely does a CSS-related article not bring some kind of bad news about Internet Explorer. This article is no exception. IE supports the `inherit` value only from version 8, except for the `direction` and `visibility` properties. Great.

4. Using Your Tools

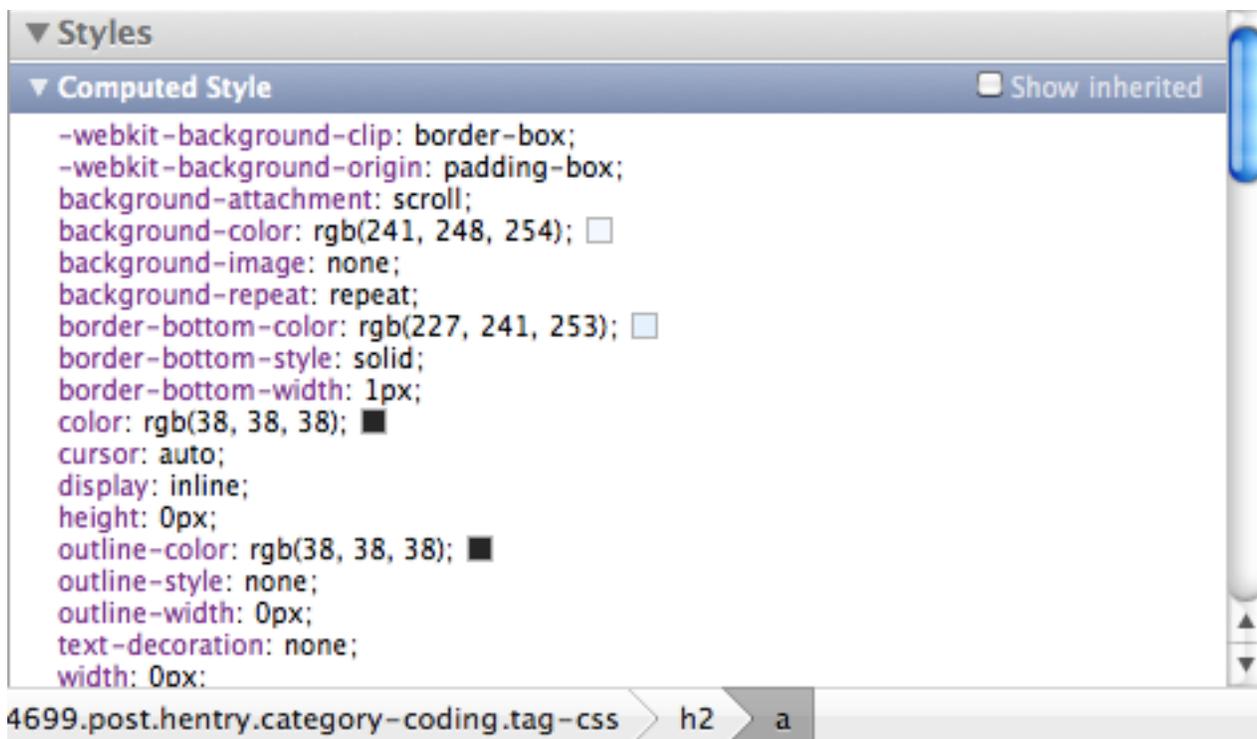
If you use tools like Firebug or Safari's Web Inspector, you can see how a given cascade works, which selectors have higher specificity and how inheritance is working on a particular element.

For example, here below is Firebug in action, inspecting an element on the page. You can see that some properties are overridden (i.e. crossed out) by other more specific rules:



Firebug in action, informing you how specificity is working.

In the next shot, Safari's Web Inspector shows the computed values of an element. This way, you can see the values even though they haven't been explicitly added to the style sheet:



With Safari's Web Inspector (and Firebug), you can view the computed values of a particular element.

5. Conclusion

Hopefully this article has opened your eyes to (or has refreshed your knowledge of) CSS inheritance and specificity. Even if you don't think about them, these issues are present in your daily work as a CSS author. Especially in the case of specificity, it's important to know how they affect your style sheets and how to plan for them so that they cause only minimal (or no) problems.

How to Use CSS3 Media Queries to Create a Mobile Website

Rachel Andrew

CSS3 continues to both excite and frustrate web designers and developers. We are excited about the possibilities that CSS3 brings, and the problems it will solve, but also frustrated by the lack of support in Internet Explorer 8. This article will demonstrate a technique that uses part of CSS3 that is also unsupported by Internet Explorer 8. However, it doesn't matter as one of the most useful places for this module is somewhere that does have a lot of support — small devices such as the iPhone and Android devices.

In this article I'll explain how, with a few CSS rules, you can create an iPhone version of your site using CSS3, that will work now. We'll have a look at a very simple example and I'll also discuss the process of adding a small screen device stylesheet to my own site to show how easily we can add stylesheets for mobile devices to existing websites.

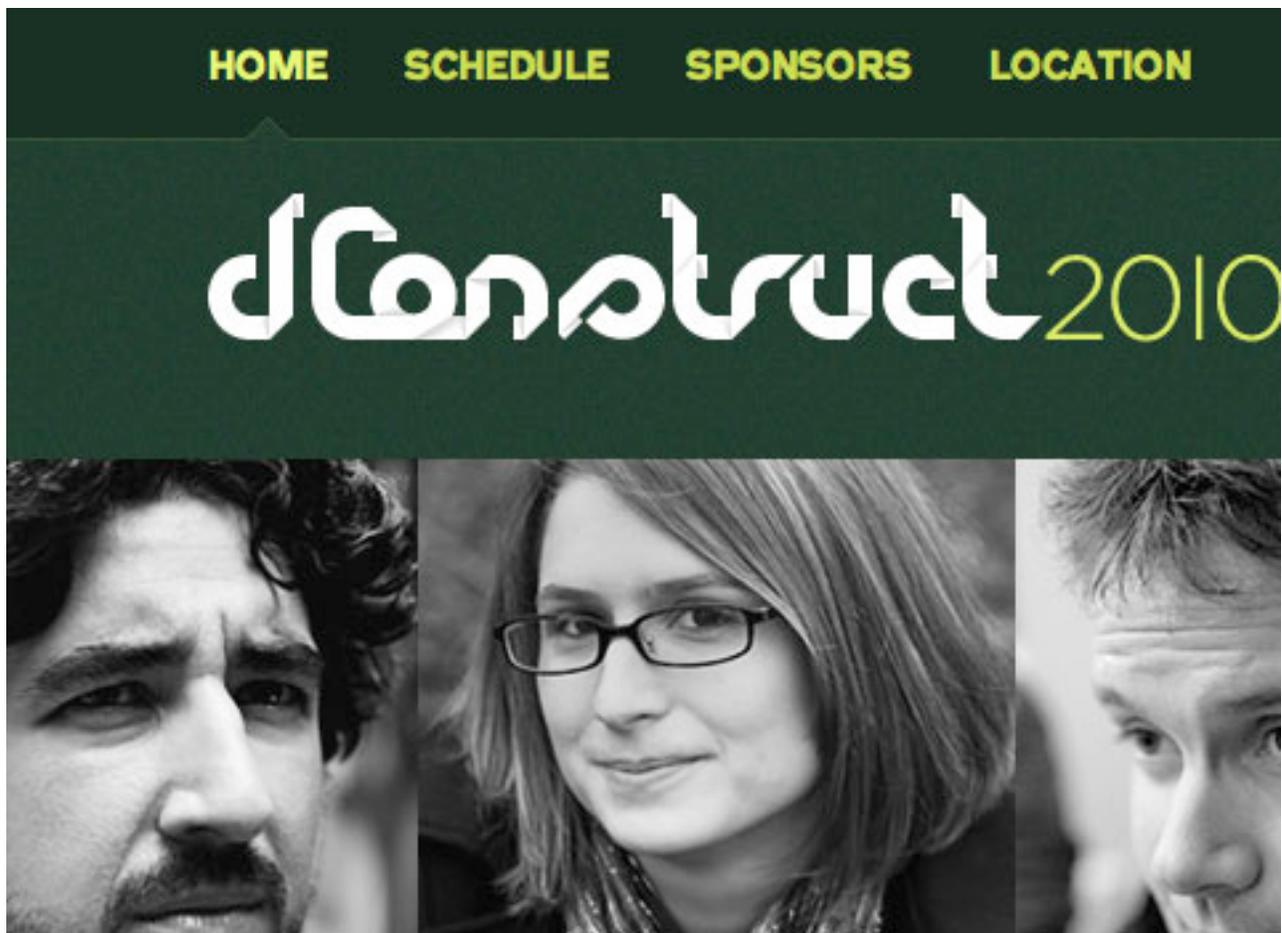
Media Queries

If you have ever created a print stylesheet for a website then you will be familiar with the idea of creating a specific stylesheet to come into play under certain conditions – in the case of a print stylesheet when the page is printed. This functionality was enabled in CSS2 by *media types*. Media Types let you specify a type of media to target, so you could target print, handheld and so on. Unfortunately, these media types never gained a lot of support by devices and, other than the print media type, you will rarely see them in use.

The Media Queries in CSS3 take this idea and extend it. Rather than looking for a *type* of device they look at the *capability* of the device, and you can use them to check for all kinds of things. For example:

- width and height (of the browser window)
- device width and height
- orientation – for example is a phone in landscape or portrait mode?
- resolution

If the user has a browser that supports media queries then we can write CSS specifically for certain situations. For example, detecting that the user has a small device like a smart phone of some description and giving them a specific layout. To see an example of this in practice, the UK Web conference dConstruct has just launched their website for the 2010 conference and this uses media queries to great effect.



The dConstruct 2010 website in Safari on a desktop computer



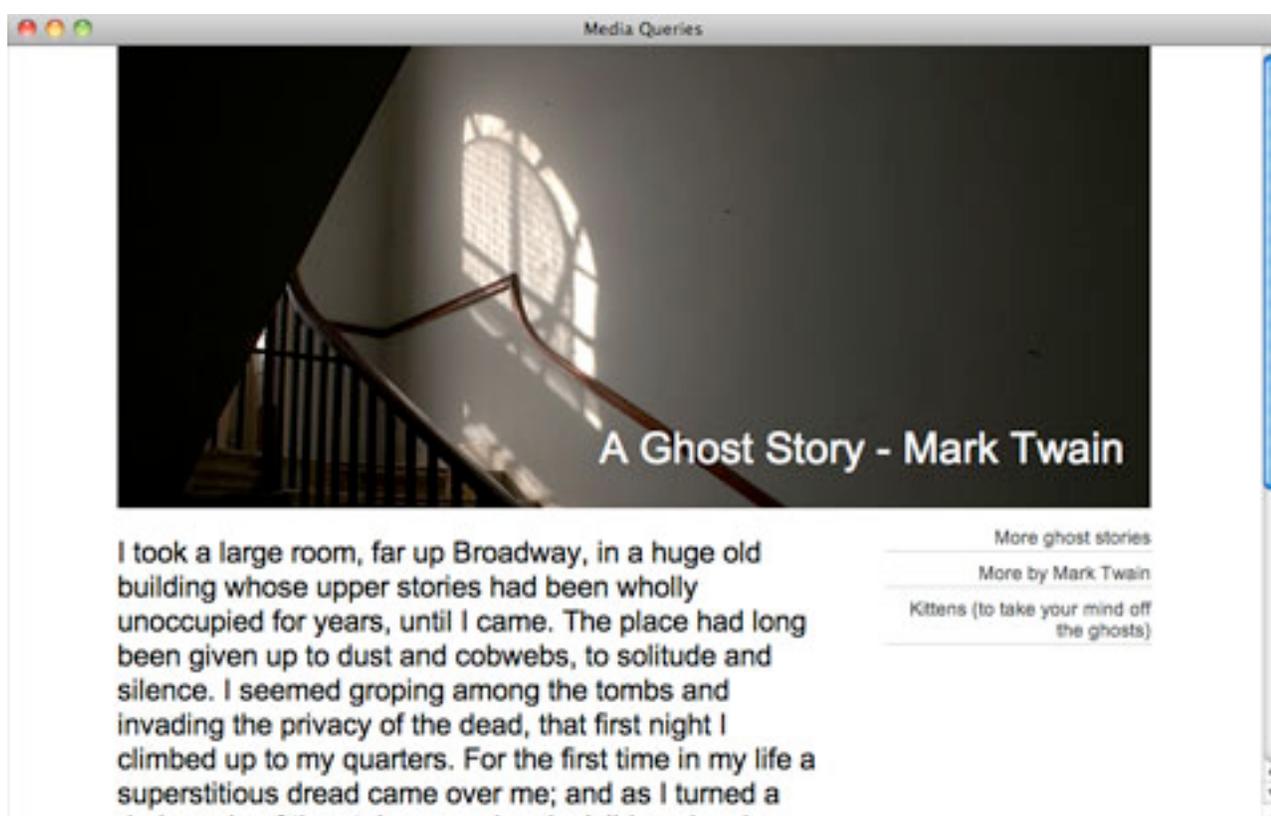
The dConstruct 2010 website on an iPhone

You can see from the above example that the site hasn't just been made smaller to fit, but that the content has actually been re-architected to be made more easy to access on the small screen of the device. In addition many people might think of this as being an iPhone layout – but it isn't. It displays in the same way on Opera Mini on my Android based phone – so

by using media queries and targeting the device capabilities the dConstruct site caters for all sorts of devices – even ones they might not have thought of!

Using Media Queries to create a stylesheet for phones

To get started we can take a look at a very simple example. The below layout is a very simple two column layout.



A very simple two column layout

To make it easier to read on a phone, I have decided to linearize the entire design making it all one column, and also to make the header area much smaller so readers don't need to scroll past the header before getting to any content.

The first way to use media queries is to have the alternate section of CSS right inside your single stylesheet. So to target small devices we can use the following syntax:

```
1 | @media only screen and (max-device-width: 480px) {  
2 |  
3 | }
```

We can then add our alternate CSS for small screen and width devices inside the curly braces. By using the cascade we can simply overwrite any styles rules we set for desktop browsers earlier in our CSS. As long as this section comes last in your CSS it will overwrite the previous rules. So, to linearize our layout and use a smaller header graphic, I can add the following:

```
1 | @media only screen and (max-device-width: 480px) {  
2 |   div#wrapper {  
3 |     width: 400px;  
4 |   }  
5 |  
6 |   div#header {  
7 |     background-image: url(media-queries-phone.jpg);  
8 |     height: 93px;  
9 |     position: relative;  
10 |   }  
11 | }
```

```
12  div#header h1 {
13      font-size: 140%;
14  }
15
16  #content {
17      float: none;
18      width: 100%;
19  }
20
21  #navigation {
22      float: none;
23      width: auto;
24  }
25 }
```

In the code above, I am using an alternate background image and reducing the height of the header, then setting the content and navigation to float none and overwriting the width set earlier in the stylesheet. These rules only come into play on a small screen device.



I took a large room, far up Broadway, in a huge old building whose upper stories had been wholly unoccupied for years, until I came. The place had long been given up to dust and cobwebs, to solitude and silence. I seemed groping among the tombs and invading the privacy of the dead, that first night I climbed up to my quarters. For the first time in my life a superstitious dread came over me; and as I turned a dark angle of the stairway and an invisible cobweb swung its slazy woof in my face and clung there, I shuddered as one who had encountered a phantom



My simple example as displayed on an iPhone

Linking a separate stylesheet using media queries

Adding the specific code for devices inline might be a good way to use media queries if you only need to make a few changes. However, if your stylesheet contains a lot of overwriting or you want to completely separate the styles shown to desktop browsers and those used for small screen devices, then linking in a different stylesheet will enable you to keep the CSS separate.

To add a separate stylesheet after your main stylesheet and use the cascade to overwrite the rules, use the following.

```
1 <link rel="stylesheet" type="text/css" media="only  
  screen and (max-device-width: 480px)" href="small-  
  device.css" />
```

Testing media queries

If you are the owner of an iPhone, Android device or other device that has a browser which supports media queries you can test your CSS yourself. However you will need to upload the code somewhere in order to view it. What about testing devices you don't own and testing locally?

An excellent site that can help you during the development process is ProtoFluid This application gives you a form to enter your URL – which can be a local URL – and view the design as if in the browser on an iPhone, iPad or a range of other devices. The screenshot below is the dConstruct site we looked at earlier as seen through the iPhone view on ProtoFluid.



The dConstruct 2010 website in ProtoFluid

You can also enter in your own window size if you have a specific device you want to test for and know the dimensions of it's screen.

To use ProtoFluid, you need to slightly modify the media query shown earlier to include `max-width` as well as `max-device-width`. This means that the media query also comes into play if the user has a normal desktop browser but is using a very tiny window.

```
1 | @media only screen and (max-width: 480px), only screen  
  | and (max-device-width: 480px) {  
2 |  
3 | }
```

After updating your code to the above, just refresh your page in the browser and then drag the window in and you should see the layout change as it hits 480 pixels. The media queries are now reacting when the viewport width hits the value you entered.

You are now all ready to test using ProtoFluid. The real beauty of ProtoFluid is that you can still use tools such as FireBug to tweak your design, something you won't have once on the iPhone. Obviously, you should still try and get a look at your layout in as many devices as possible, but ProtoFluid makes development and testing much simpler.

Note that if you don't want your site to switch layout when someone drags their window narrow you can always remove the max-width part of the query before going live, so the effect only happens for people with a small device and not just a small browser window.

Retrofitting an existing site

I have kept the example above very simple in order to demonstrate the technique. However, this technique could very easily be used to retrofit an existing site with a version for small screen devices. One of the big selling points of using CSS for layout was this ability to provide alternate versions of our design. As an experiment, I decided to take my own business website and apply these techniques to the layout.

The desktop layout

The website for my business currently has a multi-column layout. The homepage is a little different but in general we have a fixed width 3 column layout. This design is a couple of years old so we didn't consider media queries when building it.



My site in a desktop browser

Adding the new stylesheet

There will be a number of changes that I need to make to linearize this layout, so I'm going to add a separate stylesheet using media queries to load this stylesheet after the current stylesheet and only if the max-width is less than 480 pixels.

```
1 <link rel="stylesheet" type="text/css" media="only
  screen and (max-width: 480px), only screen and (max-
  device-width: 480px)" href="/assets/css/small-
  device.css" />
```

To create my new stylesheet, I take the default stylesheet for the site and save it as `small-device.css`. So this starts life as a copy of my main stylesheet. What I am going to do is go through and overwrite certain rules and then delete anything I don't need.

Shrinking the header

The first thing I want to do is make the logo fit nicely on screen for small devices. As the logo is a background image this is easy to do as I can load a different logo in this stylesheet. I also have a different background image with a shorter top area over which the logo sits.

```
1 body {
2   background-image: url(/img/small-bg.png);
3 }
4
5 #wrapper {
6   width: auto;
7   margin: auto;
8   text-align: left;
```

```
9 background-image: url(/img/small-logo.png);
10 background-position: left 5px;
11 background-repeat: no-repeat;
12 min-height: 400px;
13 }
```

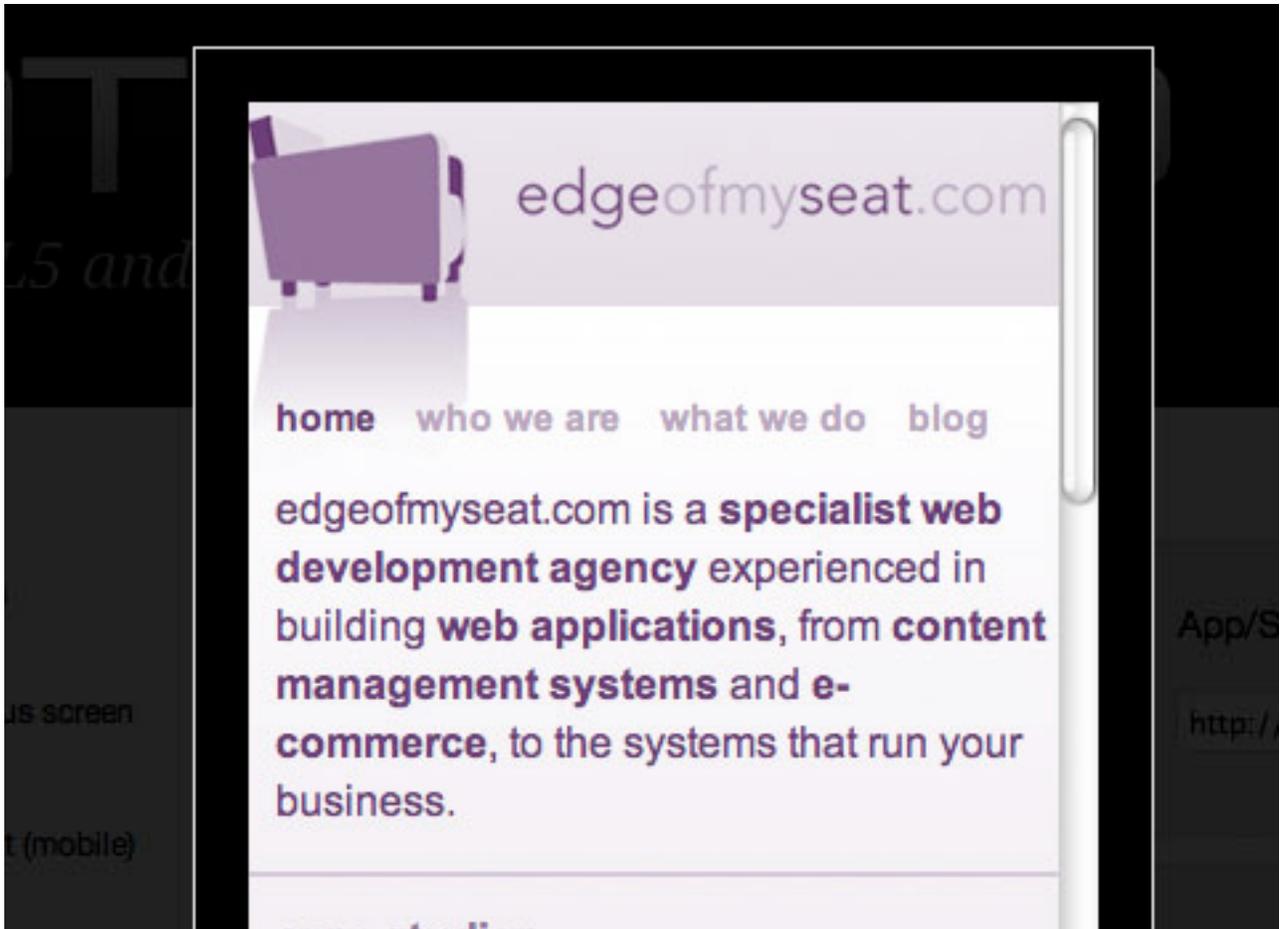
Linearizing the layout

The next main job is to linearize the layout and make it all one column. The desktop layout is created using floats so all I need to do is find the rules that float the columns, set them to `float: none` and `width: auto`. This drops all the columns one under another.

```
1 .article #aside {
2   float: none;
3   width: auto;
4 }
```

Tidying up

Everything after this point is really just a case of looking at the layout in ProtoFluid and tweaking it to give sensible amounts of margin and padding to areas that now are stacked rather than in columns. Being able to use Firebug in ProtoFluid makes this job much easier as my workflow mainly involves playing around using Firebug until I am happy with the effect and then copying that CSS into the stylesheet.

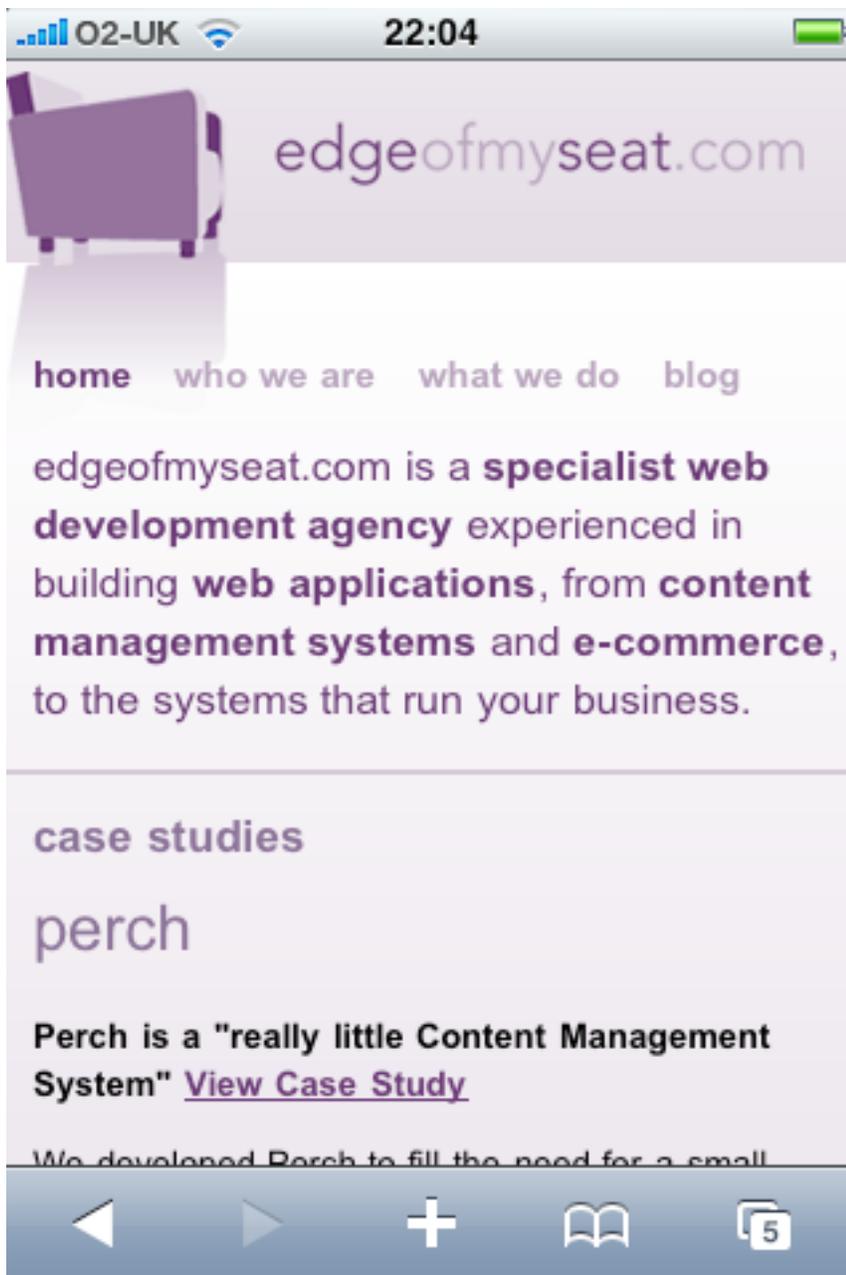


Testing in an iPhone

Having created my stylesheet and uploading it I wanted to check how it worked in a real target device. In the iPhone, I discovered that the site still loaded zoomed out rather than zooming in on my nice readable single column. A quick search on the Safari developer website gave me my answer – to add a meta tag to the head of the website setting the width of the viewport to the device width.

```
1 | <meta name="viewport" content="width=device-width" />
```

After adding the meta tag the site now displays zoomed in on the single column.



The site as it now displays on an iPhone

This was a very simple retrofit to show that it is possible to add a mobile version of your site simply. If I was building a site from scratch that I would be using media queries on, there are definitely certain choices I would make to make the process simpler. For example, considering the linearized column orders, using background images where possible, as these can be

switched using CSS, or perhaps using fluid images. Our desktop layout features a case studies carousel on the homepage, this wasn't easy to interact with on a touch screen device and so I checked using JavaScript if the browser window was very narrow and didn't launch the carousel. The way this was already written meant that the effect of stopping the carousel loading was that one case study would appear on the screen, which seems a reasonable solution for people on a small device. With a bit more time I could rewrite that carousel with an alternate version for users of mobile devices, perhaps with interactions more suitable to a touch screen.

Providing support for Media Queries in older browsers

This article covers the use of media queries in devices that have native support. If you are only interested in supporting iPhone and commonly used mobile browsers such as Opera Mini you have the luxury of not needing to worry about non-supporting browsers. If you want to start using media queries in desktop browsers then you might be interested to discover that there are a couple of techniques available which use JavaScript to add support to browsers such as Internet Explorer 8 (and lower versions) and Firefox prior to 3.5. Internet Explorer 9 will have support for CSS3 Media Queries.

Try it for yourself

Using Media Queries is one place you can really start to use CSS3 in your daily work. It is worth remembering that the browsers that support media queries also support lots of other CSS3 properties so your stylesheets that target these devices can also use other CSS3 to create a slick effect when viewed on an iPhone or other mobile device.

Responsive Web Design: What It Is and How to Use It

Kayla Knight

Almost every new client these days wants a mobile version of their website. It's practically essential after all: one design for the BlackBerry, another for the iPhone, the iPad, netbook, Kindle — and all screen resolutions must be compatible, too. In the next five years, we'll likely need to design for a number of additional inventions. When will the madness stop? It won't, of course.

In the field of Web design and development, we're quickly getting to the point of being unable to keep up with the endless new resolutions and devices. For many websites, creating a website version for each resolution and new device would be impossible, or at least impractical. Should we just suffer the consequences of losing visitors from one device, for the benefit of gaining visitors from another? Or is there another option?

Responsive Web design is the approach that suggests that design and development should respond to the user's behavior and environment based on screen size, platform and orientation. The practice consists of a mix of flexible grids and layouts, images and an intelligent use of CSS media queries. As the user switches from their laptop to iPad, the website should automatically switch to accommodate for resolution, image size and scripting abilities. In other words, the website should have the technology to automatically *respond* to the user's preferences. This would eliminate the need for a different design and development phase for each new gadget on the market.

The Concept Of Responsive Web Design

[Ethan Marcotte](#) wrote an introductory article about the approach, "[Responsive Web Design](#)," for A List Apart. It stems from the notion of responsive architectural design, whereby a room or space automatically adjusts to the number and flow of people within it:

"Recently, an emergent discipline called "responsive architecture" has begun asking how physical spaces can respond to the presence of people passing through them. Through a combination of embedded robotics and tensile materials, architects are experimenting with art installations and wall structures that bend, flex, and expand as crowds approach them. Motion sensors can be paired with climate control systems to adjust a room's temperature and ambient lighting as it fills with people. Companies have already produced "smart glass technology" that can automatically become opaque when a room's occupants reach a certain density threshold, giving them an additional layer of privacy."

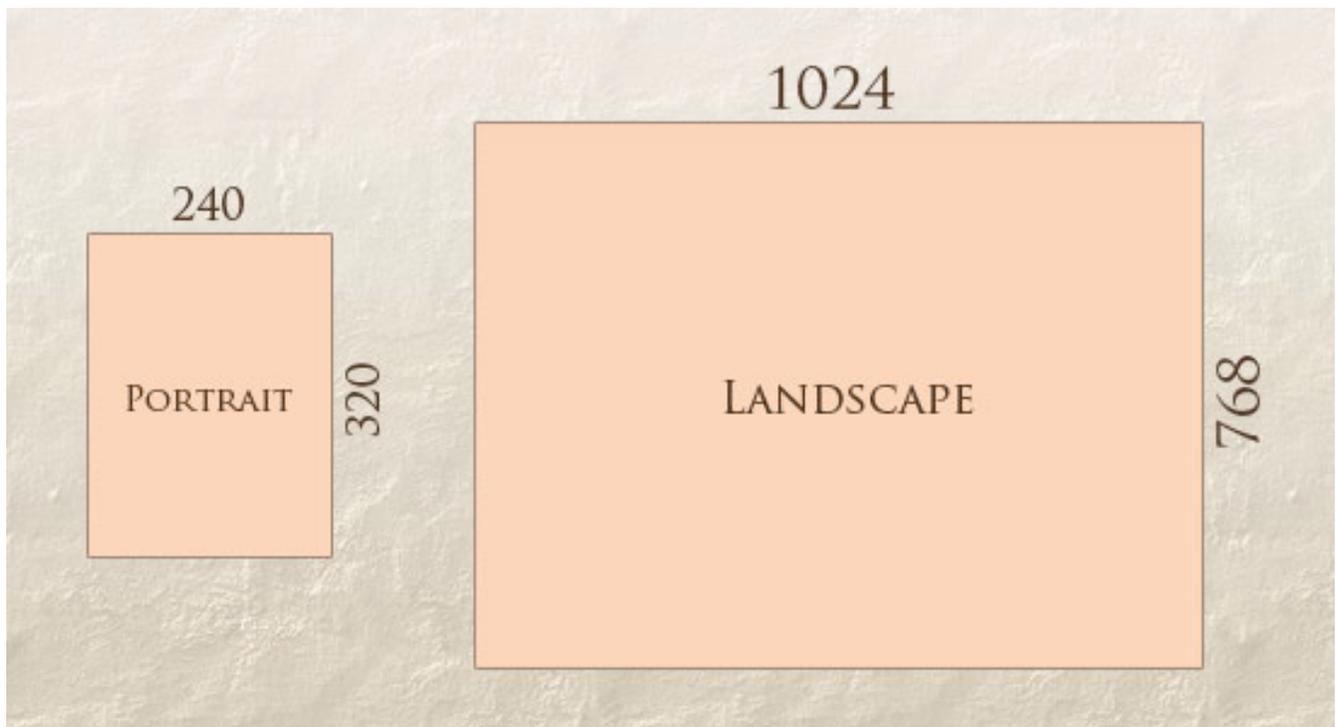
Transplant this discipline onto Web design, and we have a similar yet whole new idea. Why should we create a custom Web design for each group of users; after all, architects don't design another building for each group size and type that passes through it? Like responsive architecture, Web design should automatically adjust. It shouldn't require countless custom-made solutions for each new category of users.

Obviously, we can't use motion sensors and robotics to accomplish this the way a building would. Responsive Web design requires a more abstract way of thinking. However, some ideas are already being practiced: fluid layouts, media queries and scripts that can reformat Web pages and mark-up effortlessly (or *automatically*).

But responsive Web design is not only about adjustable screen resolutions and automatically resizable images, but rather about a whole new way of thinking about design. Let's talk about all of these features, plus additional ideas in the making.

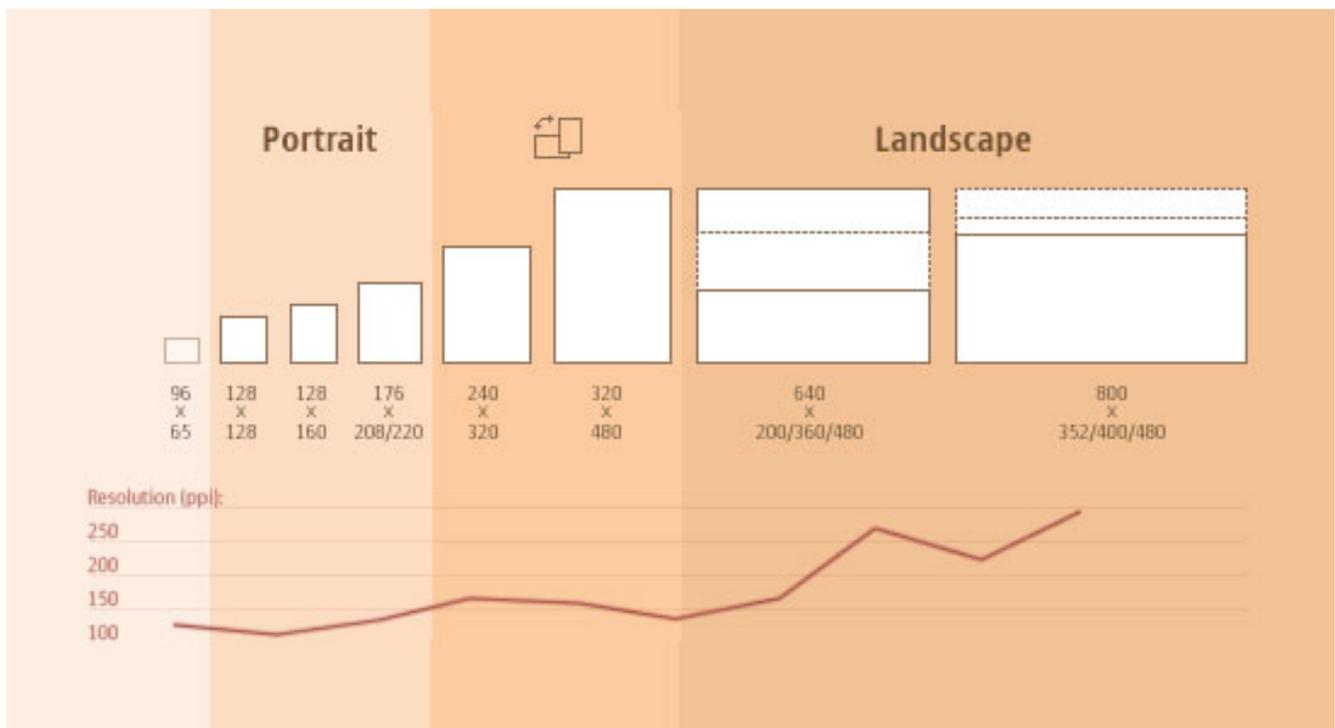
Adjusting Screen Resolution

With more devices come varying screen resolutions, definitions and orientations. New devices with new screen sizes are being developed every day, and each of these devices may be able to handle variations in size, functionality and even color. Some are in landscape, others in portrait, still others even completely square. As we know from the rising popularity of the iPhone, iPad and advanced smartphones, many new devices are able to switch from portrait to landscape at the user's whim. How is one to design for these situations?



In addition to designing for both landscape and portrait (and enabling those orientations to possibly switch in an instant upon page load), we must consider the hundreds of different screen sizes. Yes, it is possible to group them into major categories, design for each of them, and make each design as flexible as necessary. But that can be overwhelming, and who knows what the usage figures will be in five years? Besides, many users do not maximize their browsers, which itself leaves far too much room for variety among screen sizes.

Morten Hjerde and a few of his colleagues [identified statistics on about 400 devices](#) sold between 2005 and 2008. Below are some of the most common:



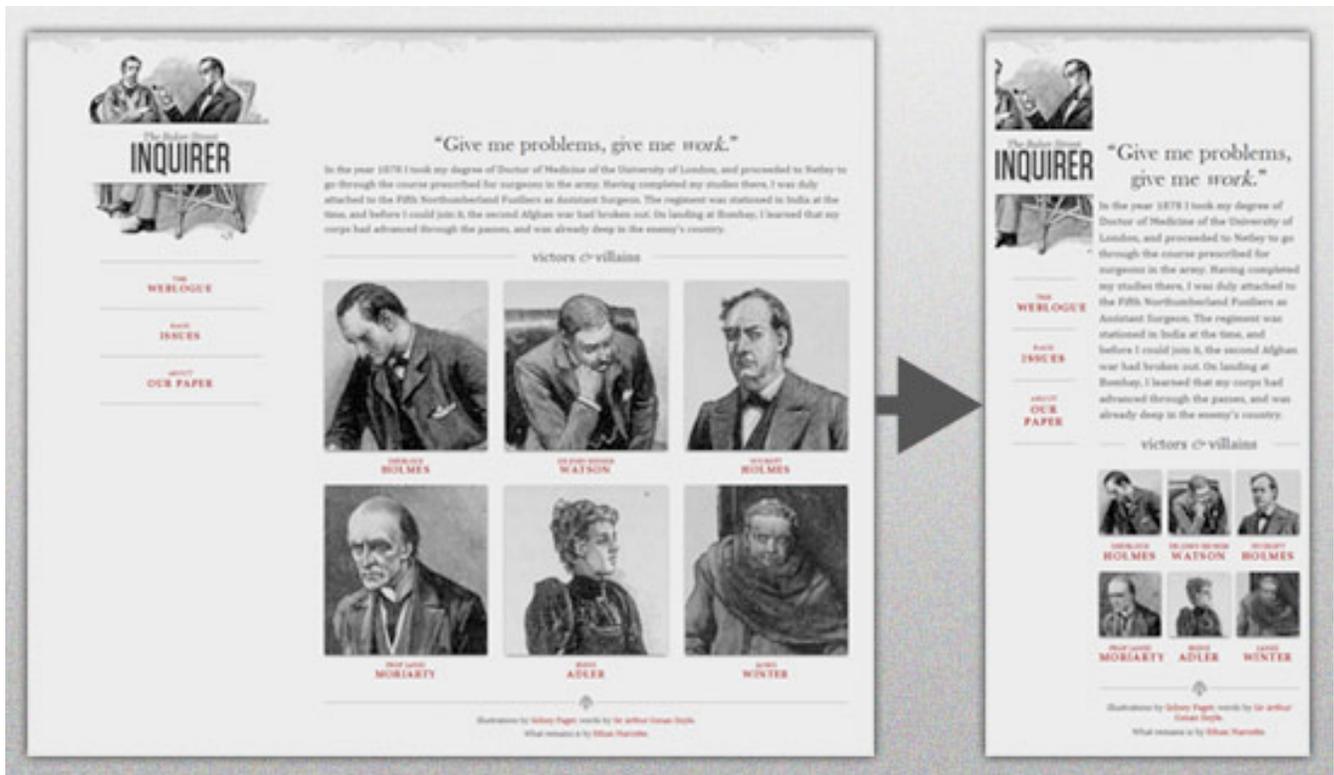
Since then even [more devices have come out](#). It's obvious that we can't keep creating custom solutions for each one. So, how do we deal with the situation?

Part of the Solution: Flexible Everything

A few years ago, when flexible layouts were almost a "luxury" for websites, the only things that were flexible in a design were the layout columns (structural elements) and the text. Images could easily break layouts, and even flexible structural elements broke a layout's form when pushed enough. Flexible designs weren't really that flexible; they could give or take a few hundred pixels, but they often couldn't adjust from a large computer screen to a netbook.

Now we can make things more flexible. Images can be automatically adjusted, and we have workarounds so that layouts never break (although they may become squished and illegible in the process). While it's not a complete fix, the solution gives us far more options. It's perfect for devices that switch from portrait orientation to landscape in an instant or for when users switch from a large computer screen to an iPad.

In Ethan Marcotte's article, he created a sample Web design that features this better flexible layout:



www.alistapart.com

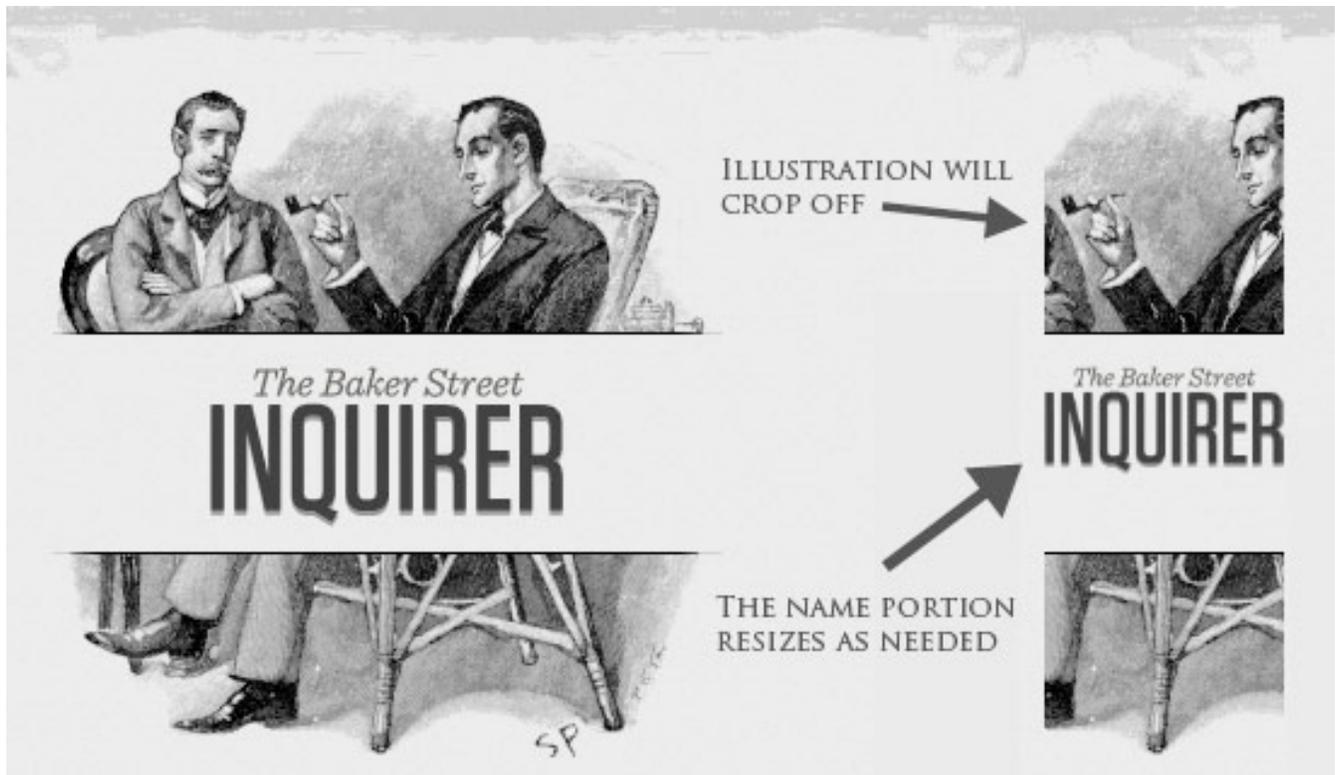
The entire design is a lovely mix of [fluid grids](#), [fluid images](#) and smart mark-up where needed. Creating fluid grids is fairly common practice, and there are a number of techniques for creating fluid images:

- [Hiding and Revealing Portions of Images](#)
- [Creating Sliding Composite Images](#)
- [Foreground Images That Scale With the Layout](#)

For more information on creating fluid websites, be sure to look at the book "Flexible Web Design: Creating Liquid and Elastic Layouts with CSS" by Zoe Mickley Gillenwater, and download the sample chapter "[Creating Flexible Images](#)." In addition, Zoe provides the following extensive list of

tutorials, resources, inspiration and best practices on creating flexible grids and layouts: "[Essential Resources for Creating Liquid and Elastic Layouts.](#)"

While from a technical perspective this is all easily possible, it's not just about plugging these features in and being done. Look at the logo in this design, for example:



www.alistapart.com

If resized too small, the image would appear to be of low quality, but keeping the name of the website visible and not cropping it off was important. So, the image is divided into two: one (of the illustration) set as a background, to be cropped and to maintain its size, and the other (of the name) resized proportionally.

```
1 | <h1 id="logo"><a href="#"></a></h1>
```

Above, the `h1` element holds the illustration as a background, and the image is aligned according to the container's background (the heading).

This is just one example of the kind of thinking that makes responsive Web design truly effective. But even with smart fixes like this, a layout can become too narrow or short to look right. In the logo example above (although it works), the ideal situation would be to not crop half of the illustration or to keep the logo from being so small that it becomes illegible and "floats" up.

Flexible Images

One major problem that needs to be solved with responsive Web design is working with images. There are a number of techniques to resize images proportionately, and many are easily done. The most popular option, noted in Ethan Marcotte's article on [fluid images](#) but first experimented with by [Richard Rutter](#), is to use CSS's `max-width` for an easy fix.

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

As long as no other width-based image styles override this rule, every image will load in its original size, unless the viewing area becomes narrower than the image's original width. The maximum width of the image is set to 100% of the screen or browser width, so when that 100% becomes narrower, so does the image. Essentially, as Jason Grigsby [noted](#):

"The idea behind fluid images is that you deliver images at the maximum size they will be used at. You don't declare the height and width in your code, but instead let the browser resize the images as

needed while using CSS to guide their relative size." It's a great and simple technique to resize images beautifully.

Note that `max-width` is not supported in IE, but a good use of `width: 100%` would solve the problem neatly in an IE-specific style sheet. One more issue is that when an image is resized too small in some older browsers in Windows, the rendering isn't as clear as it ought to be. There is a JavaScript to fix this issue, though, found in [Ethan Marcotte's article](#).

While the above is a great quick fix and good start to responsive images, image resolution and download times should be the primary considerations. While resizing an image for mobile devices can be very simple, if the original image size is meant for large devices, it could significantly slow download times and take up space unnecessarily.

Filament Group's Responsive Images

This technique, presented by the Filament Group, takes this issue into consideration and not only resizes images proportionately, but shrinks image resolution on smaller devices, so very large images don't waste space unnecessarily on small screens. Check out [the demo page here](#).

Responsive Images Test Page

In supported browsers, the following image will load either small or large version depending on screen resolution, making a single 1kb request before requesting the appropriate size.



filamentgroup.com

This technique requires a few files, all of which are available on [Github](#). First, a JavaScript file (*rwd-images.js*), the *.htaccess* file and an image file (*rwd.gif*). Then, we can use just a bit of HTML to reference both the larger and smaller resolution images: first, the small image, with a *.r* prefix to clarify that it should be responsive, and then a reference to the bigger image using `data-fullsrc`.

```
1 | 
```

The `data-fullsrc` is a custom HTML5 attribute, defined in the files linked to above. For any screen that is wider than 480 pixels, the larger-resolution image (*largeRes.jpg*) will load; smaller screens wouldn't need to load the bigger image, and so the smaller image (*smallRes.jpg*) will load.

The JavaScript file inserts a base element that allows the page to separate responsive images from others and redirects them as necessary. When the page loads, all files are rewritten to their original forms, and only the large or small images are loaded as necessary. With other techniques, all higher-resolution images would have had to be downloaded, even if the larger versions would never be used. Particularly for websites with a lot of images, this technique can be a great saver of bandwidth and loading time.

This technique is fully supported in modern browsers, such as IE8+, Safari, Chrome and Opera, as well as mobile devices that use these same browsers (iPad, iPhone, etc.). Older browsers and Firefox degrade nicely and still resize as one would expect of a responsive image, except that both resolutions are downloaded together, so the end benefit of saving space with this technique is void.

Stop iPhone Simulator Image Resizing

One nice thing about the iPhone and iPod Touch is that Web designs automatically rescale to fit the tiny screen. A full-sized design, unless specified otherwise, would just shrink proportionally for the tiny browser, with no need for scrolling or a mobile version. Then, the user could easily zoom in and out as necessary.

There was, however, one issue this simulator created. When responsive Web design took off, many noticed that images were still changing proportionally with the page even if they were specifically made for (or could otherwise fit) the tiny screen. This in turn scaled down text and other elements.

Because this works only with Apple's simulator, we can use an Apple-specific meta tag to fix the problem, placing it *below* the website's <head>

section. Thanks to [Think Vitamin's article on image resizing](#), we have the meta tag below:

```
1 <meta name="viewport" content="width=device-width;  
  initial-scale=1.0">
```

Setting the `initial-scale` to 1 overrides the default to resize images proportionally, while leaving them as is if their width is the same as the device's width (in either portrait or landscape mode). Apple's documentation has a lot more information on the [viewport meta tag](#).

Custom Layout Structure

For extreme size changes, we may want to change the layout altogether, either through a separate style sheet or, more efficiently, through a CSS media query. This does not have to be troublesome; most of the styles can remain the same, while specific style sheets can inherit these styles and move elements around with floats, widths, heights and so on.

For example, we could have one main style sheet (which would also be the default) that would define all of the main structural elements, such as `#wrapper`, `#content`, `#sidebar`, `#nav`, along with colors, backgrounds and typography. Default flexible widths and floats could also be defined.

If a style sheet made the layout too narrow, short, wide or tall, we could then detect that and switch to a new style sheet. This new child style sheet would adopt everything from the default style sheet and then just redefine the layout's structure.

Here is the ***style.css (default) content***:

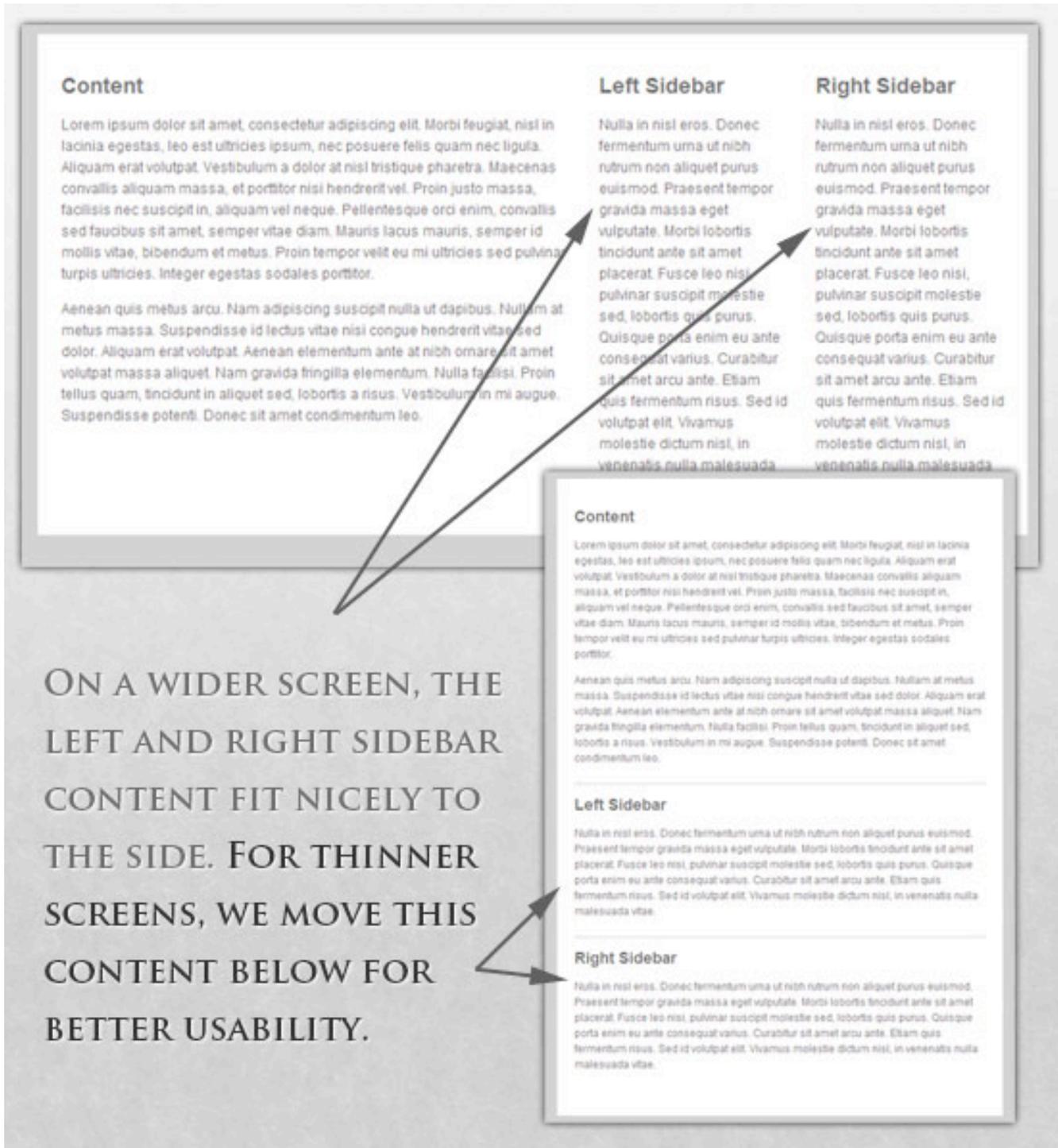
```
1 /* Default styles that will carry to the child style
   sheet */
2
3 html,body{
4   background...
5   font...
6   color...
7 }
8
9 h1,h2,h3{}
10 p, blockquote, pre, code, ol, ul{}
11
12 /* Structural elements */
13 #wrapper{
14   width: 80%;
15   margin: 0 auto;
16
17   background: #fff;
18   padding: 20px;
19 }
20
21 #content{
22   width: 54%;
23   float: left;
24   margin-right: 3%;
25 }
26
27 #sidebar-left{
28   width: 20%;
29   float: left;
30   margin-right: 3%;
```

```
31 }
32
33 #sidebar-right{
34     width: 20%;
35     float: left;
36 }
```

Here is the ***mobile.css (child)*** content:

```
1 #wrapper{
2     width: 90%;
3 }
4
5 #content{
6     width: 100%;
7 }
8
9 #sidebar-left{
10    width: 100%;
11    clear: both;
12
13    /* Additional styling for our new layout */
14    border-top: 1px solid #ccc;
15    margin-top: 20px;
16 }
17
18 #sidebar-right{
19     width: 100%;
20     clear: both;
21
22     /* Additional styling for our new layout */
```

```
23 border-top: 1px solid #ccc;
24 margin-top: 20px;
25 }
```



Media Queries

CSS3 supports all of the same media types as CSS 2.1, such as `screen`, `print` and `handheld`, but has added dozens of new media features, including `max-width`, `device-width`, `orientation` and `color`. New devices made after the release of CSS3 (such as the iPad and Android devices) will definitely support media features. So, calling a media query using CSS3 features to target these devices would work just fine, and it will be ignored if accessed by an older computer browser that does not support CSS3.

In Ethan Marcotte's article, we see an example of a media query in action:

```
1 <link rel="stylesheet" type="text/css"  
2   media="screen and (max-device-width: 480px)"  
3   href="shetland.css" />
```

This media query is fairly self-explanatory: if the browser displays this page on a screen (rather than print, etc.), and if the width of the screen (not necessarily the viewport) is 480 pixels or less, then load *shetland.css*.

New CSS3 features also include `orientation` (portrait vs. landscape), `device-width`, `min-device-width` and more. Look at "[The Orientation Media Query](#)" for more information on setting and restricting widths based on these media query features.

One can create multiple style sheets, as well as basic layout alterations defined to fit ranges of widths — even for landscape vs. portrait orientations. Be sure to look at the section of Ethan Marcotte's article entitled "[Meet the media query](#)" for more examples and a more thorough explanation.

Multiple media queries can also be dropped right into a single style sheet, which is the most efficient option when used:

```
1 /* Smartphones (portrait and landscape) ----- */
2 @media only screen
3 and (min-device-width : 320px)
4 and (max-device-width : 480px) {
5 /* Styles */
6 }
7
8 /* Smartphones (landscape) ----- */
9 @media only screen
10 and (min-width : 321px) {
11 /* Styles */
12 }
13
14 /* Smartphones (portrait) ----- */
15 @media only screen
16 and (max-width : 320px) {
17 /* Styles */
18 }
```

The code above is from a free template for multiple media queries between popular devices by Andy Clark. See the differences between this approach and including different style sheet files in the mark-up as shown in the post [“Hardboiled CSS3 Media Queries.”](#)

CSS3 Media Queries

Above are a few examples of how media queries, both from CSS 2.1 and CSS3 could work. Let’s now look at some specific how-to’s for using CSS3

media queries to create responsive Web designs. Many of these uses are relevant today, and all will definitely be usable in the near future.

The **min-width** and **max-width** properties do exactly what they suggest. The `min-width` property sets a minimum browser or screen width that a certain set of styles (or separate style sheet) would apply to. If anything is below this limit, the style sheet link or styles will be ignored. The `max-width` property does just the opposite. Anything above the maximum browser or screen width specified would not apply to the respective media query.

Note in the examples below that we're using the syntax for media queries that could be used all in one style sheet. As mentioned above, the most efficient way to use media queries is to place them all in one CSS style sheet, with the rest of the styles for the website. This way, multiple requests don't have to be made for multiple style sheets.

```
1 @media screen and (min-width: 600px) {  
2   .hereIsMyClass {  
3     width: 30%;  
4     float: right;  
5   }  
6 }
```

The class specified in the media query above (`hereIsMyClass`) will work only if the browser or screen width is above 600 pixels. In other words, this media query will run only if the **minimum width is 600 pixels** (therefore, 600 pixels or wider).

```
1 @media screen and (max-width: 600px) {  
2   .aClassforSmallScreens {  
3     clear: both;  
4   }  
5 }
```

```
4     font-size: 1.3em;
5   }
6 }
```

Now, with the use of `max-width`, this media query will apply only to browser or screen widths with a maximum width of 600 pixels or narrower.

While the above `min-width` and `max-width` can apply to either screen size or browser width, sometimes we'd like a media query that is relevant to device width specifically. This means that even if a browser or other viewing area is minimized to something smaller, the media query would still apply to the size of the actual device. The **min-device-width** and **max-device-width** media query properties are great for targeting certain devices with set dimensions, without applying the same styles to other screen sizes in a browser that mimics the device's size.

```
1 @media screen and (max-device-width: 480px) {
2   .classForiPhoneDisplay {
3     font-size: 1.2em;
4   }
5 }
```

```
1 @media screen and (min-device-width: 768px) {
2   .minimumiPadWidth {
3     clear: both;
4     margin-bottom: 2px solid #ccc;
5   }
6 }
```

There are also other tricks with media queries to target specific devices. Thomas Maier has written two short snippets and explanations for targeting the iPhone and iPad only:

-
- [CSS for iPhone 4 \(Retina display\)](#)
 - [How To: CSS for the iPad](#)

For the iPad specifically, there is also a media query property called **orientation**. The value can be either `landscape` (horizontal orientation) or `portrait` (vertical orientation).

```
1 @media screen and (orientation: landscape) {
2   .iPadLandscape {
3     width: 30%;
4     float: right;
5   }
6 }
```

```
1 @media screen and (orientation: portrait) {
2   .iPadPortrait {
3     clear: both;
4   }
5 }
```

Unfortunately, this property works only on the iPad. When [determining the orientation for the iPhone](#) and other devices, the use of `max-device-width` and `min-device-width` should do the trick.

There are also many media queries that make sense when combined. For example, the `min-width` and `max-width` media queries are combined all the time to set a style specific to a certain range.

```
1 @media screen and (min-width: 800px) and (max-width:
  1200px) {
2   .classForaMediumScreen {
3     background: #cc0000;
4     width: 30%;
5     float: right;
6   }
7 }
```

The above code in this media query applies only to screen and browser widths between 800 and 1200 pixels. A good use of this technique is to show certain content or entire sidebars in a layout depending on how much horizontal space is available.

Some designers would also prefer to link to a separate style sheet for certain media queries, which is perfectly fine if the organizational benefits outweigh the efficiency lost. For devices that do not switch orientation or for screens whose browser width cannot be changed manually, using a separate style sheet should be fine.

You might want, for example, to place media queries all in one style sheet (as above) for devices like the iPad. Because such a device can switch from portrait to landscape in an instant, if these two media queries were placed in separate style sheets, the website would have to call each style sheet file every time the user switched orientations. Placing a media query for both the horizontal and vertical orientations of the iPad in the same style sheet file would be far more efficient.

Another example is a flexible design meant for a standard computer screen with a resizable browser. If the browser can be manually resized, placing all variable media queries in one style sheet would be best.

Nevertheless, organization can be key, and a designer may wish to define media queries in a standard HTML link tag:

```
1 <link rel="stylesheet" media="screen and (max-width:
  600px)" href="small.css" />
2 <link rel="stylesheet" media="screen and (min-width:
  600px)" href="large.css" />
3 <link rel="stylesheet" media="print"
  href="print.css" />
```

JavaScript

Another method that can be used is JavaScript, especially as a back-up to devices that don't support all of the CSS3 media query options. Fortunately, there is already a pre-made JavaScript library that makes older browsers (IE 5+, Firefox 1+, Safari 2) support CSS3 media queries. If you're already using these queries, just grab a copy of the library, and include it in the mark-up: [css3-mediaqueries.js](#).

In addition, below is a sample jQuery snippet that detects browser width and changes the style sheet accordingly — if one prefers a more hands-on approach:

```
1 <script type="text/javascript" src="http://
  ajax.googleapis.com/ajax/libs/jquery/1.4.4/jquery.min.js "></
  script>
2
3 <script type="text/javascript">
4   $(document).ready(function() {
5     $(window).bind("resize", resizeWindow);
6     function resizeWindow(e) {
```

```
7     var newWindowWidth = $(window).width();
8
9     // If width width is below 600px, switch to the
mobile stylesheet
10    if(newWindowWidth < 600){           $
    ("link[rel=stylesheet]").attr({href :
    "mobile.css"});                       // Else if
width is above 600px, switch to the large stylesheet
    else if(newWindowWidth > 600){
11        $("link[rel=stylesheet]").attr({href :
    "style.css"});
12    }
13 }
14 });
15 </script>
```

There are many solutions for pairing up JavaScript with CSS media queries. Remember that media queries are not an absolute answer, but rather are fantastic options for responsive Web design when it comes to pure CSS-based solutions. With the addition of JavaScript, we can accommodate far more variations. For detailed information on using JavaScript to mimic or work with media queries, look at "[Combining Media Queries and JavaScript](#)."

Showing or Hiding Content

It is possible to shrink things proportionally and rearrange elements as necessary to make everything fit (reasonably well) as a screen gets smaller. It's great that that's possible, but making every piece of content from a large screen available on a smaller screen or mobile device isn't always the

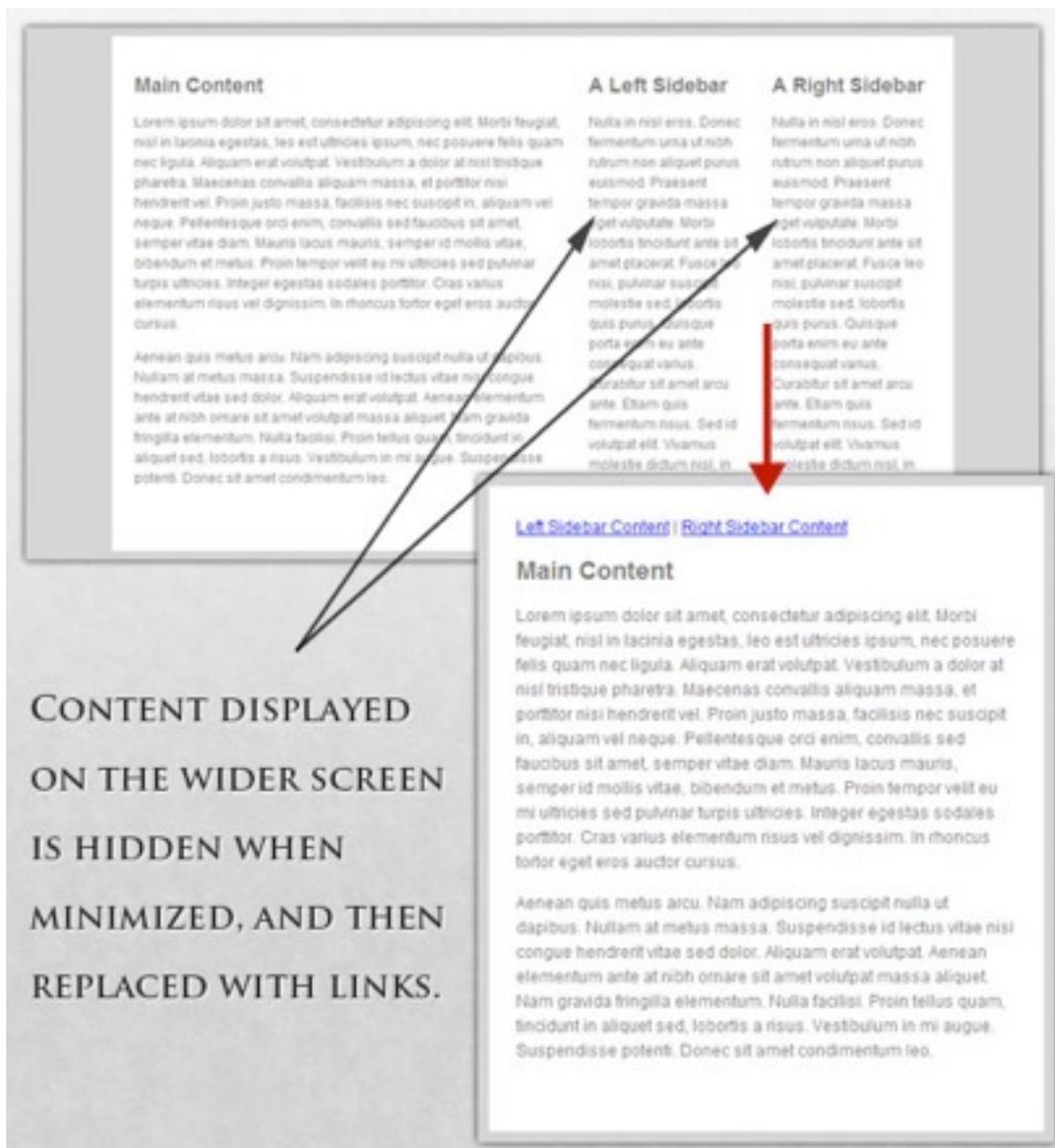
best answer. We have best practices for mobile environments: simpler navigation, more focused content, lists or rows instead of multiple columns.

Responsive Web design shouldn't be just about how to create a flexible layout on a wide range of platforms and screen sizes. It should also be about the user being able to pick and choose content. Fortunately, CSS has been allowing us to show and hide content with ease for years!

```
1 | display: none;
```

Either declare `display: none` for the HTML block element that needs to be hidden in a specific style sheet or detect the browser width and do it through JavaScript. In addition to hiding content on smaller screens, we can also hide content in our default style sheet (for bigger screens) that should be available only in mobile versions or on smaller devices. For example, as we hide major pieces of content, we could replace them with navigation to that content, or with a different navigation structure altogether.

Note that we haven't used `visibility: hidden` here; this just hides the content (although it is still there), whereas the `display` property gets rid of it altogether. For smaller devices, there is no need to keep the mark-up on the page — it just takes up resources and might even cause unnecessary scrolling or break the layout.



CONTENT DISPLAYED ON THE WIDER SCREEN IS HIDDEN WHEN MINIMIZED, AND THEN REPLACED WITH LINKS.

Here is **our mark-up**:

```
1 <p class="sidebar-nav"><a href="#">Left Sidebar
  Content</a> | <a href="#">Right Sidebar Content</a></
  p>
2
3 <div id="content">
4   <h2>Main Content</h2>
5 </div>
6
7 <div id="sidebar-left">
8   <h2>A Left Sidebar</h2>
9
10 </div>
11
12 <div id="sidebar-right">
13   <h2>A Right Sidebar</h2>
14 </div>
```

In our default style sheet below, we have hidden the links to the sidebar content. Because our screen is large enough, we can display this content on page load.

Here is the ***style.css* (default)** content:

```
1 #content{
2   width: 54%;
3   float: left;
4   margin-right: 3%;
5 }
6
7 #sidebar-left{
```

```
8 width: 20%;
9 float: left;
10 margin-right: 3%;
11 }
12
13 #sidebar-right{
14 width: 20%;
15 float: left;
16 }
17 .sidebar-nav{display: none;}
```

Now, we hide the two sidebars (below) and show the links to these pieces of content. As an alternative, the links could call to JavaScript to just cancel out the `display: none` when clicked, and the sidebars could be realigned in the CSS to float below the content (or in another reasonable way).

Here is the ***mobile.css (simpler)*** content:

```
1 #content{
2 width: 100%;
3 }
4
5 #sidebar-left{
6 display: none;
7 }
8
9 #sidebar-right{
10 display: none;
11 }
12 .sidebar-nav{display: inline;}
```

With the ability to easily show and hide content, rearrange layout elements and automatically resize images, form elements and more, a design can be transformed to fit a huge variety of screen sizes and device types. As the screen gets smaller, rearrange elements to fit mobile guidelines; for example, use a script or alternate style sheet to increase white space or to replace image navigation sources on mobile devices for better usability (icons would be more beneficial on smaller screens).

Touchscreens vs. Cursors

Touchscreens are becoming increasingly popular. Assuming that smaller devices are more likely to be given touchscreen functionality is easy, but don't be so quick. Right now touchscreens are mainly on smaller devices, but many laptops and desktops on the market also have touchscreen capability. For example, the *HP Touchsmart tm2t* is a basic touchscreen laptop with traditional keyboard and mouse that can transform into a tablet.

Touchscreens obviously come with different design guidelines than purely cursor-based interaction, and the two have different capabilities as well. Fortunately, making a design work for both doesn't take a lot of effort. Touchscreens have no capability to display CSS hovers because there is no cursor; once the user touches the screen, they click. So, don't rely on CSS hovers for link definition; they should be considered an additional feature only for cursor-based devices.

Look at the article "[Designing for Touchscreen](#)" for more ideas. Many of the design suggestions in it are best for touchscreens, but they would not necessarily impair cursor-based usability either. For example, sub-navigation on the right side of the page would be more user-friendly for

touchscreen users, because most people are right-handed; they would therefore not bump or brush the navigation accidentally when holding the device in their left hand. This would make no difference to cursor users, so we might as well follow the touchscreen design guideline in this instance. Many more guidelines of this kind can be drawn from touchscreen-based usability.

The Future Of CSS: Experimental CSS Properties

Christian Krammer

Despite contemporary browsers supporting a wealth of CSS3 properties, most designers and developers seem to focus on the quite harmless properties such as `border-radius`, `box-shadow` or `transform`. These are well documented, well tested and frequently used, and so it's almost impossible to not stumble on them these days if you are designing websites.

But hidden deep within the treasure chests of browsers are advanced, heavily underrated properties that don't get that much attention. Perhaps some of them rightly so, but others deserve more recognition. The greatest wealth lies under the hood of WebKit browsers, and in the age of iPhone, iPad and Android apps, getting acquainted with them can be quite useful. Even the Gecko engine, used by Firefox and the like, provides some distinct properties. In this article, we will look at some of the less known CSS 2.1 and CSS3 properties and their support in modern browsers.

Some explanation: For each property, I state the support: "WebKit" means that it is available only in browsers that use the WebKit engine (Safari, Chrome, iPhone, iPad, Android), and "Gecko" indicates the availability in Firefox and the like. Finally, certain properties are part of the official [CSS 2.1](#) specification, which means that a broad range of browsers, even older ones, support them. Finally, a label of [CSS3](#) indicates adherence to this specification, supported by the latest browser versions, such as Firefox 4, Chrome 10, Safari 5, Opera 11.10 and Internet Explorer 9.

WebKit-Only Properties

-webkit-mask

This property is quite extensive, so a detailed description is beyond the scope of this article and is certainly worth a more detailed examination, especially because it could turn out to be a time-saver in practical applications.

`-webkit-mask` makes it possible to apply a mask to an element, thereby enabling you to create a cut-out of any shape. The mask can either be a CSS3 gradient or a semi-transparent PNG image. An alpha value of 0 would cover the underlying element, and 1 would fully reveal the content behind. Related properties like `-webkit-mask-clip`, `-webkit-mask-position` and `-webkit-mask-repeat` rely heavily on the syntax of the ones from [background](#). For more info, see the [Surfin' Safari blog](#) and the link below.

flickr.com © agwagon2000



Original image

+



Mask (Image)

=



Masked image



Original image

+



Mask (gradient)

=



Masked image

Example

Image mask:

```
1 | .element {  
2 |   background: url(img/image.jpg) repeat;  
3 |   -webkit-mask: url(img/mask.png);  
4 | }
```

Example

Gradient mask:

```
1 | .element2 {  
2 |   background: url(img/image.jpg) repeat;  
3 |   -webkit-mask: -webkit-gradient(linear, left top, left  
4 |   bottom, from(rgba(0,0,0,1)), to(rgba(0,0,0,0)));  
5 | }
```

-webkit-text-stroke

One of the shortcomings of CSS borders is that only rectangular ones are possible. A ray of hope is `-webkit-text-stroke`, which gives text a border. Setting not only the width but the color of the border is possible. And in combination with `color: transparent`, you can create outlined text.

Examples

Assigns a blue border with a 2-pixel width to all `<h1>` headings:

```
1 | h1 {-webkit-text-stroke: 2px blue}
```

Another feature is the ability to smooth text by setting a transparent border of 1 pixel:

```
1 | h2 {-webkit-text-stroke: 1px transparent}
```

Creates text with a red outline:

```
1 | h3 {  
2 |   color: transparent;  
3 |   -webkit-text-stroke: 4px red;  
4 | }
```

Smashing Magazine

-webkit-nbsp-mode

Wrapping can be pretty tricky. Sometimes you want text to break (and not wrap) at certain points, and other times you don't want this to happen. One property to control this is `-webkit-nbsp-mode`. It lets you change the behavior of the ` ` character, forcing text to break even where it is used. This behavior is enabled by the value `space`.

-webkit-tap-highlight-color

This one is just for iOS (iPhone and iPad). When you tap on a link or a JavaScript clickable element, it is highlighted by a semi-transparent gray background. To override this behavior, you can set `-webkit-tap-highlight-color` to any color. To disable this highlighting, a color with an alpha value of 0 must be used.

Example

Sets the highlight color to red, with a 50% opacity:

```
1 | -webkit-tap-highlight-color: rgba(255, 0, 0, 0.5)
```

Supported by: iOS only (iPhone and iPad).

zoom: reset

Normally, `zoom` is an Internet Explorer-only [property](#). But in combination with the value `reset`, WebKit comes into play (which, funny enough, IE doesn't support). It enables you to override the standard behavior of zooming on websites. If set with a CSS declaration, everything except the given element is enlarged when the user zooms on the page.

-webkit-margin-collapse

Here is a property with a quite limited practical use, but it is still worth mentioning. By default, the margins of two adjacent elements collapse, which means that the bottom distance of the first element and the top distance of the second element merge into a single gap.

The best example is two `<p>`s that share their margins when placed one after another. To control this behavior, we can use `-webkit-margin-collapse`, `-webkit-margin-top-collapse` or `-webkit-margin-bottom-collapse`. The standard value is `collapse`. The separate value stops the sharing of margins, which means that both the bottom margin of the first element and the top margin of the second are included.

`-webkit-margin-collapse: collapse`

1st
<p/>

Lorem ipsum dolor sit amet,
consectetur adipiscing elit.
Aenean commodo ligula eget
dolor. Aenean massa.

`margin 1st <p/> + margin 2nd <p/>`

2nd
<p/>

Integer tincidunt. Cras dapibus.
Vivamus elementum semper
nisi. Aenean vulputate eleifend
tellus.

`-webkit-margin-collapse: separate`

Lorem ipsum dolor sit amet,
consectetur adipiscing elit.
Aenean commodo ligula eget
dolor. Aenean massa.

`margin 1st <p/>`

`margin 2nd <p/>`

Integer tincidunt. Cras dapibus.
Vivamus elementum semper
nisi. Aenean vulputate eleifend
tellus.

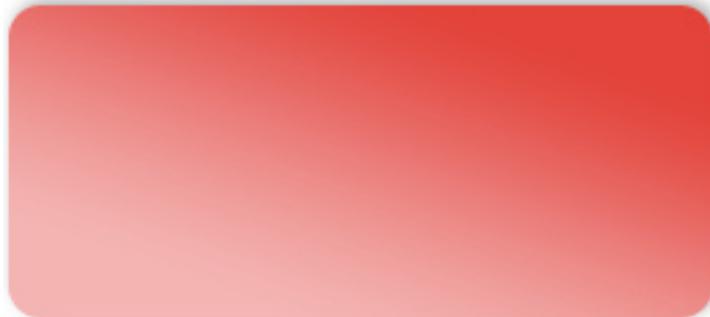
-webkit-box-reflect

Do you remember the days when almost every website featured a reflection of either its logo or some text in the header? Thankfully, those days are gone, but if you'd like to make a subtle use of this technique for your buttons, navigation or other UI elements with CSS, then `-webkit-box-reflect` is the property for you.

It accepts the keywords `above`, `below`, `left` and `right`, which set where the reflection is drawn, as well as a numeric value that sets the distance between the element and its reflection. Beyond that, mask images are supported as well (see `-webkit-mask` for an explanation of masks). The reflection is created automatically and has no effect on the layout. Following elements are created using only CSS, and the second button is reflected using the `-webkit-box-reflect` property.



Original element



Reflection of the element

Examples

This reflection would be shown under its parent element and have a spacing of 5 pixels:

```
1 | -webkit-box-reflect: below 5px;
```

This reflection would be cast on the right side of the element, with no distance (0); additionally, a mask would be applied (`url(mask.png)`):

```
1 | -webkit-box-reflect: right 0 url(mask.png);
```

-webkit-marquee

Here is another property that recalls the good ol' days when marquees were quite common. Interesting that this widely dismissed property turns out to be useful today, when we shift content on tiny mobile screens that would otherwise not be fully visible without wrapping.

The weather application by [ozPDA](#) makes great use of it. (If you don't see shifting text, just select another city at the bottom of the app. WebKit browser required.)

Example

```
1 .marquee {  
2   white-space: nowrap;  
3   overflow: -webkit-marquee;  
4   width: 70px;  
5   -webkit-marquee-direction: forwards;  
6   -webkit-marquee-speed: slow;  
7   -webkit-marquee-style: alternate;  
8 }
```

There are some prerequisites for the marquee to work. First, `white-space` must be set to `nowrap` if you want the text to be on one line. Also, `overflow` must be set to `-webkit-marquee`, and `width` set to something narrower than the full length of the text.

The remaining properties ensure that the text scrolls from left to right (`-webkit-marquee-direction`), shifts back and forth (`-webkit-marquee-style`) and moves at a slow rate (`-webkit-marquee-speed`). Additional properties are `-webkit-marquee-repetition`, which sets how many iterations the marquee should pass through, and `-webkit-marquee-increment`, which defines the degree of speed in each increment.

Gecko-Only Properties

font-size-adjust

Unfortunately, this useful CSS3 property is supported only by Firefox at the moment. We can use it to specify that the font size for a given element should relate to the height of lowercase letters (x-height) rather than the height of uppercase letters (cap height). For example, Verdana is much more legible at the same size than Times, which has a much shorter x-height. To compensate for this behavior, we can adjust the latter with `font-size-adjust`.

This property is particularly useful in CSS font stacks whose fonts have different x-heights. Even if you're careful to use only similar fonts, `font-size-adjust` can provide a solution when problems arise.

Example

If Verdana is not installed on the user's machine for some reason, then Arial is adjusted so that it has the same aspect ratio as Verdana, which is 0.58 (at a font size of 12px, differs on other sizes).

```
1 p {  
2   font-family: Verdana, Arial, sans-serif;  
3   font-size: 12px;  
4   font-size-adjust: 0.58;  
5 }
```

Unadjusted text

Lorem ipsum dolor sit amet.

.....
Lorem ipsum dolor sit amet.

Verdana

Arial

font-size-adjust: 0.58 for Arial

.....
Lorem ipsum dolor sit amet.

.....
Lorem ipsum dolor sit amet.

Supported by: Gecko.

image-rendering

A few years ago, images that were not displayed at their original size and were scaled by designers, could appear unattractive or just plain wrong in the browser, depending on the size and context. Nowadays, browsers have a much better algorithm for displaying resized images, however, it's great to have a full control over the ways your images will be displayed when scaled, especially with responsive images becoming a de facto standard in responsive Web designs.

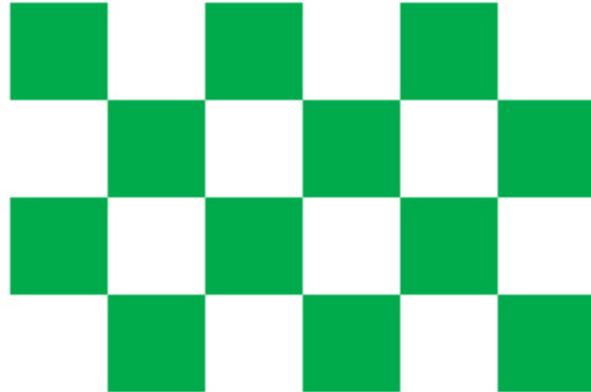
This Gecko-specific property is particularly useful if you have an image with sharp lines and want to maintain them after resizing. The relevant value would be `-moz-crisp-edges`. The same algorithm is used at `optimizeSpeed`, whereas `auto` and `optimizeQuality` indicate the standard behavior (which is to resize elements with the best possible quality). The `image-rendering` property can also be applied to `<video>` and `<canvas>` elements, as well as background images. It is a CSS3 property, but is currently supported only by Firefox.



`optimizeQuality`



`-moz-crisp-edges`



`optimizeQuality`



`-moz-crisp-edges`



It's also worth mentioning `-ms-interpolation-mode: bicubic`, although it is a proprietary Internet Explorer property. Nevertheless, it enables Internet Explorer 7 to render images at a much higher quality after resizing which is useful because by default this browser handles such tasks pretty poorly.

Supported by: Gecko.

-moz-border-top-colors

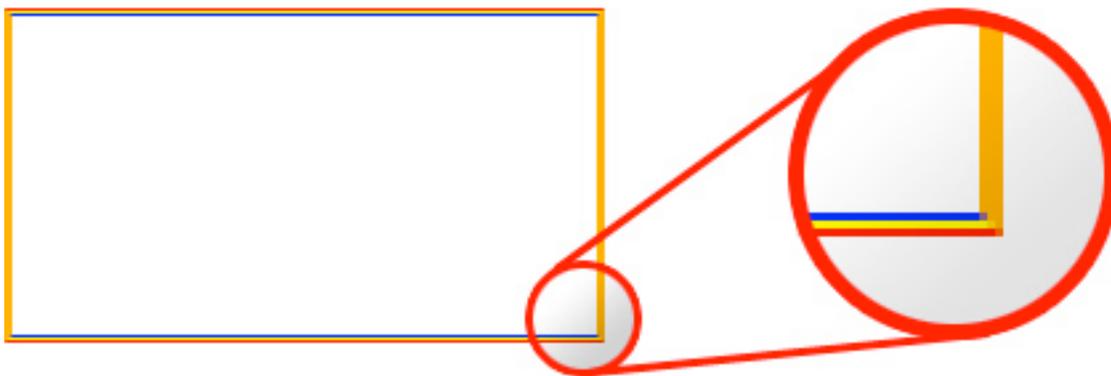
This property could be filed under 'eye-candy'. It allows you to assign different colors to borders that are wider than 1 pixel. Also available are `-moz-border-bottom-colors`, `-moz-border-left-colors` and `-moz-border-right-colors`.

Unfortunately, there is no condensed version like `-moz-border-colors` for this property, so the `border` property must be set in order for it to work, whereas `border-width` should be the same as the number of the given color values. If it is not, then the last color value is taken for the rest of the border.

Example

Below, the element's border would have a standard color of orange applied to the left and right side (because `-moz-border-left-colors` and `-moz-border-right-colors` are not set). The top and bottom borders have a kind of gradient, with the colors red, yellow and blue.

```
1 | div {  
2 |   border: 3px solid orange;  
3 |   -moz-border-top-colors: red yellow blue;  
4 |   -moz-border-bottom-colors: red yellow blue;  
5 | }
```



Supported by: Gecko.

Mixed Properties

-webkit-user-select and -moz-user-select

There might be times when you don't want users to be able to select text, whether to protect it from copying or for another reason. One solution is to set `-webkit-user-select` and `-moz-user-select` to `none`. Please use this property with caution: since most users are looking for information that they can copy and store for future reference, this property is neither helpful nor effective. In the end, the user could always look up the source code and take the content even if you have forbidden the traditional copy-and-paste. We do not know why this property exists in both WebKit and Gecko browsers.

Supported by: WebKit, Gecko.

-webkit-appearance and -moz-appearance

Ever wanted to easily camouflage an image to look like a radio button? Or an input field to look like a checkbox? Then `appearance` will come in handy. Even if you wouldn't always want to mask a link so that it looks like a button (see example below), it's nice to know that you can do it if you want.

Example

```
1 a {  
2   -webkit-appearance: button;  
3   -moz-appearance: button;  
4 }
```

Supported by: WebKit, Gecko.

text-align: -webkit-center/-moz-center

This is one property (or value, to be exact) whose existence is quite surprising. To center a block-level element, one would usually set `margin` to `0 auto`. But you could also set the `text-align` property of the element's container to `-moz-center` and `-webkit-center`. You can align left and right with `-moz-left` and `-webkit-left` and then `-moz-right` and `-webkit-right`, respectively.

Supported by: WebKit, Gecko.

CSS 2.1. Properties

counter-increment

How often have you wished you could automatically number an ordered list or all of the headings in an article? Unfortunately, there is still no CSS3 property for that. But let's look back to CSS 2.1, in which `counter-increment` provides a solution. That means it's been around for several years, and even supported in Internet Explorer 8. Did you know that? Me neither.

In conjunction with the `:before` pseudo-element and the `content` property, `counter-increment` can add automatic numbering to any HTML tag. Even nested counters are possible.

Example

For numbered headings, first reset the counter to start at 1:

```
1 |body {counter-reset: thecounter}
```

Every `<h1>` would get the prefix "Section," including a counter that automatically increments by 1 (which is default and can be omitted), where `thecounter` is the name of the counter:

```
1 | .counter h1:before {
2 |   counter-increment: thecounter 1;
3 |   content: "Section"counter(thecounter) ":";
4 | }
```

Example

For a nested numbered list, the counter is reset and the automatic numbering of `` is switched off because it features no nesting:

```
1 | ol {
2 |   counter-reset: section;
3 |   list-style-type: none;
4 | }
```

Then, every `` is given automatic incrementation, and the separator is set to be a point (.), followed by a blank.

```
1 | li:before {
2 |   counter-increment: section;
3 |   content: counters(section, ".") " ";
4 | }
```

```
1 <ol>
2   <li>item</li>           <!-- 1 -->
3   <li>item                 <!-- 2 -->
4     <ol>
5       <li>item</li>       <!-- 1.1 -->
6       <li>item</li>       <!-- 1.2 -->
7     </ol>
8   </li>
9   <li>item</li>           <!-- 3 -->
10 </ol>
```

Supported by: CSS 2.1., all modern browsers, IE 7+.

quotes

Are you tired of using wrong quotes just because your CMS doesn't know how to properly convert them to the right ones? Then start using the `quotes` property to set them how you want. This way, you can use any character. You would then assign the quotes to the desired element using the `:before` and `:after` pseudo-elements. Unfortunately, the otherwise progressive WebKit browsers don't support this property, which means no quotes are shown at all.

Example

The first two characters determine the quotes for the first level of a quotation, the last two for the second level, and so on:

```
1 q {
2   quotes: '«' '»' "<" ">";
3 }
```

These two lines assign the quotes to the selected element:

```
1 | q:before {content: open-quote}
2 | q:after  {content: close-quote}
```

So, `<p><q>This is a very <q>nice</q> quote.</q> </p>` would give us: **«This is a very <nice> quote.»**

Supported by: CSS 2.1., all browsers except WebKit, even IE 7+.

Question: To add the character directly, does the CSS document have to have a UTF-8 character set? That's a tough one. Unfortunately, I can't give a definitive answer. My experimentation has shown that no character set has to be set for the `quotes` property to work properly. However the `utf-8` character set doesn't work because it shows "broken" characters (for example, "»"). With the `iso-8859-1` character set, everything works fine.

This is how the W3C [describes it](#): "While the quotation marks specified by 'quotes' in the previous examples are conveniently located on computer keyboards, high-quality typesetting would require different ISO 10646 characters."

CSS3 Properties You May Have Heard About But Can't Remember

To round out things, let's go over some CSS3 properties that are not well known and maybe not as appealing as the classic ones `border-radius` and `box-shadow`.

text-overflow

Perhaps you're familiar with this problem: a certain area is too small for the text that it contains, and you have to use JavaScript to cut the string and append "..." so that it doesn't blow out the box.

Forget that! With CSS3 and `text-overflow: ellipsis`, you can force text to automatically end with "..." if it is longer than the width of the container. The only requirement is to set `overflow` to `hidden`. Unfortunately, this is not supported by Firefox but will hopefully be implemented in a coming release.

Example

```
1 | div {  
2 |   width: 100px;  
3 |   text-overflow: ellipsis;  
4 | }
```

`overflow: hidden`

Lorem ipsum dolor sit amet,
consectetur
<http://www.thisisawebsitewithaverylongurl.com>
Aenean commodo ligula eget dolor.
Aenean massa. Cum sociis natoque
penatibus et magnis dis parturient
montes, nascetur ridiculus mus.

`overflow: hidden +
text-overflow: ellipsis`

Lorem ipsum dolor sit amet,
consectetur
<http://www.thisisawebsitewithave...>
Aenean commodo ligula eget dolor.
Aenean massa. Cum sociis natoque
penatibus et magnis dis parturient
montes, nascetur ridiculus mus.

Supported by: CSS 3, all browsers except Firefox, even IE6+.

word-wrap

With text in a narrow column, sometimes portions of it are too long to wrap correctly. Link URLs especially cause trouble. If you don't want to hide the overflowing text with `overflow: hidden`, then you can set `word-wrap` to `break-word`, which causes it to break when it reaches the limit of the container.

Example

```
1 | div {  
2 |   width: 50px;  
3 |   word-wrap: break-word;  
4 | }
```

Default behavior

Donec quam felis, ultricies nec, pellentesque http://www.thisisaweb sitewithalongurl.com eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu.

word-wrap: break-word

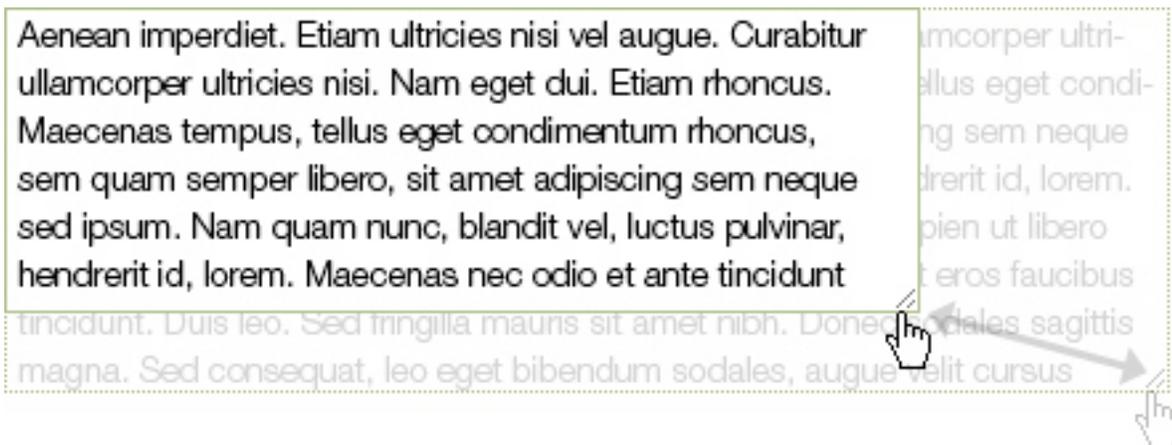
Donec quam felis, ultricies nec, pellentesque http://www.thisisaweb sitewith alongurl.com eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu.

Supported by: CSS 3, all browsers, even IE6+.

resize

If you use Firefox or Chrome, then you must have noticed that text areas by default have a little handle in the bottom-right corner that lets you resize them. This standard behavior is achieved by the CSS3 property `resize: both`.

But it's not limited to text areas. It can be used on any HTML element. The `horizontal` and `vertical` values limit the resizing to the horizontal and vertical axes, respectively. The only requirement is that `overflow` be set to anything other than `visible`.



Supported by: CSS3, all the latest browsers except Opera and Internet Explorer.

background-attachment

When you assign a background image to an element that is set to `overflow: auto`, it is fixed to the background and doesn't scroll. To disable this behavior and enable the image to scroll with the content, set `background-attachment` to `local`.

Default behavior



`background-attachment: local`



Supported by: CSS 3, all the latest browsers except Firefox.

text-rendering

With more and more websites rendering fonts via the `@font-face` attribute, legibility becomes a concern. Problems can occur particularly at small font sizes. While there is still no CSS property to control the subtle details of displaying fonts online, you can enable [kerning](#) and [ligatures](#) via `text-rendering`.

Gecko and WebKit browsers handle this property quite differently. The former enables these features by default, while you have to set it to `optimizeLegibility` in the latter.

`text-rendering: optimizeSpeed`

LYoWAT - ff fi fl ffl

`text-rendering: optimizeLegibility`

LYoWAT - ff fi fl ffl

Supported by: CSS3, all WebKit browsers and Firefox.

transform: rotateX/transform: rotateY

If you've already dived into CSS3 and transformations a bit, then you're probably familiar with `transform: rotate()`, which rotates an element around its z-axis.

But did you know that it is also possible to spin it "into the deep" (i.e. around its x-axis and y-axis)? These transformations are particularly useful in combination with `-webkit-backface-visibility: hidden`, if you want to rotate an element and reveal another one at its back. This technique is described by Andy Clarke in his latest book, *Hardboiled Web Design*, and it can be seen in action on a [demo page](#).

Example

If you hover over the element, it will turn by 180°, revealing its back:

```
1 | div:hover {  
2 |   transform: rotateY(180deg);  
3 | }
```

rotateX
Front view



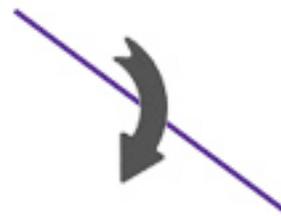
Side view



rotateY
Front view



Plan view



Quick tip: To just mirror an element, you can either set `transform` to `rotateX(180deg)` (and respectively `rotateY`) or set `transform` to `scaleX(-1)` (and respectively `scaleY`).

Supported by: CSS3, only WebKit browsers, in combination with `-webkit-backface-visibility` only Safari and iOS (iPhone and iPad).

Some Last Words

As you hopefully have seen, there are many unknown properties that range from being nice to have to being very useful. Many of them are still at an experimental stage and may never leave it or even be discarded in future

browser releases. Others will hopefully be adopted by all browser manufacturers in coming versions.

While it is hard to justify using some of them, the WebKit-specific ones are gaining more and more importance with the success of the iOS devices and Android. And of course some CSS3 properties are more or less ready to be used now.

And if you don't like vendor-specific properties, you can see them as experiments that still could be implemented in the code to improve the user experience for users browsing with the modern browsers. By the way, [CSS validator](#) from the W3C now also supports vendor-specific properties, which result in warnings rather than errors.

Happy experimenting!

Technical Web Typography: Guidelines and Techniques

Harry Roberts

The Web is [95% typography](#), or so they say. I think this is a pretty accurate statement: we visit websites largely with the intention of reading. That's what you're doing now — reading. With this in mind, does it not stand to reason that your typography should be one of the most considered aspects of your designs?

Unfortunately, for every person who is obsessed with even the tiniest details of typography, a dozen or so people seem to be indifferent. It's a shame; if you're going to spend time writing something, don't you want it to look great and be easy to read?

Creative and Technical Typography

I'm not sure these two categories are recognized in the industry but, in my mind, the two main types of typography are *creative* and *technical*.

Creative typography involves making design decisions such as which face to use, what mood the type should create, how it should be set, what tone it should have — for example, should it be airy, spacious and open (light) or condensed, bold and tight, with less white space (dark)? These decisions must be made on a per-project basis. You probably wouldn't use the same font on a girl's party invitation and an obituary. For me, this is creative typography: it is design-related and changes according to its application.

Technical typography is like type theory; certain rules and practices apply to party invitations just as well as they do to obituaries. These are little rules that always hold, are proven to work and are independent of design. The good news is that, because they are rules, even the most design-challenged people can make use of them and instantly raise the quality of their text from bog-standard to bang-tidy.

We'll focus on technical type in this article. We'll discuss the intricacies and nuances of a small set of rules while learning the code to create them.

We'll learn about:

- How to choose a font face
- How to choose a font size
- Using a grid
- Working out the measure
- Vertical rhythm and baseline grids
- Choosing a typographic scale
- How to use proper quotes
- How to use proper dashes
- How to use proper ellipses
- How to hang punctuation
- Dealing with images in grids

Fair warning: this is an in-depth article. It requires some basic CSS knowledge. If you'd rather learn a little at a time, use the links above to jump from section to section.

If any of the code examples seem out of context or confusing, then [here is the final piece](#) that we're going to create (merely for your reference).

Setting Things Up

To begin, copy and paste this into an *index.html* file, and save it to your desktop:

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4 <meta charset="utf-8" />
5 <title>Your Name</title>
6 <link rel="stylesheet" type="text/css" href="css/
  style.css" />
7 </head>
8 <body>
9
10 <h1>Your Name</h1>
11
12 </body>
13 </html>
```

Next, copy and paste this (slightly modified) CSS reset into your *style.css* sheet, and save that to your machine, too:

```
1 /*-----*\
2 RESET
3 \*-----*/
4 body, div, dl, dt, dd, ul, ol, li,
5 h1, h2, h3, h4, h5, h6,
```

```
6 pre, form, fieldset, input, textarea,
7 p, blockquote, th, td {
8   margin: 0;
9   padding: 0;
10 }
11 table {
12   border-collapse: collapse;
13   border-spacing: 0;
14 }
15 fieldset, img {
16   border: 0;
17 }
18 address, caption, cite, dfn, th, var {
19   font-style: normal;
20   font-weight: normal;
21 }
22 caption, th {
23   text-align: left;
24 }
25 h1, h2, h3, h4, h5, h6 {
26   font-size: 100%;
27   font-weight: normal;
28 }
29 q:before, q:after {
30   content: '';
31 }
32 abbr, acronym {
33   border: 0;
34 }
35
36 /*-----*\
```

```
37 | MAIN
38 | \*-----*/
39 | html {
40 |   background: #fff;
41 |   color: #333;
42 | }
```

Choosing A Font Face

First, let's choose a face in which to set our project. There is, as you know, a solid base of Web-safe fonts to choose from. There are also amazing services like [Fontdeck](#) and [Typekit](#) that leverage `@font-face` to add non-standard fonts in a fairly robust way.

We're not going to use any of those, though. To prove that technical type can make *anything* look better, let's restrict ourselves to a typical font stack.

Let's use a serif stack for this project, because technical type works wonders on serif faces:

```
1 | html {
2 |   font-family: Cambria, Georgia, "Times New Roman",
   | Times, serif;
3 |   background: #fff;
4 |   color: #333;
5 | }
```

Cambria is a beautiful font, specifically designed for on-screen reading and to be aesthetically pleasing when printed at small sizes. If you want to alter this or use a sans-serif stack, be my guest.

On Using Helvetica

If you'd like to use Helvetica in your stack, remember that Helvetica [looks awful](#) as body copy on a PC. To alleviate this, use the following trick to serve Helvetica to Macs and Arial to PCs (you can find more details about this trick in Chris Coyier's recent article [Sans-Serif](#)):

```
1 html {  
2   font-family: sans-serif; /* Serve the machine's  
   default sans face. */  
3   background: #fff;  
4   color: #333;  
5 }
```

Beware! This is a hack. It works by using a system's default sans font as the font for the page. By default, a Mac will use Helvetica and a PC will use Arial. However, if a user has changed their system preferences, this will not be the case, so use with caution.

Choosing A Font Size

Oliver Reichenstein authored [an inspiring article](#), way back in 2006, stating that the ideal size for type on the Web is 16 pixels: the user agents' standard. This insightful article changed the way I work with type; it's well worth a read. We'll use 16 pixels as a base size, then. If you want to use another font size, feel free, but if you stick with 16 pixels, your CSS should look something like this:

```
1 html {
2   font-family: Cambria, Georgia, "Times New Roman",
   Times, serif;
3   background: #fff;
4   color: #333;
5 }
```

If you want to use, say, 12 pixels, it will look like this:

```
1 html {
2   font-family: Cambria, Georgia, "Times New Roman",
   Times, serif;
3   font-size: 0.75em; /* 16 * 0.75 = 12 */
4   background: #fff;
5   color: #333;
6 }
```

You'll be left with a [basic layout \(demo\)](#).

Choosing A Grid System

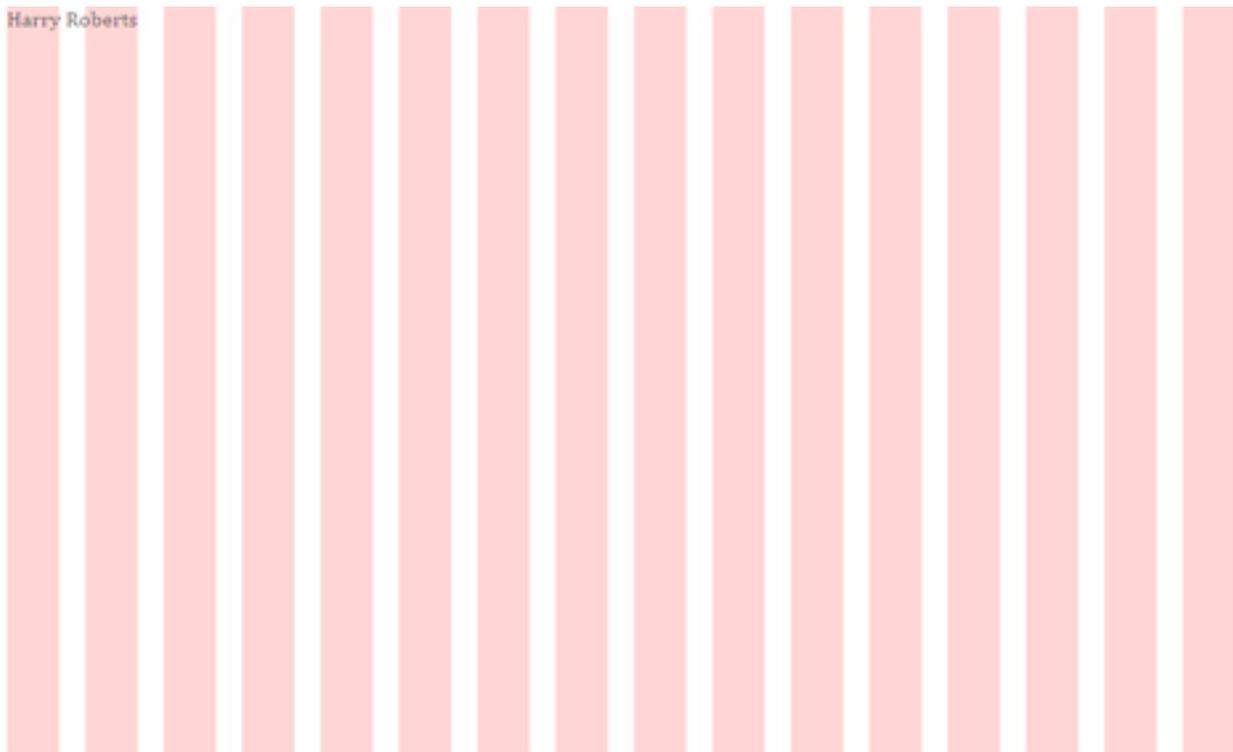
The grid is an amazing tool, and it's not just for typographical ventures. It ensures order and harmony in your designs.

Some grid systems out there, in my opinion, go a little overboard and offer 30 or more columns, all awkwardly sized. For this tutorial, we'll use [Nathan Smith's 16-column 960 Grid System \(demo\)](#). 960.gs is amazing; its beauty lies in its simplicity. It is an ideal size for designs narrower than 1020 pixels, it has a good number of columns, and the numbers are easy to work with. You might also notice that the 960 Grid System only has 940 pixels of usable space. "960" comes from the 10 pixels of gutter space on either side.

Update your CSS to use a [guide background image](#):

```
1 html {  
2   font-family: Cambria, Georgia, "Times New Roman",  
   Times, serif;  
3   background: url(.../img/css/grid-01.png) center top  
   repeat-y #fff;  
4   color: #333;  
5   width: 940px;  
6   padding: 0 10px;  
7   margin: 0 auto;  
8 }
```

You should now have something like this:



Choosing A Measure

We have our font size, so now we need to work out our ideal line length, or “measure.” Robert Bringhurst writes in *The Elements of Typographic Style* that, “anything from 45 to 75 characters is widely regarded as a satisfactory length of line...”

A measure that is too short causes the eye to jump awkwardly from the end of line x to the start of line $x + 1$, and a measure that’s too long can cause the reader’s eye to double back. You can circumvent this somewhat by following these rules of thumb:

- for a longer measure, use slightly greater leading
- for a shorter measure, use slightly smaller leading

So, a measure of 45 to 75 characters is the optimum for readability in columns of text. I can pretty much guarantee that after you learn this, every massively, overly long measure you see on the Web will annoy you spectacularly.

Here are 69 characters, a nice middle ground between the recommended 45 and 75:

```
1 | Lorem ipsum dolor sit amet, consectetur adipiscing  
  | elit accumsan
```

Paste that into your page, and count how many red columns it covers. This is how wide your measure will be:



Here we have text spanning eight columns, which is 460 pixels of 960.gs. Update your CSS to read as follows:

```
1 /*-----*\
2 MAIN
3 \*-----*/
4
5 html {
6   font-family: Cambria, Georgia, "Times New Roman",
   Times, serif;
7   background: url(.../img/css/grid-01.png) center top
   repeat-y #fff;
8   color: #333;
9 }
10
11 body {
```

```
12 | width: 460px;  
13 | margin: 0 auto;  
14 | }
```



If you picked a font size other than 16 pixels, make sure your measurements reflect this!

Vertical Rhythm: Setting A Baseline

Leading (which rhymes with “wedding”) is a typographical term from way back when typographers manually set type in letterpress machines and the like. The term refers to the act of placing lead between lines of type in order to add vertical space. In CSS, we call this `line-height`.

Line height should be around 140%. This isn't a great number to work with, and it's only a general rule, so we'll use 150% (or 1.5 em). This way, we simply need to multiply the font size by one and a half to determine our leading.

Some general rules for leading:

- with continuous copy, use large leading
- with light text on dark background, use large leading
- with long line lengths, use large leading
- with large x-height, use large leading
- with short burst of information, use small leading

If you used a 16-pixel font size, then your line height will be 24 pixels ($16\text{ pixels} \times 1.5\text{ em} = 24\text{ pixels}$). If you used a 12-pixel font size, then your line height will be 18 pixels ($12\text{ pixels} \times 1.5\text{ em} = 18\text{ pixels}$).

The Magic Number

For math-based tips on typography, check out [this video on Web type](#) by Tim Brown. The fun starts at 13:35.

The pixel value for your line height (24 pixels) will now be your magic number. This number means *everything* to your design. All line heights and margins will be this number or multiples thereof. I find it useful to always keep it in mind and stick to it.

Now that we know our general line height, we can define a baseline grid. The grid we currently have aligns only the elements in the y axis (up and down). A baseline grid aligns in the x axis, too. We need to update our

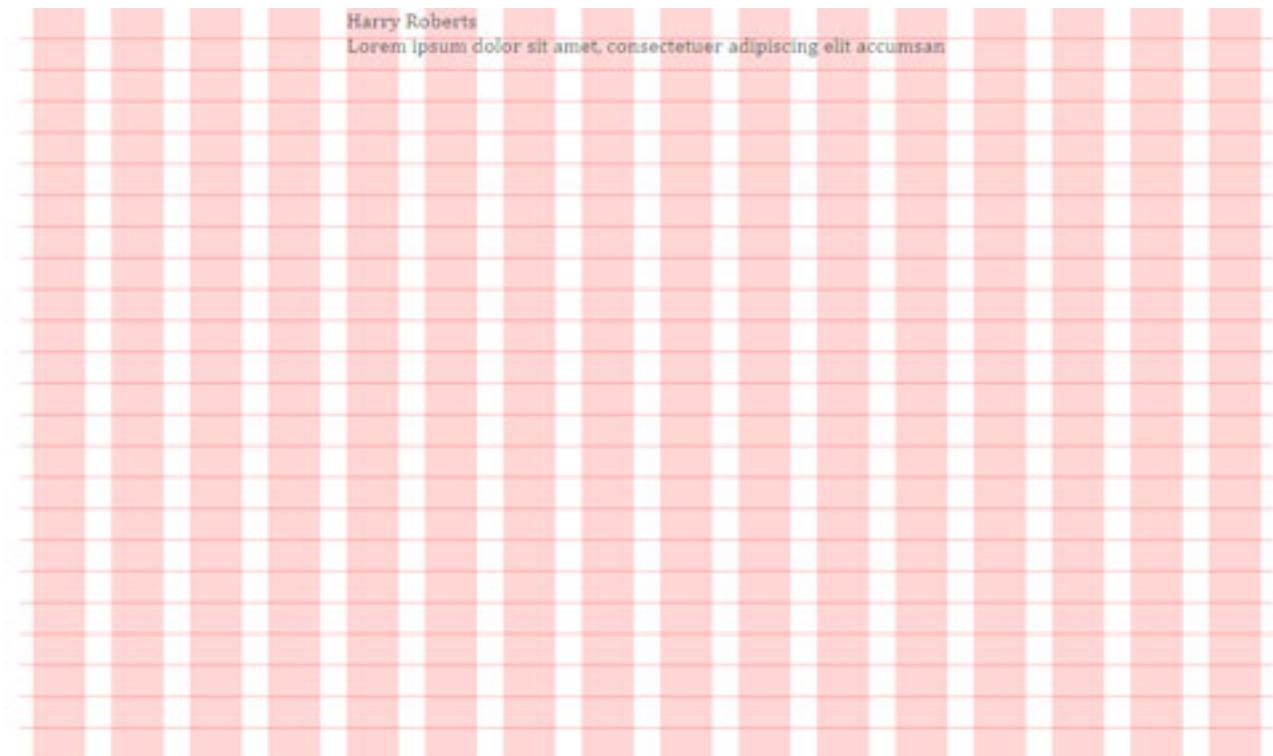
background image now to be 24 pixels high and have a solid 1-pixel line at the bottom, like [this](#).

Again, if you chose a font size of 12 pixels and your line height became 18 pixels, then your background image needs to be 18 pixels high with a solid horizontal line on the 18th row of pixels.

Your CSS should now look something like this:

```
1 html {  
2  
3 }  
4  
5 body {  
6   width: 460px;  
7   margin: 0 auto;  
8   line-height: 1.5em;  
9 }
```

Your page should now look something like this:



As you can see, the text hovers slightly above the horizontal guideline. This doesn't mean that anything is set incorrectly; it is merely the offset. This could hinder the process, so either tweak the padding on the `body` to move the page or alter the position of the background image to move it around a little. Some tinkering in Firebug tells me that the CSS needs to be as follows:

```
1 html {  
2  
3   background: url(.../img/css/grid.png) center -6px  
   repeat-y #fff;  
4  
5 }
```

That gives me the following — and everything lines up perfectly:

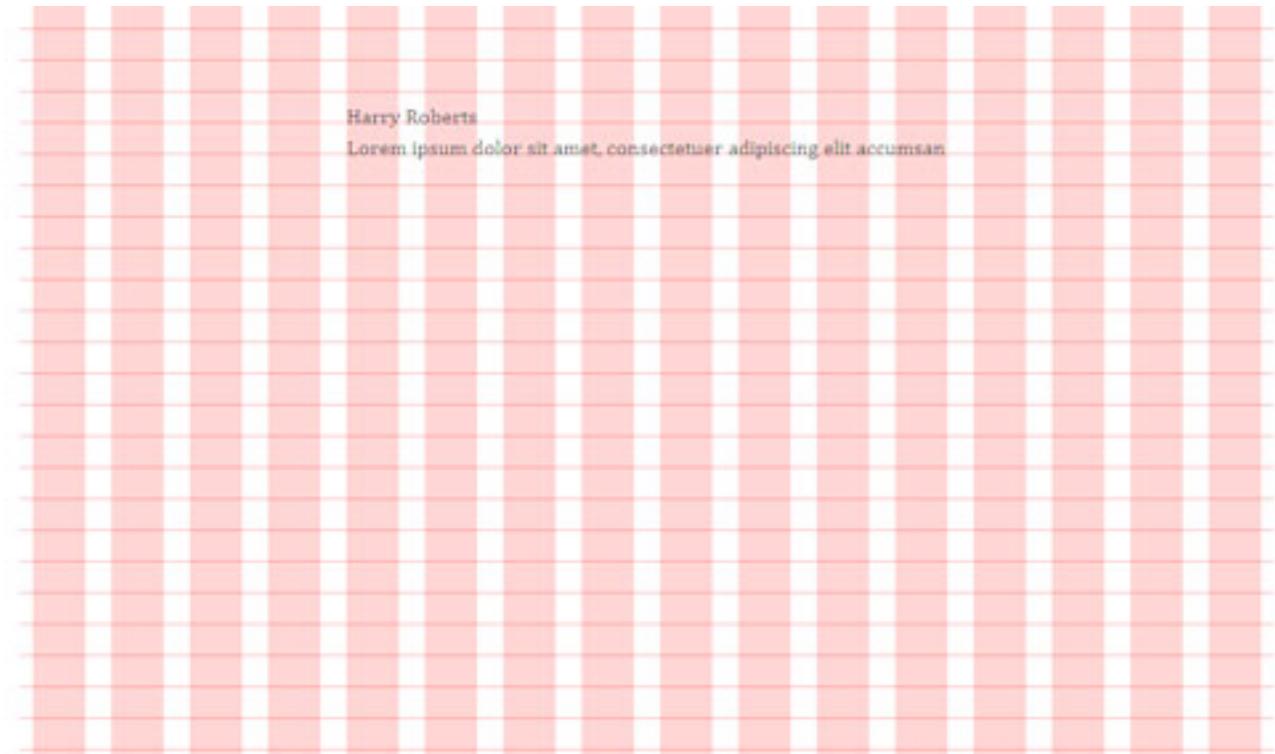


Now, let's get back to the magic number. Maybe you think the text is sitting too close to the top of the screen? Well, to remedy that, we'll move the text down the page by a multiple of that magic number — let's say 72 ($3 \times 24 = 72$ pixels). Now adjust your CSS to read:

```
1 body {  
2   width: 460px;  
3   margin: 0 auto;  
4   line-height: 1.5em;  
5   padding-top: 72px;  
6 }
```

Substitute your own magic number if you used a different font size.

We should get this:



It took some doing, but our canvas is ready at last!

Setting A Scale

Okay, our reset has made our h1 and p the same size. We need to get some basic font styles in there. Add this block of code to the end of your CSS:

```
1 /*-----*\
2  TYPE
3  \*-----*/
4 /*--- HEADINGS ---*/
5
6 h1, h2, h3, h4, h5, h6 {
7   margin-bottom: 24px;
8   font-weight: bold;
```

```
9 }
10
11 /*--- PARAGRAPHS ---*/
12
13 p {
14   margin-bottom: 24px;
15 }
```

Recognize your magic number? Let's refresh the page and take a look:



Your magic number will now be the default `margin-bottom` value for all of your elements. This, combined with the line height, will keep everything on the baseline grid.

What we now need, though, are some different font sizes for the headings. We need a typographic scale. I suggest this:

- h1 = 24 pixels
- h2 = 22 pixels
- h3 = 20 pixels
- h4 = 18 pixels
- h5 = 16 pixels
- h6 = 16 pixels

Many people work in pixels, but I much prefer working in ems. An em is proportional to the current size of the font: 1 em in 72-point Georgia is 72 points, and 1 em in 8-point Garamond is 8 points.

So, if our base font size is 16 pixels (1 em), then 24 pixels would be 1.5 ems ($24 \div 16 = 1.5$). If we continue, we end up with:

- h1 = 24 pixels $\rightarrow 24 \div 16 =$ **1.5 ems**
- h2 = 22 pixels $\rightarrow 22 \div 16 =$ **1.375 ems**
- h3 = 20 pixels $\rightarrow 20 \div 16 =$ **1.25 ems**
- h4 = 18 pixels $\rightarrow 18 \div 16 =$ **1.125 ems**
- h5 = 16 pixels $\rightarrow 16 \div 16 =$ **1 ems**
- h6 = 16 pixels $\rightarrow 16 \div 16 =$ **1 ems**

Next, we need to make sure the line height of each is 24 pixels. This means that the h1 at a 24-point font size will have a line height of 1 em. Here's the math:

$$(magic\ number) \div (font\ size) = (line\ height)$$

Using our scale, the full CSS for the headings (including the math) is:

```
1 /*--- HEADINGS ---*/
2 h1, h2, h3, h4, h5, h6 {
3   margin-bottom: 24px;
4   font-weight: bold;
5 }
6
7 h1 {
8   font-size: 1.5em; /* 24px --> 24 ÷ 16 = 1.5 */
9   line-height: 1em; /* 24px --> 24 ÷ 24 = 1 */
10 }
11
12 h2 {
13   font-size: 1.375em; /* 22px --> 22 ÷ 16 = 1.375 */
14   line-height: 1.0909em; /* 24px --> 24 ÷ 22 =
15   1.090909(09) */
16 }
17 h3 {
18   font-size: 1.25em; /* 20px --> 20 ÷ 16 = 1.25 */
19   line-height: 1.2em; /* 24px --> 24 ÷ 20 = 1.2 */
20 }
21
22 h4 {
```

```
23 font-size: 1.125em; /* 18px --> 18 ÷ 16 = 1.125 */
24 line-height: 1.333em; /* 24px --> 24 ÷ 18 =
1.3333333(3) */
25 }
26
27 h5, h6 {
28 font-size: 1em; /* 16px --> 16 ÷ 16 = 1 */
29 line-height: 1.5em; /* 24px --> 24 ÷ 16 = 1.5 */
30 }
```

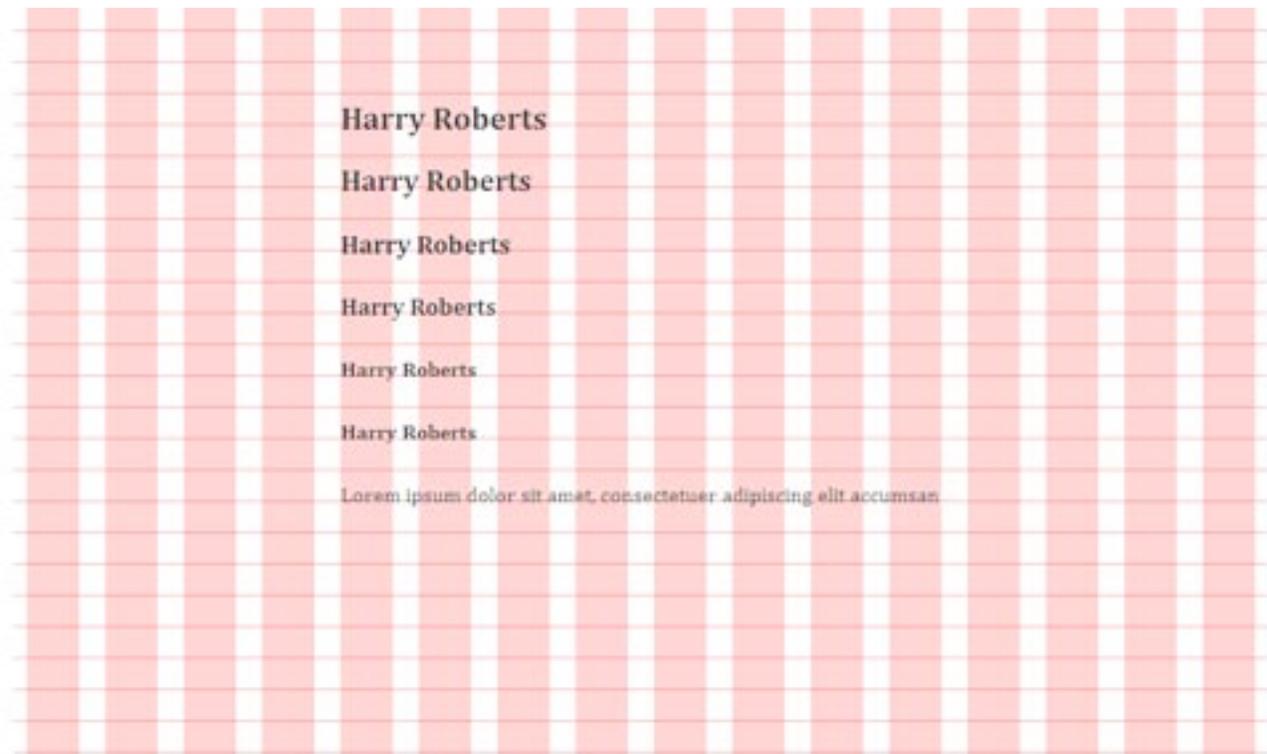
There's our typographic scale.

Now, to test it, let's try the following markup:

```
1 <body>
2
3 <h1>Your Name</h1>
4
5 <h2>Your Name</h2>
6
7 <h3>Your Name</h3>
8
9 <h4>Your Name</h4>
10
11 <h5>Your Name</h5>
12
13 <h6>Your Name</h6>
14
15 <p>Lorem ipsum dolor sit amet, consectetur
adipiscing elit accumsan</p>
```

```
16 |  
17 | </body>
```

You might notice that not all of the lines of text sit perfectly on a gridline, but that's okay because they all honor the baseline! This is what I get:



You might think that something has gone wrong. But if you look, the paragraph lies just fine once you get back to the normal font size. To be honest, I'm not entirely sure about what causes this effect; the numbers we used are all correct, and the vertical rhythm as a whole remains intact, but individual lines of larger text appear to be offset from the baseline. I think this could be due, in part, to the glyphs' setting in their em box.

What Next?

Head back into your markup and remove everything except the `h1`. Now we're ready to do something useful. Let's make a little "About you"-page.

The `h1` is the name. And the markup can simply be:

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8" />
5   <title>Harry Roberts</title>
6   <link rel="stylesheet" type="text/css" href="css/
  style.css" />
7 </head>
8
9 <body>
10
11 <h1>Harry Roberts</h1>
12
13 </body>
14 </html>
```

Now let's add a little introductory paragraph about yourself. Mine reads:

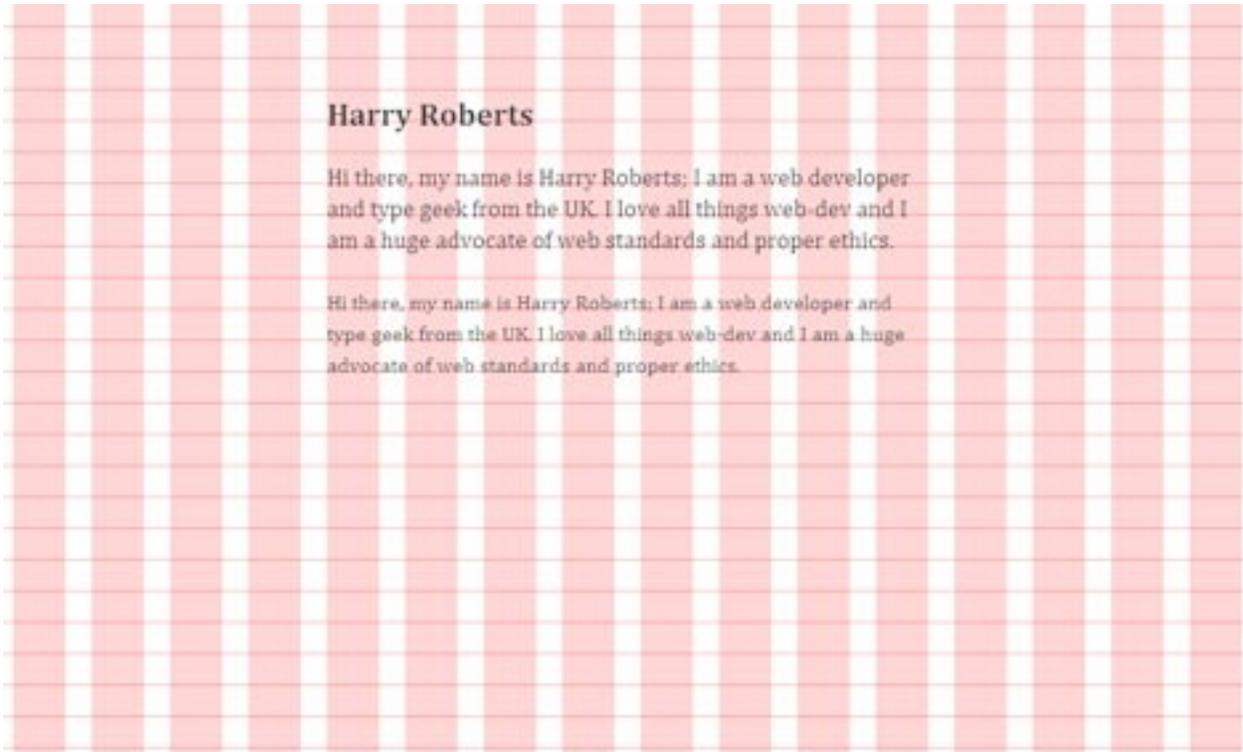
```
1 <p>Hi there. My name is Harry Roberts.
2 I am a Web developer and type geek from the UK.
3 I love all things Web dev, and I am a huge advocate
4 of Web standards and proper ethics.</p>
```

Let's experiment with altering the font size arbitrarily. Add this to your CSS:

```
1 *--- PARAGRAPHS ---*/
2 p {
3   margin-bottom: 24px;
4 }
5
6 body > p:first-of-type {
7   font-size: 1.125em;
8   /* 18px → 18 ÷ 16 = 1.125 */
9
10  line-height: 1.333em;
11  /* 24px → 24 ÷ 18 = 1.3333(3) */
12 }
```

Here we're giving the first paragraph — a direct descendant of the `body` element — a font size of 18 pixels and a line height of 24 pixels. See, there's your magic number again!

You might again see slight oddities with the paragraph sitting on the baseline. To make sure the vertical rhythm is still intact, duplicate the paragraph once more. You should get this:



Here you can see that the vertical rhythm is still intact and working correctly.

Now for the best bits.

Tips on Technical Typography

There's a good chance that you won't want the grid to always be on, so change this CSS:

```
1 | /*-----*\
2 | MAIN
3 | \*-----*/
4 |
5 | html {
```

```
6  font-family: Cambria, Georgia, "Times New Roman",
   Times, serif;
7  background: url(.../img/css/grid.png) center -6px
   repeat-y #fff;
8  color: #333;
9  }
10
11 body {
12  width: 460px;
13  margin: 0 auto;
14  line-height: 1.5em;
15  padding-top: 72px;
16 }
```

... to this:

```
1 /*-----*\
2  MAIN
3  \*-----*/
4
5 html {
6  font-family: Cambria, Georgia, "Times New Roman",
  Times, serif;
7  color: #333;
8 }
9
10 body {
11  width: 460px;
12  margin: 0 auto;
13  line-height: 1.5em;
14  padding-top: 72px;
15  background: #fff;
16 }
17
18 body:hover {
19  background: url(.../img/css/grid.png) center -6px
  repeat-y #fff;
20 }
```

This will show the grid when you hover over the `body`, and hide it when you don't.

Spacing And Setting Paragraphs

We've sorted out the magic number, and we know we should use it to space the elements, but there's more than one way to denote the beginning of a new paragraph. One is the method we're already using: inserting a blank space (one magic number) between the paragraphs. The second is indentation.

Typically, you would indent every paragraph except the first in a body of text; the first paragraph has no indent and the second, third, fourth and so on do have an indent (typically with a width of 1 em).

Enric Jardi writes in *Twenty-Two Tips on Typography* that, "... you must not use both indentation and a line break at the same time; that is redundant."

Here's some quick CSS for indenting only the second and subsequent paragraphs in a body of text:

```
1 p {  
2   margin-bottom: 24px;  
3 }  
4  
5 p+p {  
6   text-indent: 1em;  
7   margin-top: -24px;  
8 }
```

For an explanation of how and why this works, refer to my other article, "[Mo' Robust Paragraph Indenting](#)." You might also want to look at [Jon Tan's silo](#).

Alignment

When setting type on the Web, use a range-left, ragged-right style. This basically means left-justifying the type. If you use a sufficiently long measure, then your rags (the uneven edges on the right side of a left-aligned paragraph) will generally be clean; the raggedness of their appearance can, however, be exacerbated at short measures, where a large percentage of each line could end up as white space.

Justified typesetting can look great but lead to unsightly “rivers” in the text. To avoid this, rewrite the copy to remove them, or use something like [Hyphenator.js](#), which is remarkably effective.

Proper Quotations Marks, Dashes And Ellipses

Quotation Marks

Many people are unaware that there are proper quotation marks and “ambidextrous” quotation marks. The single- and double-quotation keys on your keyboard are not, in fact, true quotation marks. They are catch-alls that can function as both left and right single and double quotation marks; they are, essentially, four glyphs in one key.

The reason behind this is simply space. A keyboard cannot feasibly fit proper left and right single and double quotation marks.

So, what is a proper quotation mark? A curly (or “book”) quotation mark is rounded and more angular than an ambidextrous (keyboard-style) quotation mark. Left single and left double quotation marks look like this: ‘ and “ (&lquo; and “, respectively). Right single and right double

quotation marks look like this: ' and " (’ and ”, respectively).

Many people incorrectly refer to ambidextrous quotation marks as “primes,” but a prime is a different glyph. Single and double primes look like this: ' and " (′ and ″, respectively). They are used to denote feet and inches (e.g. 5'10").

I said that one key incorporates four glyphs. In fact, two keys incorporate six glyphs.

Which Quotation Marks Should You Use?

The type of quotation marks to use (double or single) varies from country to country and style guide to style guide. Double quotation marks are typically used to quote a written or spoken source, and single quotation marks are used for quotes within quotes.

However, I much prefer Jost Hochuli's advice in [Detail in Typography](#): "... the appearance is improved by using the more obtrusive double quotation marks for the less frequent quotations within quotations." Which basically means, for a nicer appearance, use single quotation marks, and then double quotation marks for quotes within quotes. (If I had a penny for every time I said *quotes* in this section.)

For example:

'And then he said to me, "Do you like typography?" And naturally I said that I did.'

Use a right single quotation mark where you'd normally use an apostrophe in text: "I'm a massive typography nerd!" (I’m a massive typography nerd!)

In short, stop using those horrible keyboard quotation marks, and start using lovely curly marks in your work.

Non-English Quotation Marks

The quotation marks we've covered are used in English, but quotes look different in other languages.

French and Spanish use *guillemets*, «like this» («like this»). In Danish, they are used »like this« (»like this«). In German, using a combination of bottom and regular double quotation marks is common, „like this“ („like this“).

For a great overview of other non-English quotation marks, see the Wikipedia entry on "[Non-English Usage of Quotation Marks](#)."

Dashes

We know that keyboards can't accommodate all quotation marks; and they can't accommodate all types of dashes either. The hyphen key (-) is another catch-all. There are three main types of dash: the em dash, en dash and hyphen.

The em dash (—) denotes a break in thought—like this. It's called the "em" dash because, traditionally, it is one em wide. Em dashes are usually set without spaces on either side (as above).

The en dash (–) is traditionally half the width of an em dash. It is used to denote ranges, as in “please read pages 17–25” (17–25). It can also denote relational phrases, as in “father–son” or “New York–London.”

The hyphen simply ties together compound words, as in “front-end developer.”

The em dash, en dash and hyphen are different, and each has unique uses.

Ellipsis

An ellipsis is used to denote a thought trailing off. It is also used as a placeholder for omitted text in lengthy quotations.

The ellipsis has become the bane of my life. I often come across people who use a series of dots (periods) in place of a proper ellipsis, like so.....

An ellipsis is not three dots. It is one glyph that looks like three dots. Its HTML entity is `…` (as in horizontal ellipsis).

So there were a few glyphs for you to use with quotes, primes, dashes and ellipses. Let’s recap:

Name/Glyph	HTML entity	Example
Quotes and primes		
Left single quote ` and right single quote `	<code>&lsquo;</code> ; and <code>&rsquo;</code> ;	'Hey, this is a quote!'
Left double quote " and right double quote "	<code>&ldquo;</code> ; and <code>&rdquo;</code> ;	'Hey, this is a quote "within another" quote!'
Single prime ' and double prime "	<code>&prime;</code> ; and <code>&Prime;</code> ;	The girl is 7'10"!
Dashes		
Em dash —	<code>&mdash;</code> ;	A break in thought—like this
En dash –	<code>&ndash;</code> ;	Ages 2–5
Hyphen -	- key	front-end developer
Ellipsis		
Ellipsis ...	<code>&hellip;</code> ;	To be continued...

In addition to these common glyphs, there are numerous others: from the division symbol (\div or `÷`) to the section symbol (§ or `§`). If you're interested in special characters and glyphs, then Wikipedia's article on "[Punctuation](#)" is a good place to start (just keep clicking from there).

Hanging Punctuation

Punctuation should be hung; quotation marks and bullet points should be clear of the edges of surrounding text. If that doesn't make sense, don't worry! Let's add a new section to your page. Remove that duplicated paragraph and replace it with a list of facts about yourself. Mine looks like this:

```
1 <ul>
2 <li>
3   Web development
4 <ul>
5   <li>HTML (5)</li>
6   <li>CSS (3)</li>
7   <li>Accessibility</li>
8   <li>Progressive enhancement</li>
9 </ul>
10 </li>
11 <li>
12   Web design
13 <ul>
14   <li>Typography</li>
15   <li>Grids</li>
16 </ul>
17 </li>
18 </ul>
```

Add this to the end of your CSS sheet:

```
1 /*--- LISTS ---*/
2 ul, ol {
3   margin-bottom: 24px;
4   /* Remember, if your magic number is
5    different to this, use your own. */
6 }
7
8 ul {
9   list-style: square outside;
10 }
```

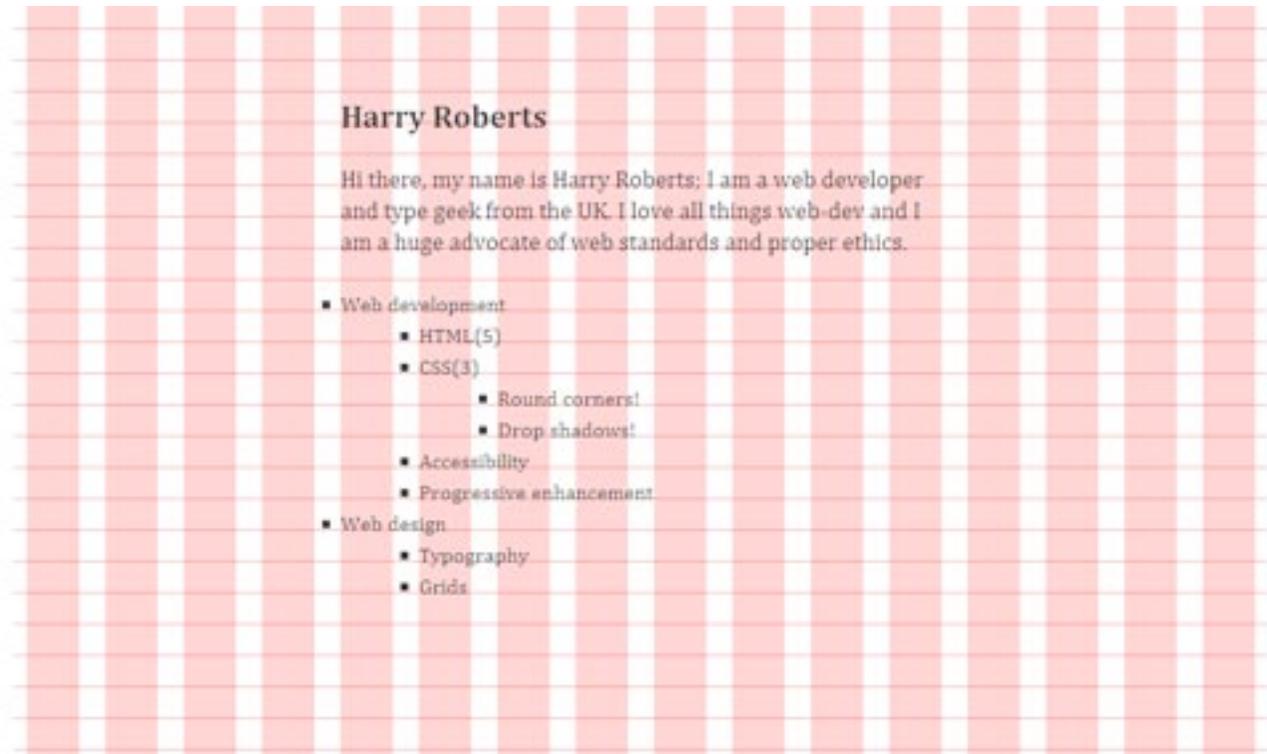
```
11 |
12 | ul ul,
13 | ol ol {
14 |   margin: 0 0 0 60px;
15 | }
```

My page now looks like this:



We've given the lists our magic number as a margin, set the bullets to be hung outside of the text (i.e. the bullets will sit in the white of the gutter, not the pink of the column) and indented lists within lists by one grid column.

Experiment by nesting lists more and more deeply:



Hang quotation marks as well as bullets. Let's add some more text and a quote to our page:

```
1 <p>Vestibulum adipiscing lectus ut risus adipiscing  
2 mattis sed ac lectus. Cras pharetra lorem eget diam  
3 consectetur sit amet congue nunc consequat. Curabitur  
4 consectetur ullamcorper varius. Nulla sit amet sem ac  
5 velit auctor aliquet. Quisque nec arcu non nulla  
6 adipiscing  
7 rhoncus ut nec lorem. Vestibulum non ipsum arcu.  
8 Quisque  
9 dapibus orci vitae massa fringilla sit amet viverra  
10 nulla.</p>  
11  
12 <blockquote>
```

```
10 |
11 | <p>&ldquo;Lorem ipsum dolor sit amet,
12 | consectetur adipiscing elit. In accumsan diam
13 | vitae velit. Aliquam vehicula, turpis sed egestas
14 | porttitor, est ligula egestas leo, at interdum
15 | leo ante ut risus.&rdquo;
16 | <b>&mdash;Joe Bloggs</b></p>
17 |
18 | </blockquote>
```

The markup here is pretty straightforward: a `blockquote` surrounding a paragraph. You might also notice the use of a `b` element to mark up the name. The [HTML spec](#) states that “The `b` element [is used for] spans of text whose typical typographic presentation is boldened.” This is a loose definition, so its use for bolding the name of a person is acceptable.

Now, add this to the end of your CSS sheet:

```
1 | /*--- QUOTES ---*/
2 | blockquote {
3 |   margin: 0 60px 0 45px;
4 |   border-left: 5px solid #ccc;
5 |   padding-left: 10px;
6 |   text-indent: -0.4em;
7 | }
8 |
9 | blockquote b {
10 |   display: block;
11 |   text-indent: 0;
12 | }
```

Here we indent the quote by 60 pixels from the left and right (i.e. 45-pixel margin + 5-pixel border + 10-pixel padding = 60 pixels), taking it in by one column of the grid. We then use a negative `text-indent` to make the opening quote hang outside of the body of text. The number I used works perfectly for Cambria, but you can experiment with the font of your choice. (Don't forget to remove the `text-indent` on the b.) Now we know how to hang bullets and quotation marks.



Maybe you're wondering why I'm using double quotation marks here after recommending single quotation marks. The reason is purely aesthetic. Hanging double quotation marks in `blockquote` tags simply looks nicer.

Guillemets

Now that we've done that, let's add a "Read more" link to get us from this little page to, say, our portfolio's full "About" page. We want to imply direction or movement with this link because it's going to take us elsewhere. We could, as many people do, use a [guillemet](#) (`>`, `»`), but — as we covered earlier — French, German and [other languages](#) use this glyph as a quotation mark. Therefore, it shouldn't be used stylistically. Add this markup to your page:

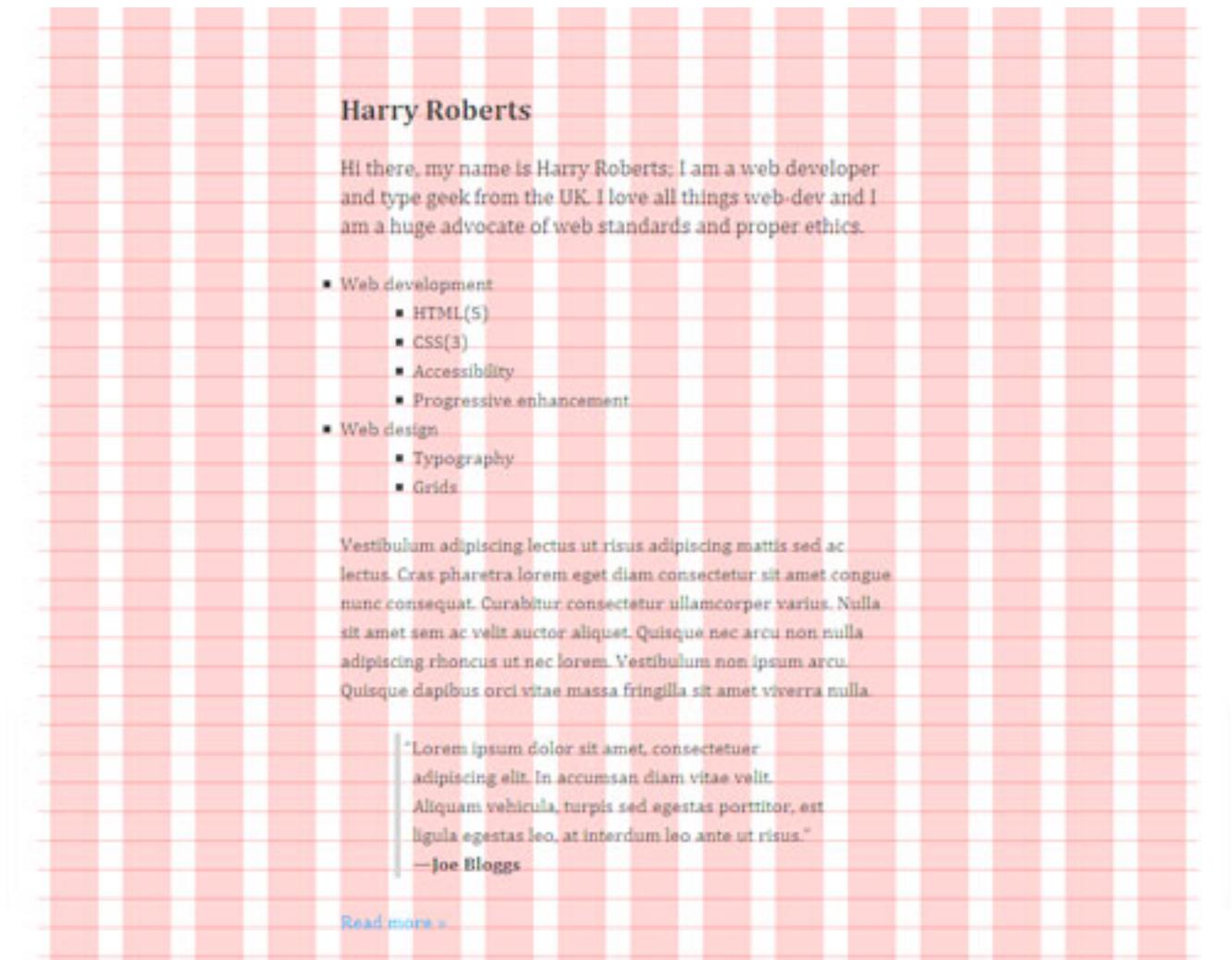
```
1 | <p><a href="http://csswizardry.com/about/ "  
2 | class="read-more">Read more</a></p>
```

Add this to the end of your CSS file:

```
1 | /*--- LINKS ---*/  
2 | a {  
3 |   color: #09f;  
4 |   text-decoration: none;  
5 | }  
6 |  
7 | a:hover {  
8 |   text-decoration: underline;  
9 | }  
10 |  
11 | a:active,  
12 | a:focus {
```

```
13 position: relative;
14 top: 1px;
15 }
16
17 .read-more:after {
18 content: "\00A0\00BB"; /* Insert a space then right
   angled-quote */
19 }
```

This simply places [an encoded space and right-angled quotation mark](#) after the “Read more” link by using CSS, which means you don’t have to add that quotation mark to your markup.



You can use `content: " " ;` to [keep the markup clean](#). This means that other things, such as stylistic right-angled quotation marks and other decorative items of type, can be added with CSS to keep the markup free of non-semantic elements.

Let's say you wanted to add tildes to either side of a heading:

```
1 h1 {  
2   text-align: center;  
3 }  
4 h1:before {
```

```
5 content: "\007E\00A0"; /* Insert an tilde, then a
   space. */
6 }
7 h1:after {
8 content: "\00A0\007E"; /* Insert a space, then an
   tilde. */
9 }
```

Some Images

Elements such as tables and images are notoriously difficult to fit into baseline grids unless you save every one as a multiple of your magic number. However, we can float images left and right within the y axis of the grid and allow text to fit the baseline around it. Grab an image of yourself (or your cat or whatever) and crop it to a width that fits our [16-column 960.gs](#). I'm going to use a 160-pixel-wide image (i.e. three grid columns).

Place it in the markup just after your `h1`, thus:

```
1 ...
2 <body>
3
4 <h1>Harry Roberts</h1>
5
6 
```

If you hit “Refresh” now, it will likely break the baseline. Never fear — add this CSS:

```
1 /*-----*\
2 IMAGES
3 \*-----*/
4
5 #me {
6   float: right;
7   margin: 0 0 0 20px;
8   display: block;
9   width: 148px;
10  height: auto;
11  padding: 5px;
12  border: 1px solid #ccc;
13
14  -moz-border-radius: 2px;
15  -webkit-border-radius: 2px;
16  -o-border-radius: 2px;
17  border-radius: 2px;
18 }
```

Notice how the image doesn't appear to sit on the baseline, but the rest of the text does? Beautiful, isn't it?

So, there you have it. Using nothing more than plain ol' browser text, one image, a lot of typography knowledge and some careful, caring attention, you've made a full page of typographic splendor — a page that uses a grid, a baseline, proper punctuation and glyphs, an ideal measure and leading and a typographic scale.

Now get in there! Add elements, subtract them, add colors, add your own creative type. Just remember the few simple rules and your magic number, and you're away!

Harry Roberts

Hi there, my name is Harry Roberts. I am a web developer and type geek from the UK. I love all things web-dev and I am a huge advocate of web standards and proper ethics.



- Web development
 - HTML(5)
 - CSS(3)
 - Accessibility
 - Progressive enhancement
- Web design
 - Typography
 - Grids

Vestibulum adipiscing lectus ut risus adipiscing mattis sed ac lectus. Cras pharetra lorem eget diam consectetur sit amet congue nunc consequat. Curabitur consectetur ullamcorper varius. Nulla sit amet sem ac velit auctor aliquet. Quisque nec arcu non nulla adipiscing rhoncus ut nec lorem. Vestibulum non ipsum arcu. Quisque dapibus orci vitae massa fringilla sit amet viverra nulla.

“Lorem ipsum dolor sit amet, consectetur adipiscing elit. In accumsan diam vitae velit. Aliquam vehicula, turpis sed egestas porttitor, est ligula egestas leo, at interdum leo ante ut risus.”
— Joe Bloggs

‘I’d like you to—if you would—correct all the “typographical mistakes” in this paragraph. It should take 1-5 minutes...’

[Read more >](#)

The final piece, with the grid.

Harry Roberts

Hi there, my name is Harry Roberts; I am a web developer and type geek from the UK. I love all things web-dev and I am a huge advocate of web standards and proper ethics.



- Web development
 - HTML(5)
 - CSS(3)
 - Accessibility
 - Progressive enhancement
- Web design
 - Typography
 - Grids

Vestibulum adipiscing lectus ut risus adipiscing mattis sed ac lectus. Cras pharetra lorem eget diam consectetur sit amet congue nunc consequat. Curabitur consectetur ullamcorper varius. Nulla sit amet sem ac velit auctor aliquet. Quisque nec arcu non nulla adipiscing rhoncus ut nec lorem. Vestibulum non ipsum arcu. Quisque dapibus orci vitae massa fringilla sit amet viverra nulla.

“Lorem ipsum dolor sit amet, consectetur adipiscing elit. In accumsan diam vitae velit. Aliquam vehicula, turpis sed egestas porttitor, est ligula egestas leo, at interdum leo ante ut risus.”
—Joe Bloggs

‘I’d like you to—if you would—correct all the “typographical mistakes” in this paragraph. It should take 1-5 minutes...’

[Read more >](#)

The final piece, without the grid.

Final Words

In this admittedly long piece, we have discussed only *technical* typography. Everything in this article can be applied to almost any project. None of it was speculation; these are tried and tested methods and proven tips for beautiful Web type.

If you'd like to see more creative applications of Web type, then check out the work of the following creatives (each of whom has had much influence on my career so far):

- Oliver Reichenstein of [Information Architects](#)
A huge inspiration to me and a very knowledgeable guy who has a passion and talent for readable, sensible and beautiful type on the Web.
- [Jon Tan](#)
Jon's website is gorgeous. He is a member of the International Society of Typographic Designers (ISTD), and his writings and "silo" (on his personal website) are a hive of typographical information.
- [Jos Buivenga](#)
Not strictly a Web-type guy, but Jos is the creator of some of the most beautiful (and free!) fonts in existence. His work got me hooked on typography.
- [Khoi Vinh](#)
His timelessly beautiful website spurred my love for grids. He also recently wrote [a book](#) on that very subject.

Keep in mind that you don't have to be the world's best designer to create beautiful type. You just have to *care*.

The Future of CSS Typography

Inayaili de Leon

There has been an increasing and sincere interest in typography on the web over the last few years. Most websites rely on text to convey their messages, so it's not a surprise that text is treated with utmost care. In this article, we'll look at some useful techniques and clever effects that use the power of style sheets and some features of the upcoming CSS Text Level 3 specification, which should give Web designers finer control over text.

Keep in mind that these new properties and techniques are either new or still in the works, and some of the most popular browsers do not yet support them. But we feel it's important that you, as an informed and curious Web designer, know what's around the corner and are able to experiment in your projects.

A Glance At The Basics

One of the most common CSS-related mistakes made by budding Web designers is creating inflexible style sheets that have too many classes and IDs and that are difficult to maintain.

Let's say you want to change the color of the headings in your posts, keeping the other headings on your website in the default color. Rather than add the class `big-red` to each heading, the sensible approach would be to take advantage of the `DIV` class that wraps your posts (probably `post`) and create a selector that targets the heading you wish to modify, like so:

```
1 | .post h2 {  
2 |   font-weight: bold;  
3 |   color: red;  
4 | }
```

This is just a quick reminder that there is no need to add classes to everything you want to style with CSS, especially text. Think simple.

The Font Property

Instead of specifying each property separately, you can do it all in one go using the font shorthand property. The order of the properties should be as follows: `font-style`, `font-variant`, `font-weight`, `font-size`, `line-height`, `font-family`.

When using the font shorthand, any values not specified will be replaced by their parent value. For example, if you define only `12px Helvetica, Arial, sans-serif`, then the values for `font-style`, `font-variant` and `font-weight` will be set as `normal`.

The font property can also be used to specify system fonts: `caption`, `icon`, `menu`, `message-box`, `small-caption`, `status-bar`. These values will be based on the system in use, and so will vary according to the user's preferences.

Other Font Properties

A few font-related properties and values are not as commonly used. For example, instead of using `text-transform` to turn your text into all caps, you could use `font-variant: small-caps` for a more elegant effect.

You could also be very specific about the [weight of your fonts](#), instead of using the common `regular` and `bold` properties. CSS allows you to specify font weight with values from 100 to 900 (i.e. 100, 200, 300, etc.). If you decide to use these, know that the 400 value represents the `normal` weight, while 700 represents `bold`. If a font isn't given a weight, it will default to its parent weight.

Another useful property, sadly supported only in Firefox for now, is `font-size-adjust`, which allows you to specify an aspect ratio for when a fall-back font is called. This way, if the substitute font is smaller than the preferred one, the text's x-height will be preserved. A good explanation of how `font-size-adjust` works can be found on [the W3C website](#).

Dealing With White Space, Line Breaks And Text Wrapping

Several CSS properties deal with these issues, but the specs are still in the works (at the "Working Draft" stage).

White Space

The `white-space` property lets you specify a combination of properties for which it serves as a shorthand: [white-space-collapsing](#) and [text-wrap](#). Here's a breakdown of what each property stands for:

- `normal`
white-space-collapsing: collapse/text-wrap: normal
- `pre`
white-space-collapsing: preserve/text-wrap: none

- `nowrap`
`white-space-collapsing: collapse/text-wrap: none`
- `pre-wrap`
`white-space-collapsing: preserve/text-wrap: normal`
- `pre-line`
`white-space-collapsing: preserve-breaks/text-wrap: normal`

This property can be useful if you want to, for example, display snippets of code on your website and preserve line breaks and spaces. Setting the container to `white-space: pre` will preserve the formatting.



The screenshot shows the WordPress dashboard with a table titled "Right Now" under the heading "At a Glance". The table contains four rows of statistics:

Right Now	
<i>At a Glance</i>	
1,798 Posts	108,01
30 Pages	107,29
13 Categories	
414 Tags	7.

[WordPress](#) uses `white-space: nowrap` on its dashboard so that the numbers indicating posts and comments don't wrap if the table cell is too small.

Word Wrap

One property that is already well used is `word-wrap`. It supports one of two values: `normal` and `break-word`. If you set `word-wrap` to `break-word` and a word is so long that it would overflow the container, it is broken at a random point so that it wraps within the container.



The [International Gorilla Conservation Programme](#) website uses *word-wrap* for its commenters' names.

In theory, *word-wrap*: *break-word* should only be allowed when *text-wrap* is set to either *normal* or *suppress* (which suppresses line breaking). But in practice and for now, it works even when *text-wrap* is set to something else.

Bear in mind that according to the specification, the *break-strict* value for the *word-break* property is at risk of being dropped.

Word And Letter Spacing

Two other properties that are often used are *word-spacing* and *letter-spacing*. You can use them to control—you guessed it—the spacing between words and letters, respectively. Both properties support three different values that represent optimal, minimum and maximum spacing.



[Show & Tell](#) uses *letter-spacing* on its navigation links.

For *word-spacing*, setting only one value corresponds to the optimal spacing (and the other two are set to `normal`). When setting two values, the first one corresponds to the optimal and minimum spacing, and the second to the maximum. Finally, if you set all three values, they correspond to all three mentioned above. With no justification, optimal spacing is used.

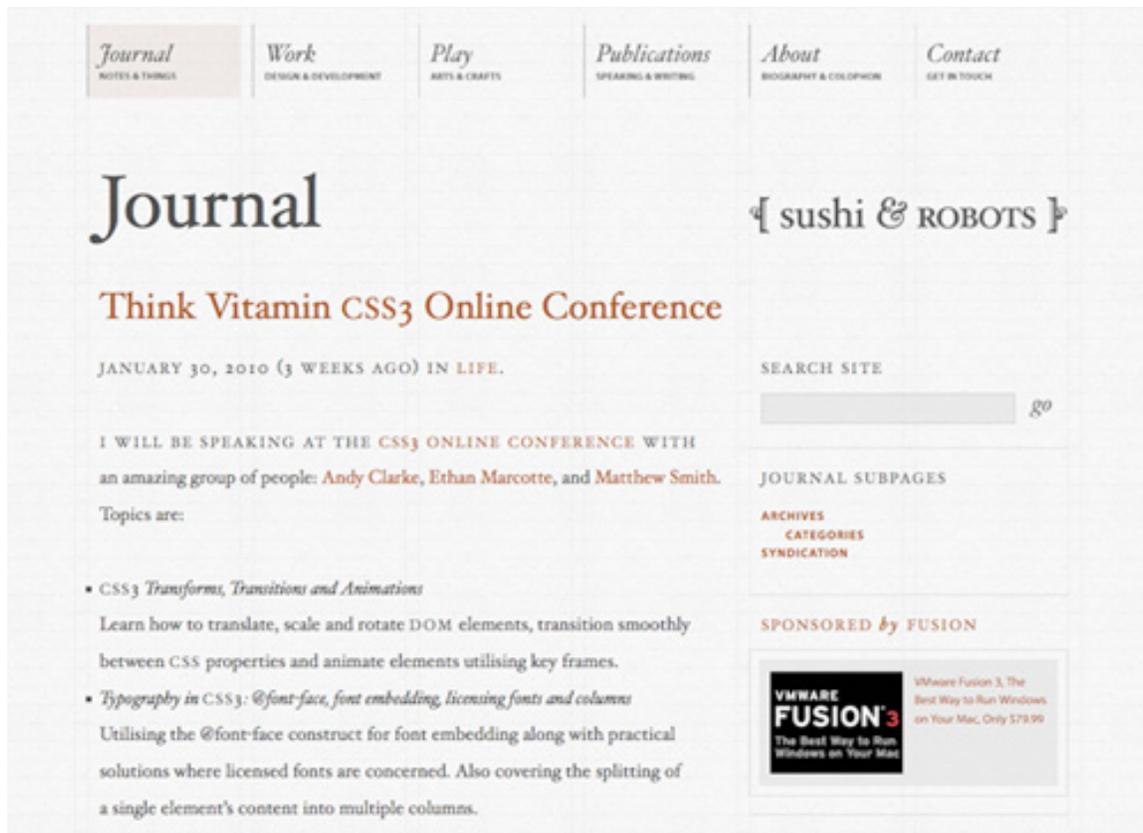
It works slightly different for *letter-spacing*. One value only corresponds to all three values. The others work as they do for *word-spacing*.

The specifications contain a few requests for more information and examples on how [white-space processing](#) will work and how it can be used and be useful for languages such as Japanese, Chinese, Thai, Korean, etc. So, if you'd like help out, why not give it a read (it's not *that* long), and see how you can contribute?

Indentation And Hanging Punctuation

Text indentation and hanging punctuation are two typographical features that are often forgotten on the Web. This is probably due to one of three factors:

1. Setting them is not as straightforward as it could be
2. There has been a conscious decision not to apply them
3. Designers simply aren't aware of them or don't know how to properly use them



The [Sushi & Robots](#) website has hanging punctuation on bulleted lists.

Mark Boulton has a good brief explanation of hanging punctuation in his [“Five Simple Steps to Better Typography”](#) series, and Richard Rutter mentions indentation on his website, [The Elements of Typographic Style Applied to the Web](#). These are two very good reads for any Web designer.

So, the theory is that you should apply a small indentation to every text paragraph after the first one. You can easily do this with an adjacent sibling combinator:

```
1 | p + p {  
2 |   text-indent: 1em;  
3 | }
```

This selector targets every paragraph (i.e. `p`) that follows another paragraph; so the first paragraph is not targeted.

Another typographic rule of thumb is that bulleted lists and quotes should be “hung.” This is so that the flow of the text is not disrupted by these visual distractions.

The CSS Text Level 3 specification has an (incomplete) reference to an [upcoming hanging-punctuation property](#).

For now, though, you can use the `text-indent` property with negative margins to achieve the desired effect:

```
1 | blockquote {  
2 |   text-indent: -0.2em;  
3 | }
```

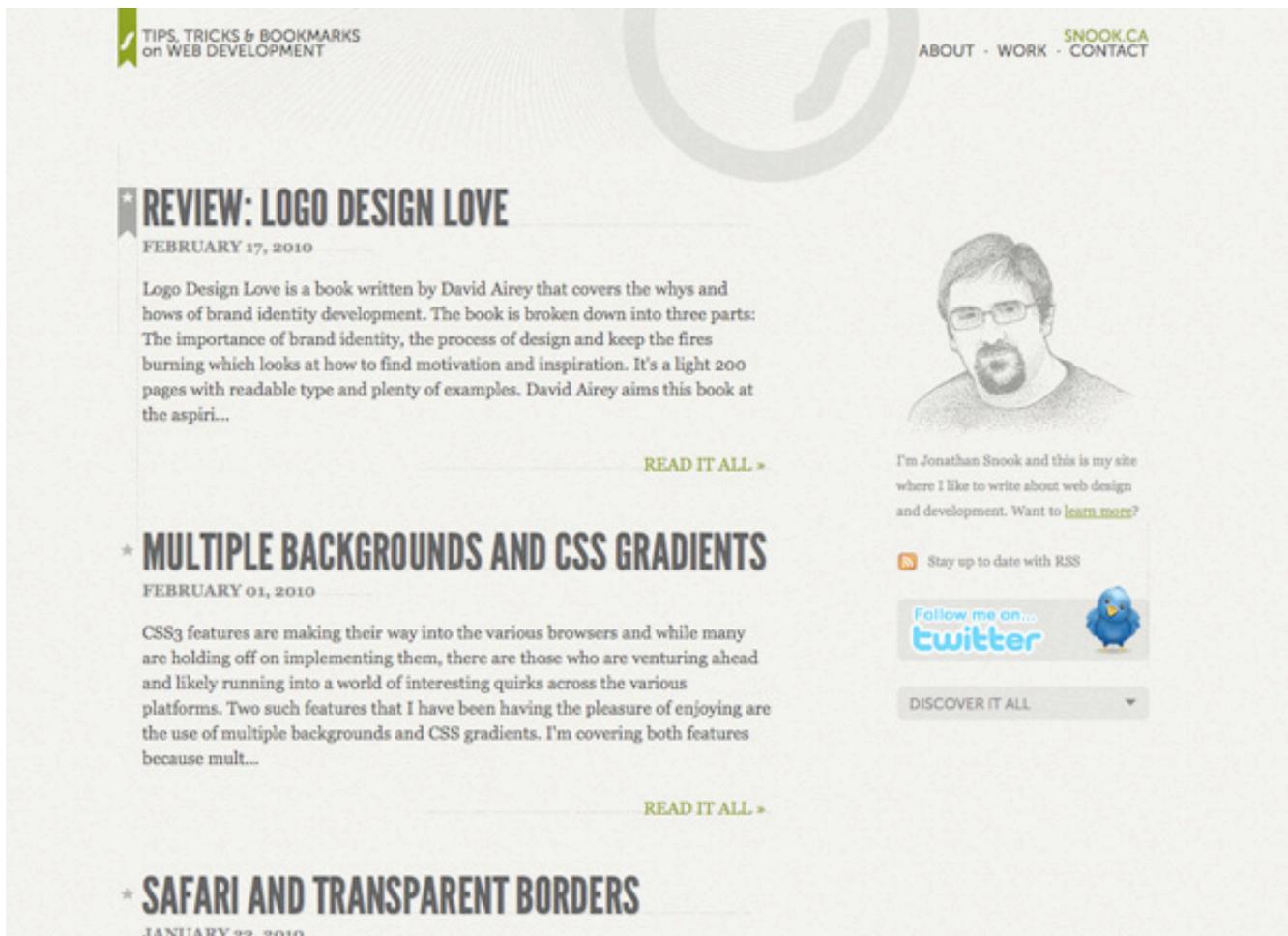
For bulleted lists, just make sure that the position of the bullet is set to `outside` and that the container `div` is not set to `overflow: hidden`; otherwise, the bullets will not be visible.

Web Fonts And Font Decoration

font-face

Much talk has been made on the Web about `font-face` and whether it's a good thing—especially after the appearance of [Typekit](#) (and the still-in-private-beta [Fontdeck](#)). The debate is mainly about how much visual clutter this could bring to Web designs. Some people (the argument goes) aren't sufficiently font-savvy to be able to pull off a design in which they are free to use basically any font they wish. Wouldn't our sensitive designer eyes be safer if only tested, approved Web-safe fonts were used?

On whatever side of the argument you fall, the truth is that the examples of websites that use `font-face` beautifully are numerous.



[Jonathan Snook's](#) recently redesigned website uses the `font-face` property.

The `font-face` property is fairly straightforward to grasp and use. Upload the font you want to use to your website (make sure the licence permits it), give it a name and set the location of the file.

In its basic form, this is what the `font-face` property looks like:

```
1 @font-face {  
2   font-family: Museo Sans;  
3   src: local("Museo Sans"), url(MuseoSans.ttf)  
   format("opentype");  
4 }
```

The two required `font-face` descriptors are `font-family` and `src`. In the first, you indicate how the font will be referenced throughout your CSS file. So, if you want to use the font for `h2` headings, you could have:

```
1 | h2 {  
2 |   font-family: Museo Sans, sans-serif;  
3 | }
```

With the second property (`src`), we are doing two things:

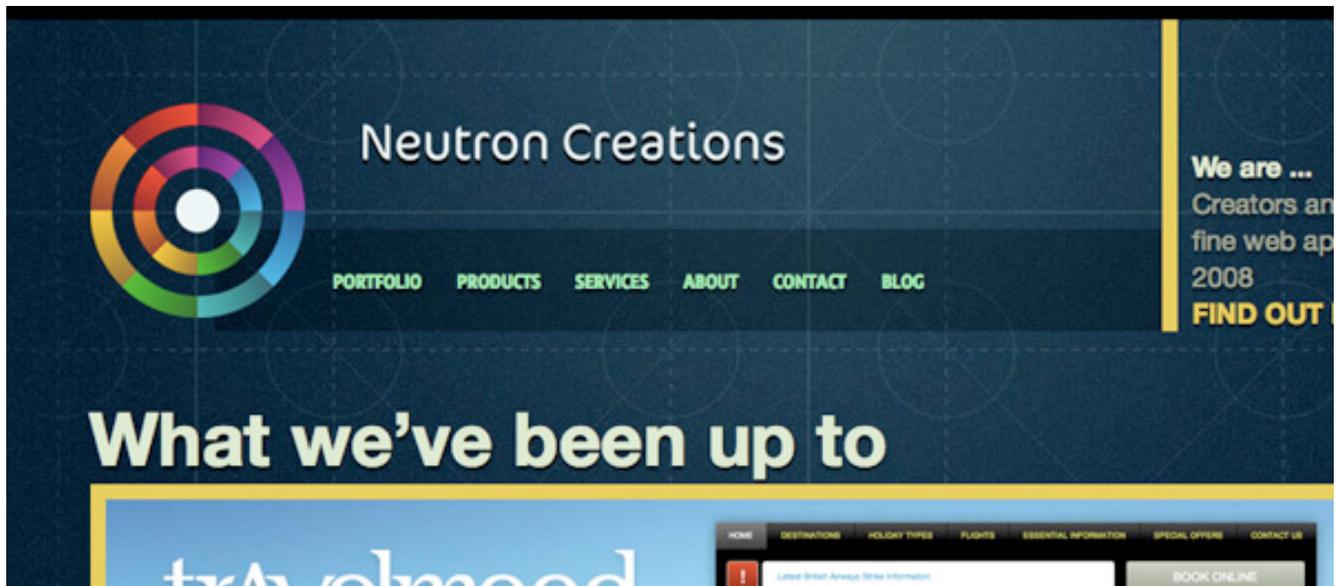
1. If the font is already installed on the user's system, then the CSS uses the local copy instead of downloading the specified font. We could have skipped this step, but using the local copy saves on bandwidth.
2. If no local copy is available, then the CSS downloads the file linked to in the URI. We also indicate the format of the font, but we could have skipped that step, too.

For this property to work in IE, we would also need the EOT version of the font. Some font shops offer multiple font formats, including EOT, but in many cases we will need to convert the TrueType font using Microsoft's own [WEFT](#), or another tool such as [ttf2eot](#).

Some good resources for finding great fonts that can be used with `font-face` are [Font Squirrel](#) and [Fontspring](#).

text-shadow

The `text-shadow` property allows you to add a shadow to text easily and purely via CSS. The shadow is applied to both the text and text decoration if it is present. Also, if the text has `text-outline` applied to it, then the shadow is created from the outline rather than from the text.



Neutron Creations website uses `text-shadow`.

With this property you can set the horizontal and vertical position of the shadow (relative to the text), the color of the shadow and the blur radius. Here is a complete `text-shadow` property:

```
1 p {  
2   text-shadow: #000000 1px 1px 1px;  
3 }
```

Both the color and blur radius (the last value) are optional. You could also use an RGBA color for the shadow, making it transparent:

```
1 p {  
2   text-shadow: rgba(0, 0, 0, 0.5) 1px 1px 1px;  
3 }
```

Here we define the R, G and B values of the color, plus an additional alpha transparency value (hence the a, whose value here is 0.5).

The specification still has some open questions about `text-shadow`, like how should the browser behave when the shadow of an element overlaps the text of an adjoining element? Also, be aware that multiple text shadows and the `text-outline` property may be dropped from the specification.

New Text-Decoration Properties

One problem with the `text-underline` property is that it gives us little control. The [latest draft of the specification](#), however, suggests new and improved properties that may give us fine-grained control. You can't use them yet, but we'll give you a condensed sneak peek at what may come.

- **`text-decoration-line`**

Takes the same values as `text-decoration: none, underline, overline` and `line-through`.

- **`text-decoration-color`**

Specifies the color of the line of the previous property.

- **`text-decoration-style`**

Takes the values of `solid, double, dotted, dashed` and `wave`

- **`text-decoration`**

The shorthand for the three preceding properties. If you specify a value of only one of `none, underline, overline` or `line-through`, then the property will be backwards-compatible with CSS Level 1 and 2. But if you specify all three values, as in `text-decoration: red dashed underline`, then it is ignored in browsers that don't support them.

- **text-decoration-skip**

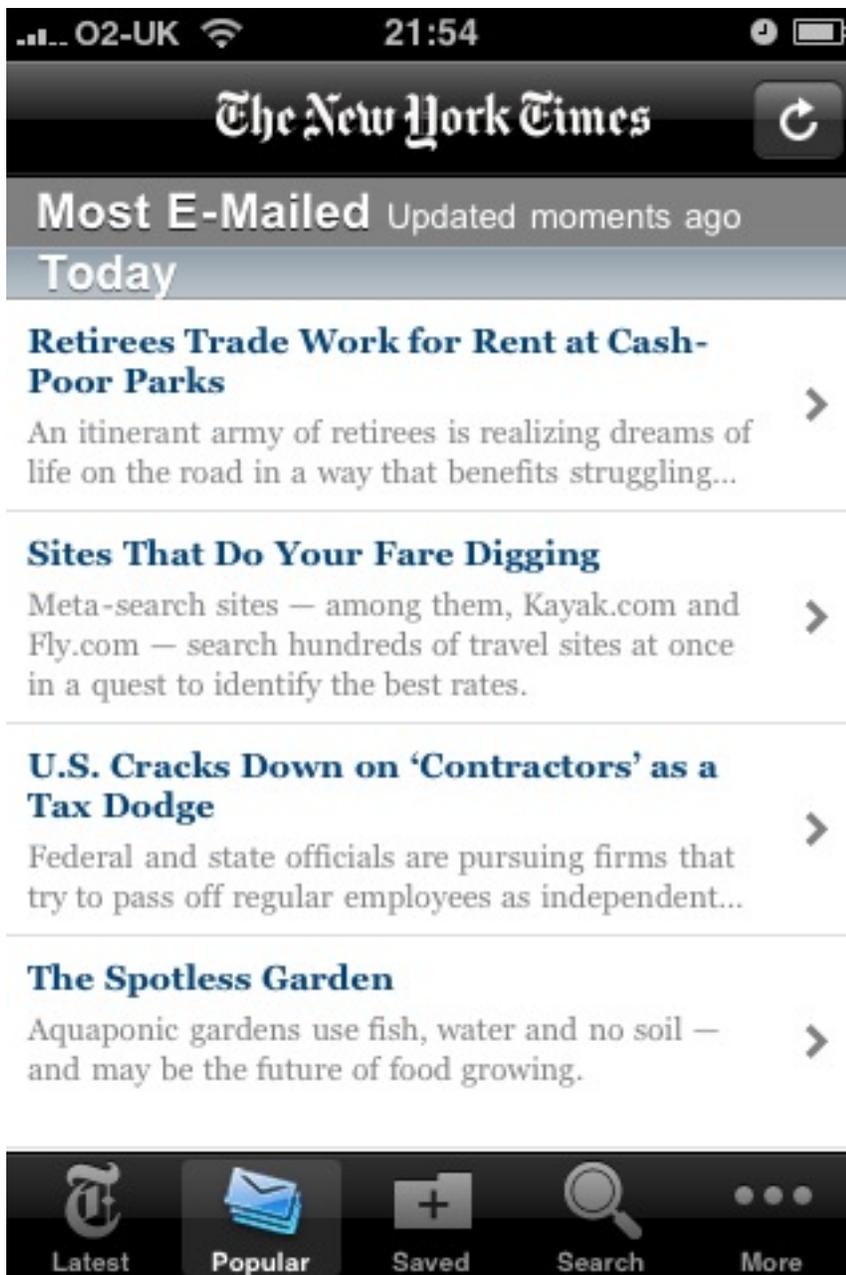
Specifies whether the text decoration should skip certain types of elements. The proposed values are `none`, `images`, `spaces`, `ink` and `all`.

- **text-underline-position**

With this property, you can control, for example, whether the underline should cross the text's descenders or not: `auto`, `before-edge`, `alphabetic` and `after-edge`.

Controlling Overflow

The `text-overflow` property lets you control what is shown when text overflows its container. For example, if you want all of the items in a list of news to have the same height, regardless of the amount of text, you can use CSS to add ellipses (...) to the overflow to indicate more text. This technique is commonly seen in iPhone apps and websites.



The [New York Times](#) iPhone app uses an ellipsis for overflowing text.

This property works in the latest versions of Safari and Opera and in IE6 (where the overflowing element should have a set width, such as 100%) and IE7. To be able to apply the property to an element, the element has to have `overflow` set to something other than `visible` and `white-`

`space: nowrap`. To make it work in Opera, you need to add the vendor-specific property:

```
1 li {  
2   white-space: nowrap;  
3   width: 100%;  
4   overflow: hidden;  
5   -o-text-overflow: ellipsis;  
6   text-overflow: ellipsis;  
7 }
```

In the [Editor's draft of the specification](#), you can see that other properties related to `text-overflow` are being considered, such as `text-overflow-mode` and `text-overflow-ellipsis`, for which `text-overflow` would be the shorthand.

Alignment And Hyphenation

Controlling hyphenation online is tricky. Many factors need to be considered when setting automatic hyphenation, such as the fact that different rules apply to different languages. Take Portuguese, in which you can hyphenate a word only at the end of a syllable; for double consonants, the hyphen must be located right in the middle.

The specification is still being developed, but the proposed properties are:

- `hyphenate-dictionary`
- `hyphenate-before` and `hyphenate-after`
- `hyphenate-lines`
- `hyphenate-character`

This is a good example of how the input of interested Web designers is vital. Thinking about and testing these properties before they are finalized has nothing to do with being “edgy” or with showing off. By proposing changes to the specification and illustrating our comments with examples, we are contributing to a better and stronger spec.

Another CSS3 property that hasn’t been implemented in most browsers (only IE supports it, and only partially) is `text-align-last`. If your text is set to `justify`, you can define how to align the last line of a paragraph or the line right before a forced break. This property takes the following values: `start`, `end`, `left`, `right`, `center` and `justify`.

Unicode Range And Language

Unicode Range

The `unicode-range` property lets you define the range of Unicode characters supported by a given font, rather than providing the complete range. This can be useful to restrict support for a wide variety of languages or mathematical symbols, and thus reduce bandwidth usage.

Imagine that you want to include some Japanese characters on your page. Using the `font-face` rule, you can have multiple declarations for the same `font-family`, each providing a different font file to download and a different Unicode range (or even overlapping ranges). The browser should only download the ranges needed to render that specific page.

To see examples of how `unicode-range` could work, head over to the [spec’s draft page](#).

Language

Use the `:lang` pseudo-class to create language-sensitive typography. So, you could have one background color for text set in French (`fr`) and another for text set in German (`de`):

```
1 | div:lang(fr) {  
2 |   background-color: blue;  
3 | }  
4 |  
5 | div:lang(de) {  
6 |   background-color: yellow;  
7 | }
```

You might be wondering why we couldn't simply use an attribute selector and have something like the following:

```
1 | div[lang|=fr] {  
2 |   background-color: blue;  
3 | }
```

Here, we are targeting all `div` elements whose `lang` attribute is or starts with `fr`, followed by an `-`. But if we had elements inside that `div`, they wouldn't be targeted by this selector because their `lang` attribute isn't specified. By using the `:lang` pseudo-class, the `lang` attribute is inherited to all children of the elements (the whole `body` element could even be holding the attribute).

The good news is that all latest versions of the major browsers support this pseudo-class.

Conclusion

In surveying the examples in this article, you may be wondering why to bother with most of them.

True, the specification is far from being approved, and it could change over time, but now is the time for experimentation and to contribute to the final spec.

Try out these new properties, and think of how they could be improved or how you could implement them to make your life easier in future. Having examples of implementations is important to the process of adding a property to the spec and, moreover, of implementing it in browsers.

You can start with the simple step of subscribing to the [CSS Working Group blog](#) to keep up to date on the latest developments.

So, do your bit to improve the lot of future generations of Web designers... and your own!

Using CSS3: Older Browsers and Common Considerations

Dave Sparks

With the arrival of IE9, Microsoft has signaled its intent to work more with standards-based technologies. With IE still the single most popular browser and in many ways the browser for the uninitiated, this is hopefully the long awaited start of us Web craftsmen embracing the idea of using CSS3 as freely as we do CSS 2.1. However, with IE9 not being supported on versions of Windows before Vista and a lot of businesses still running XP and reluctant (or unable) to upgrade, it might take a while until a vast majority of our users will see the new technologies put to practice.

While plenty of people out there are using CSS3, many aren't so keen or don't know where to start. This article will first look at the ideas behind CSS3, and then consider some good working practices for older browsers and some new common issues.

A Helpful Analogy

The best analogy to explain CSS3 that I've heard relates to the world of film. Filmmakers can't guarantee what platform their viewers will see their films on. Some will watch them at the cinema, some will watch them at home, and some will watch them on portable devices. Even among these few viewing options, there is still a massive potential for differences: IMAX, DVD, Blu-ray, surround sound — somebody may even opt for VHS!

So, does that mean you shouldn't take advantage of all the great stuff that Blu-ray allows with sound and video just because someone somewhere will not watch the film on a Blu-ray player? Of course not. You make the experience as good as you can make it, and then people will get an experience that is suitable to what they're viewing the movie on.

A lot about CSS3 can be compared to 3-D technology. They are both leading-edge technologies that add a lot to the experience. But making a film without using 3-D technology is still perfectly acceptable, and sometimes even necessary. Likewise, you don't need to splash CSS3 gradients everywhere and use every font face you can find. But if some really do improve the website, then why not?

However, simply equating CSS3 to 3-D misses the point. In many cases, CSS3 simply allows us to do the things that we've been doing for years, but without all the hassle.

To Gracefully Degrade or Progressively Enhance?

In film, you create the best film you can make and then tailor the product to the viewing platform. Sound familiar? If you have dabbled in or even taken a peek at CSS3, it should.

There are two schools of thought with CSS3 usage, and it would be safe to say that the fundamental principle of both is to maintain a website's usability for those whose browsers do not support CSS3 capabilities, while providing CSS3 enhancements for those whose browsers do. In other words, make sure the film still looks good even without the 3-D specs. In many ways, the schools of thought are similar, and regardless of which you adopt, you will face many of the same concerns and issues, only from different angles.

Graceful Degradation

With graceful degradation, you code for the best browsers and ensure that as the various layers of CSS3 are peeled away on older browsers, those users still get a usable (even if not necessarily as pleasing an) experience.

The approach is similar (although not identical) to using an IE6-only style sheet, whereby you serve a certain set of styles to most users, while serving alternate styles to users of IE6 and lower. Normally, the IE6 version of a website removes or replaces styling properties that don't work in IE6, along with fixes for any layout quirks. Graceful degradation differs in that it makes use of the natural fallbacks in the browser itself, and fixes are determined by browser capabilities rather than specific browser versions. Also, graceful degradation does not normally require an entirely different set of styles. The result, though, is that the majority of users get the normal view, and then tweaks are applied for people who have yet to discover a better browser.

Aggressive graceful degradation is at the heart of Andy Clarke's recent book, *Hardboiled Web Design*, and the [accompanying website](#) makes great use of graceful degradation. There are plenty of other examples, including [Do Websites Need to Look Exactly the Same in Every Browser.com](#), which was built to showcase the technique, and Virgin Atlantic's [vtravelled blog](#), designed by John O'Nolan, which shows some great subtle fallbacks that most users wouldn't even notice. And if you're a WordPress user, why not compare your admin dashboard in IE to it in another browser?

Progressive Enhancement

Progressive enhancement follows the process in reverse: that is, building for lower-support browsers and then using CSS3 to enhance the experience of

those with more capable browsers. This used to be done, and still is by some, with separate enhancement style sheets.

As a starting point, most people will code for a sensible standards-based browser, then add some code to support browsers such as IE7 and 8, and then possibly throw in some fixes for IE6 for good measure, and then step back and think, "How can I improve this with CSS3?" From there, they would add properties such as rounded corners, gradients, `@font-face` text replacement and so on.

As browser makers add support, progressive enhancement appears to be taking a back seat to graceful degradation. But progressive enhancement is a very good approach for getting started with CSS3 properties and learning how they work.

Examples of the technique include the personal websites of [Sam Brown](#) and [Elliot Jay Stocks](#), which both feature enrichment-type style sheets, Elliot has spoken on the matter, and the slides from his 2009 Web Directions South talk, "[Stop Worrying and Get on With It \(Progressive Enhancement and Intentional Degradation\)](#)," make for good reading.



Elliot Jay Stock's presentation ['Stop Worrying and Get on With It \(Progressive Enhancement and Intentional Degradation\)'](#)

Comparing the two, graceful degradation can be considered a top-down approach, starting with browsers most capable of utilizing CSS3 and working down to older browsers that lack support.

Progressive enhancement works the other way, bottom-up, using a standards-based browser of choice as the baseline, along maybe with IE7, and then adding CSS3 for browsers that support it. Its benefit is that it is easy to work with when you're just getting used to CSS3, and it's also a sensible approach when adding CSS3 to older websites.

Whichever approach you choose, there are a number of things to consider,

what with all the CSS3 properties that are coming out. Later on, we will look at considerations for certain key properties.

How To Do It?

Whatever your approach, you will no doubt find yourself thinking through the common fallback process at some point: what would this element look like with a certain property, and what would it look like without it? Would it look fine or broken? If it would look broken, there's a good chance you will need to do something about it.

As a typical path, you would first implement a feature with CSS3, then with CSS 2.1, then (maybe) with JavaScript, and then with whatever hack you used to use for legacy browsers. You could argue that progressive enhancement would slightly modify this path, using CSS 2.1 first, then CSS3.

At each stage, you should determine whether degrading or enhancing a feature would become unnecessarily complex and whether simply providing an alternative would be more sensible.

Ordering Properties

Let's take a quick look at ordering properties and how browsers interpret them. Browser makers initially offer CSS3 functionality via browser prefixes: `-moz` for Mozilla, `-webkit` for Chrome and Safari, `-o` for Opera, etc. Each browser then ignores any prefixes not meant for it. The convention is to list the browser-specific prefixes first and then the default property, as follows:

```
1 | .somediv {  
2 |   -moz-border-radius: 5px;  
3 |   -webkit-border-radius: 5px;  
4 |   border-radius: 5px; }
```

Yes, this creates a little overhead, but when you consider how such effects were achieved before CSS3, it's really not much.

Browsers that don't support the property will ignore it. Any browser that does support it will implement its browser-specific version; and when it eventually supports the generic property, it will implement that.

Why order it in this way? Once all of the browsers implement a property the same way, then they will adopt the default version of the property; until then, they will use the prefixed version. By listing the properties in the order shown above, we ensure that the standard version is implemented as the fallback once it is supported, hopefully leading to more consistent rendering across browsers.

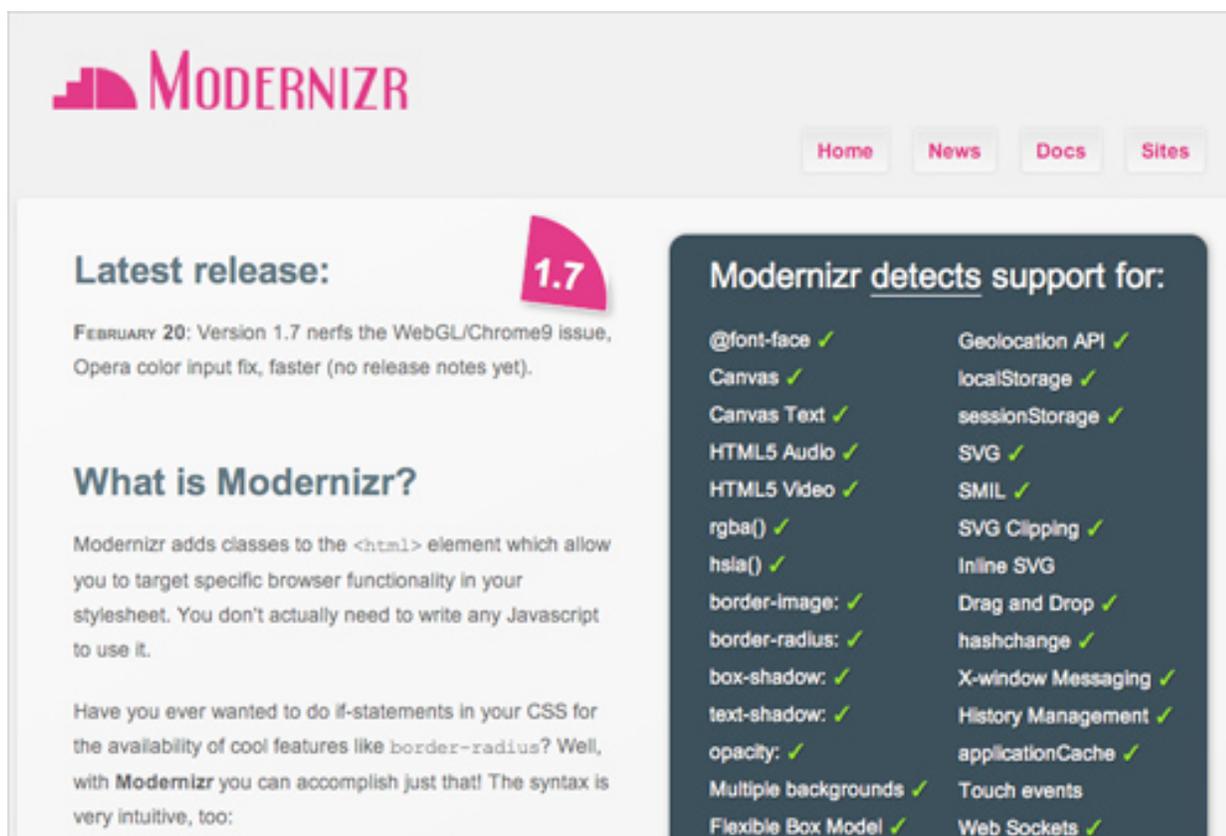
JavaScript

JavaScript is the most common method of enabling cross-browser CSS3 features support, and it can either be used as a substitute for or to enable CSS3 properties in older browsers or be used as an alternative.

Modernizr

A useful JavaScript tool for implementing CSS3 fallbacks is [Modernizr](#). For anyone working with CSS3 in a production environment (as opposed to merely as a proof of concept), it is essential. Modernizr enables you to use

CSS3 for properties where it is supported, and to provide sensible alternatives where it isn't.



Modernizr works by adding classes to the `html` element of the page, which you would then call in the style sheet.

For example, to display a different background when CSS3 gradients are not supported, your code would look something like this:

```
1 | .somediv {  
2 |   background: -webkit-gradient(linear, 0% 0%, 0% 100%,  
3 |     from(#660C0C), to(#616665), color-stop(  
4 |     6, #0D0933)); }  
5 |  
6 | .no-cssgradients .somediv {  
7 |   background: url('/images/gradient.jpg'); }
```

Conversely, to display a different background only where the CSS3 property is supported, you would do this:

```
1 .cssgradients .somediv {
2   background: -webkit-gradient(linear, 0% 0%, 0% 100%,
3     from(#660C0C), to(#616665), color-stop(
4     6, #0D0933)); }
5 .somediv {
6   background: url('/images/gradient.jpg'); }
```

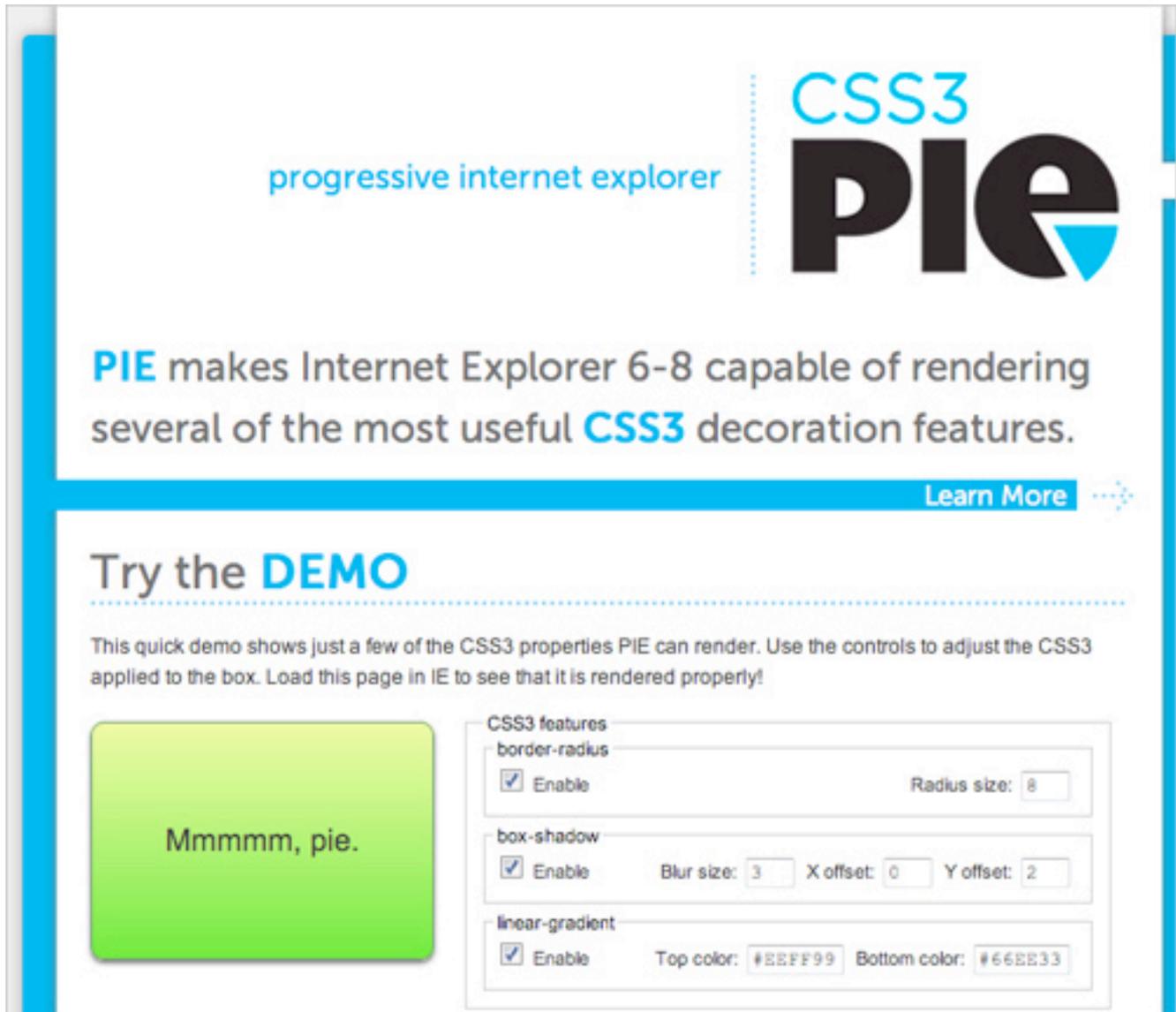
In this way, you control what is shown in the absence of a property, and you tailor the output to what is sensible. In this case, you could serve a gradient image in the absence of support for CSS3 gradients.

With this additional control, you can tailor the output quite accurately and avoid any clashes that might arise from a missing property.

CSS3 PIE

Sadly, this has nothing to do with the tasty dessert. [CSS3 PIE](#) stands for *progressive Internet Explorer*. As the official description says:

PIE makes Internet Explorer 6 to 8 capable of rendering several of the most useful CSS3 decoration features.



The screenshot shows the CSS3 PIE website. At the top, it says "progressive internet explorer" and "CSS3 PIE". Below that, it states "PIE makes Internet Explorer 6-8 capable of rendering several of the most useful CSS3 decoration features." There is a "Learn More" link. The main section is titled "Try the DEMO" and includes a description: "This quick demo shows just a few of the CSS3 properties PIE can render. Use the controls to adjust the CSS3 applied to the box. Load this page in IE to see that it is rendered properly!". On the left is a green gradient box with the text "Mmmm, pie.". On the right are controls for "CSS3 features": "border-radius" (checked, Radius size: 8), "box-shadow" (checked, Blur size: 3, X offset: 0, Y offset: 2), and "linear-gradient" (checked, Top color: #EEFF99, Bottom color: #66EE33).

<http://css3pie.com>

While it doesn't support a myriad of features, it does allow you to use `box-shadow`, `border-radius` and linear gradients in IE without doing much extra to the code. First, upload the CSS PIE JavaScript file, and then when you want to apply the functionality, you would include it in the CSS, like so:

```
1 .somediv {
2   -webkit-border-radius: 5px;
3   -moz-border-radius: 5px;
4   border-radius: 5px;
5   behavior: url(path/to/PIE.htc); }
```

Fairly straightforward, and it can save you the hassle of having to use JavaScript hacks to achieve certain effects in IE.

Selectivizr

CSS3 has expanded its repertoire beyond advanced selectors such as `[rel="selector"]` and pseudo-selectors such as `:focus`, to include selectors such as `:nth-of-type`, which give you much more control and focus and allow you to dispense with a lot of presentational classes and IDs. Support for selectors varies greatly, especially with the wide variety of additional selectors being introduced.

selectivizr

CSS3 selectors for IE

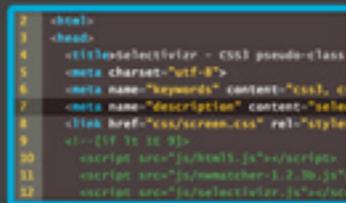
selectivizr is a JavaScript utility that emulates CSS3 pseudo-classes and attribute selectors in Internet Explorer 6-8. Simply include the script in your pages and selectivizr will do the rest.

 **DOWNLOAD**
v1.0.1 - (4k .ZIP archive)



Enhancing IE's selector engine

Selectivizr adds support for 19 CSS3 pseudo-classes, 2 pseudo-elements and every attribute selector to older versions of IE. It can also fix a few of the browser's native selector implementations.



JavaScript-knowledge: none

Selectivizr works automatically so you don't need any JavaScript knowledge to use it — you won't even have to modify your style sheets. Just start writing CSS3 selectors and they will work in IE.



Works with existing tools

Selectivizr requires a JavaScript library to work. If your website already uses one of the 7 supported libraries you just need to add the selectivizr script to your pages. If not, you will need to pick a library too.

Therefore, the third weapon in your CSS3 arsenal will most likely be [Selectivizr](#), which enables advanced CSS3 selectors to be used in older browsers and is aimed squarely at old IE versions.

Head over to the Selectivizr website and download and add the script. You will have to pair it with a JavaScript framework such as jQuery or MooTools, but chances are you're working with one already. The home page is worth a quick look because not all selectors are supported by all JavaScript libraries, so make sure what you need is supported by your library of choice.

Problems?

The main issue with all of the solutions above is that they're based on JavaScript, and some website owners will be concerned about users who have neither CSS3 support nor JavaScript enabled. The best solution is to code sensibly and make use of natural CSS fallbacks and allow the browser to ignore CSS properties that it doesn't recognize.

This may well make your website look a bit less like the all-singing, all-dancing CSS3-based design that you had in mind, but remember that the number of people without CSS3 support *and* without JavaScript enabled will be low, and the best you can do is make sure they get a usable, functional and practical experience of your website, thus allowing you to continue tailoring the output to the user's platform.

Some CSS3 Properties: Considerations And Fallbacks

Many CSS3 properties are being used, and by now we have gotten used to the quirks and pitfalls of each iteration of the CSS protocol. To give you a quick start on some of the more popular CSS3 properties, we'll look at some of the issues you may run into and some sensible ways to fall back in older browsers.

All of the information in this article about browser support is correct as of May 2011. You can keep up to date and check out further information about support by visiting [findmebyIP](#). Support has not been checked in browser versions older than Chrome 7.0, Firefox 2.0, Opera 9, Safari 2 and Internet Explorer 6.

Border Radius

Support: Google Chrome 7.0+, Firefox (2.0+ for standard corners, 3.5+ for elliptical corners), Opera 10.5+, Safari 3.0+, IE 9

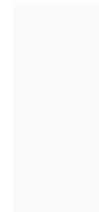
Property: border-radius

Vendor prefixes: -webkit-border-radius, -moz-border-radius

Example usage (even corners with a radius of 5 pixels):

```
1 | .somediv {  
2 |   -moz-border-radius: 5px;  
3 |   -webkit-border-radius: 5px;  
4 |   border-radius: 5px; }
```

Fallback behavior: rounded corners will display square.



WordPress log-in button in IE (left) and Google Chrome (right).

Without the hassle of extra divs or JavaScript or a lot of well-placed, well-sliced images, we can give elements rounded corners with the use of the straightforward `border-radius` property.

What about browsers that don't support `border-radius`? The easiest answer is, don't bother. Is having rounded corners in unsupported browsers really worth the hassle? If it is, then you need only do what you've been doing for years: JavaScript hacks and images.

Could this property trip you up? Actually, `border-radius` is pretty straightforward. Be careful using it on background images, because there are certainly some bugs in some browser versions that keep the corners of images from appearing rounded. But aside from that, this is one of the best-supported CSS3 properties so far.

Border Image

Support: Google Chrome 7.0+, Firefox 3.6+, Opera 11, Safari 3.0+, no support in IE

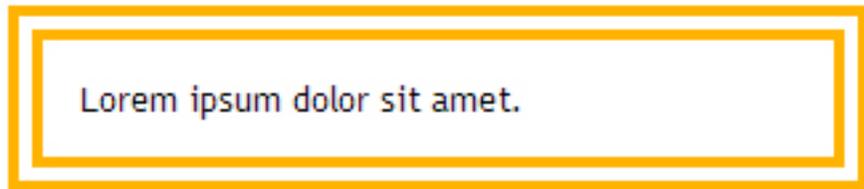
Property: `border-image`, `border-corner-image`

Vendor prefixes: `-webkit-border-image`, `-moz-border-image`

Example usage (a repeating image with a slice height and width of 10 pixels):

```
1 | .somediv {  
2 |   -webkit-border-image: url(images/border-image.png) 10  
   |   10 repeat;  
3 |   -moz-border-image: url(images/border-image.png) 10 10  
   |   repeat;  
4 |   border-image: url(images/border-image.png) 10 10  
   |   repeat; }
```

Fallback behavior: shows standard CSS border if property is set, or no border if not specified.



A `border-image` demo on CSS3.info. The bottom paragraph shows a standard property of `border: double orange 1em`.

The `border-image` property is less heralded among the new properties, partly because it can be a bit hard to wrap your head around. While we won't go into detail here, consider the image you are working with, and test a few variations before implementing the property. What will the border look like if the content overflows? How will it adjust to the content? Put some thought into your choice between `stretch` and `repeat`.

Experiment with an image before applying a border to make sure that everything is correct, and test different sizes and orientations to see how a repeating border looks.



A border image in use on Blog.SpoonGraphics. The image on the left is the base image for the border.

There isn't much in the way of fallbacks, aside from the traditional method of coding for eight slice-image borders, mapped out with extra containing divs. This is a lot of work and is really unnecessary. Selecting an appropriate border color and width should be a sensible fallback for browsers without border-image support.

Box Shadow

Support: Google Chrome 7.0+, Firefox 3.6+, Safari 3.0+, IE 9

Property: box-shadow

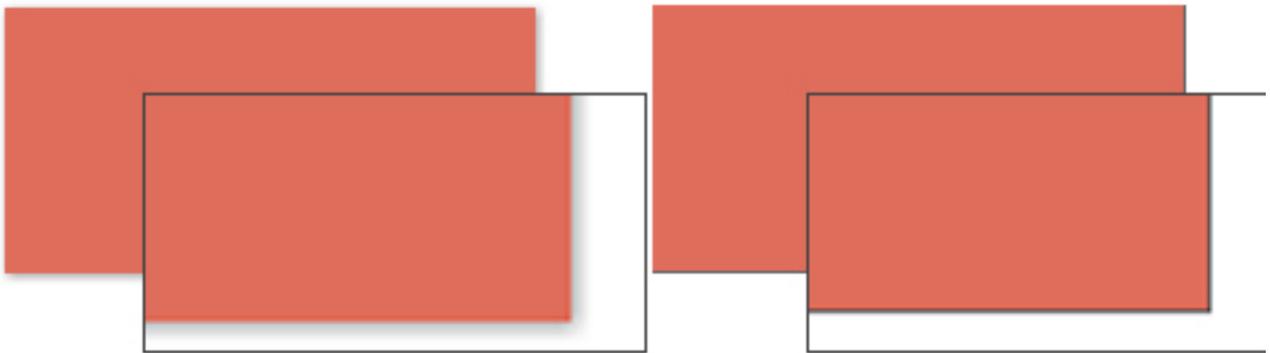
Vendor prefixes: `-webkit-box-shadow`, `-moz-box-shadow` (`-moz` no longer needed as of Firefox 4)

Example usage (showing a black shadow, offset down by 10 pixels and right by 10 pixels, and with a blur radius of 5 pixels):

```
1 .somediv {
2   -moz-box-shadow: 10px 10px 5px #000;
3   -webkit-box-shadow: 10px 10px 5px #000;
4   box-shadow: 10px 10px 5px #000; }
```

Fallback behavior: shadow is not displayed.

Box shadow allows you to quickly and easily add a little shadow to your elements. For anyone who has used shadows in Photoshop, Fireworks or the like, the principles of box shadow should be more than familiar.



A subtle box shadow on the left, and a selective borders fallback on the right.

In its absence? You could use selective borders (i.e. a left and bottom border to imitate a thin box shadow).

```
1 .somediv {
2   -moz-box-shadow: 1px 1px 5px #888;
3   -webkit-box-shadow: 1px 1px 5px #888;
4   box-shadow: 1px 1px 5px #888; }
5
6 .no-boxshadow .somediv {
7   border-right: 1px solid #525252;
8   border-bottom: 1px solid #525252; }
```

RGBa and HSLa

RGBa support: Google Chrome 7.0+, Firefox 3.0+, Opera 10+, Safari 3.0+, IE 9

HSLA support: Google Chrome 7.0+, Firefox 3.0+, Opera 10+, Safari 3.0+

Property: `rgba`, `hsla`

Fallback behavior: the color declaration is ignored, and the browser falls back to the previously specified color, the default color or no color.

```
1 | .somediv {  
2 |   background: #f00;  
3 |   background: rgba(255, 0, 0, 0.5); }
```

In the example above, both background declarations specify the color red. Where RGBa is supported, it will be shown at 50% (0.5), and in other cases the fallback will be to the solid red (#f00).



[24 Ways](#) makes great creative use of RGBa.

While there is broad support for opacity, its downside is that everything associated with an element becomes transparent. But now we have two new ways to define color: RGBa (red, green, blue, alpha) and HSLa (hue, saturation, light, alpha).

Both offer new ways to define colors, with the added benefit of allowing you to specify the alpha channel value.

The obvious fallback for RGBA and HSLA is a solid color; not a problem, but the main thing to watch out for is legibility. A semi-transparent color can have quite a different tone to the original. An RGB value shown as a solid color and the same value at .75 opacity can vary massively depending on the background shade, so be sure to check how your text looks against the background.

If transparency is essential, you could also use a background PNG image. Of course, this brings with it the old IE6 problem, but that can be solved with JavaScript.

Transform

Support: Google Chrome 7.0+, Firefox 3.6+, Opera 10.5+, Safari 3.0+

3-D transforms support: Safari

Property: `transform`

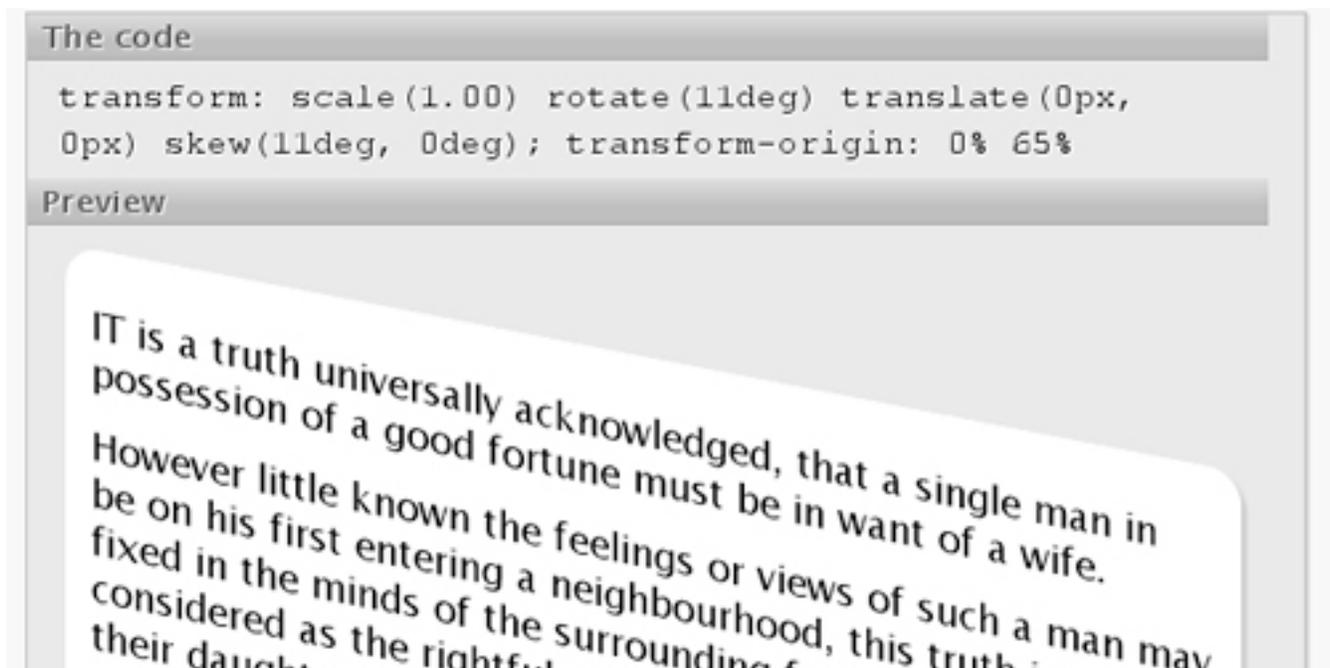
Vendor prefixes: `-o-transform`

Example usage (rotating a div 45° around the center, and scaling it to half the original size — for illustration only, so the `translate` and `skew` values are not needed):

```
1 | .somediv {
2 |   -webkit-transform: scale(0.50) rotate(45deg)
3 |     translate(0px, 0px) skew(0deg, 0deg);
4 |   -webkit-transform-origin: 50% 50%;
5 |   -moz-transform: scale(0.50) rotate(45deg)
```

```
6   translate(0px, 0px) skew(0deg, 0deg);
7   -moz-transform-origin: 50% 50%;
8   -o-transform: scale(0.50) rotate(45deg)
9   translate(0px, 0px) skew(0deg, 0deg);
10  -o-transform-origin: 50% 50%;
11  transform: scale(0.50) rotate(45deg)
12  translate(0px, 0px) skew(0deg, 0deg);
13  transform-origin: 50% 50%; }
```

Fallback behavior: the transform is ignored, and the element displays in its original form.



[Westciv](#) offers a useful tool for playing around with transforms.

The `transform` property gives you a way to rotate, scale and skew an element and its contents. It's a great way to adjust elements on the page and give them a slightly different look and feel.

A simple fallback in the absence of an image-based transform is to use an alternative image that is already rotated. And if you want to rotate content? Well, you can always use JavaScript to rotate the element. Another simple alternative is to rotate the background element in an image editor beforehand and keep the content level.

We've gotten by with level elements for so many years, there's no reason why people on old browsers can't continue to put up with them.

Animations and Transitions

Transitions support: Google Chrome 7.0+, Firefox 4.02, Opera 10.5+, Safari 3.0+

Animations support: Google Chrome 7.0+, Safari 3.0+

Property: transition

Vendor prefixes: -webkit-transition, -moz-transition, -o-transition

Example usage (a basic linear transition of text color, triggered on hover):

```
1 .somediv:hover {
2   color: #000;
3   -webkit-transition: color 1s linear;
4   -moz-transition: color 1s linear;
5   -o-transition: color 1s linear;
6   transition: color 1s linear; }
```

A basic animation that rotates an element on hover:

```
1 @-webkit-keyframes spin {
2   from { -webkit-transform: rotate(0deg); }
3   to { -webkit-transform: rotate(360deg); }
4 }
5
6 .somediv:hover {
7   -webkit-animation-name: spin;
8   -webkit-animation-iteration-count: infinite;
9   -webkit-animation-timing-function: linear;
10  -webkit-animation-duration: 10s; }
```

Fallback behavior: both animations and transitions are simply ignored by unsupported browsers. With animations, this means that nothing happens, and no content is animated. With transitions, it depends on how the transition is written; in the case of a hover, such as the one above, the browser simply displays the transitioned state by default.



[The 404 page](#) for the 2010 Future of Web Design conference attracted attention for its spinning background. Visit the website in IE and you'll see a static background image.

Animations and transitions in CSS3 are slowly seeing more use, from subtle hover effects to more elaborate shifting and rotating of elements across the page. Most effects either start right at page load or (more frequently) are used to enhance a hover effect. Once you get down and dirty with animations, there's great fun to be had, and they're much more accessible to designers now than before.

Starting off small with the CSS3 `transition` property is best, subtly transitioning things such as link hovers before moving on to bigger things.

Once you're comfortable with basic transitions and transforms, you can get into the more involved `animation` property. To use it, you declare keyframes of an animation with `@-webkit-keyframes` and then call this keyframe animation in other elements, declaring its timing, iterations, etc. Note that animations work better with CSS3 transforms than with other

properties, so stick to `transform` and `translate` rather than shifting margins or absolute positioning.

Of course, people have been animating with JavaScript for years. But if you want to do something as simple as animating a hover state, then it hardly seems worth the extra coding. The simplest thing to do for unsupported browsers is to specify a state for hover, without any transition to it.

Font Face (not new in CSS3)

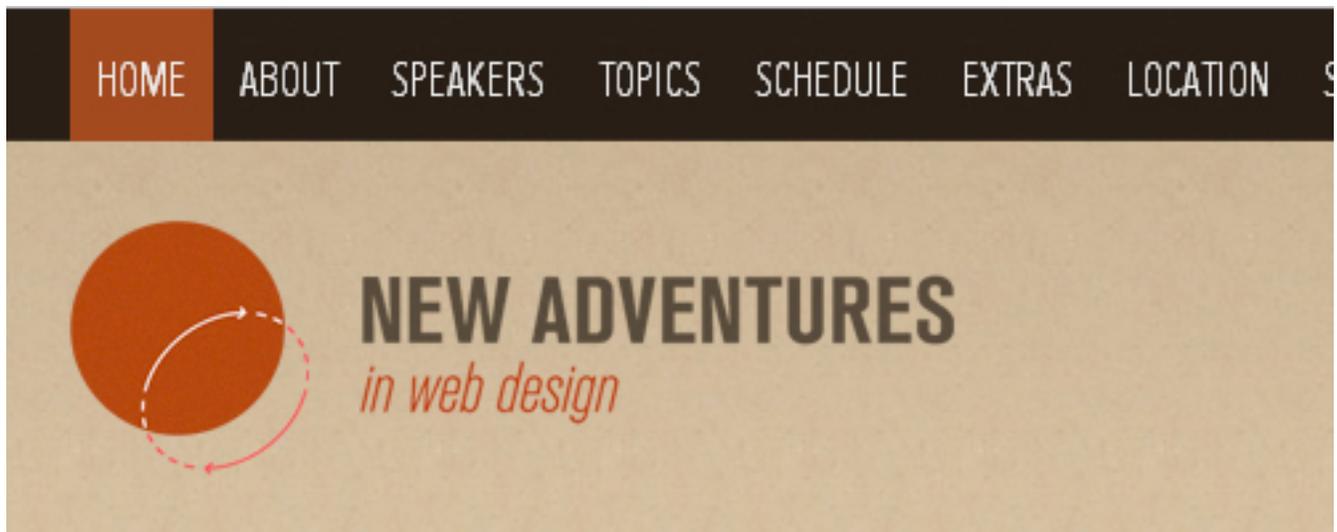
Support for different font formats: Google Chrome 7.0+, Firefox 3.1+, Opera 10+, Safari 3.1+, IE 6+

Property: `@font-face`

Example usage (a `@font-face` declaration for Chunk Five, an OTF font, and calling it for `h1` headings):

```
1 @font-face {
2   font-family: ChunkF; src: url('ChunkFive.otf'); }
3
4 h1 {
5   font-family: ChunkF, serif; }
```

Fallback behavior: just as when any declared font isn't available, the browser continues down the font stack until it finds an available font.



The [New Adventures in Web Design](#) conference serves fonts from [Typekit](#).

Okay, this isn't strictly new to CSS3. Many of you will point out that this has been around as long as IE5. However, text replacement is certainly starting to see increased usage as browser makers roll out increased support for `@font-face`.

One issue that `@font-face` suffers from is that a font isn't displayed on the screen until the browser has downloaded it to the user's machine, which sometimes means that the user sees a "flash of unstyled text" (FOUT). That is, the browser displays a font from further down the stack or a default font until it has finished downloading the `@font-face` file; once the file has downloaded, the text flashes as it switches to the `@font-face` version. So, minimizing the flash by stacking fonts of a similar size and weight is important. If the stack is poorly compiled, then not only could the text be resized, but so could containing elements, which will confuse users who have started reading the page before the proper font has loaded.

The good news is that Firefox 4 doesn't have a FOUT any longer, IE9, however, does have a FOUT but WebInk has written a script [FOUT-B-GONE](#)

which takes these facts into account and helps you hide the FOUT from your users in FF3.5-3.6 and IE.

Too Many Friends

DECEMBER 26, 2010 - 11:30 PM

Too Many Friends

DECEMBER 26, 2010 - 11:30 PM

On his blog, Web designer [Florian Schroiff](#) uses `@font-face` to serve the Prater font (bottom), falling back to Constina, Georgia (top) and Times New Roman.

Many font delivery services, including [TypeKit](#) and [Google Web Fonts](#), deliver their fonts via JavaScript, which gives you control over what is displayed while the font is being downloaded as well as what happens when the font actually loads.

Because browsers wait until the full file of a font kit has loaded before displaying it, plenty of services allow you to strip out unnecessary parts of the kit to cut down on size. For example, if you're not going to be using small caps, then you can strip it out of the file so that the font renders more quickly.

Advanced Selectors

Support (varies depending on the selector used): Google Chrome 7.0+, Firefox 3.6+, Opera 9.0+, Safari 2.0+, IE 6+

Property: many, including `:nth-of-type`, `:first-child`, `:last-child`, `[attr="..."]`

Example usage (coloring only links that point to Smashing Magazine, and coloring odd-numbered rows in tables):

```
1 a[href*=smashingmagazine.com] {  
2   color:#f00; }  
3  
4 tr:nth-of-type(odd) {  
5   background: #ddd; }
```

Fallback behavior: In the absence of support for advanced selectors, the browser does not apply the targeted style to the element and simply treats it as any other element of its type. In the two examples above, the link would take on the standard link properties but not the specified color, and the odd-numbered table rows would be colored the same as other table rows.

Advanced selectors are a great tool for reducing the code on a website. You will be able to get rid of many presentational classes and gain more control of the selections in your style sheet.

Using Selectivizr, you can get older browsers to support these advanced selectors, which makes the selectors easier to use and more robust.

Abandoning classes and IDs altogether in favor of `nth-type` is tempting. But don't throw them away just yet. Use advanced selectors when an element's style is based on its location in the document or a series; for example, using `nth-type(odd)` for table rows or using `last-of-type` to remove some padding at the bottom of a list.

If an element's style is based on its content, then stick with classes and IDs. That is, if inserting a new element or changing the order of items would break the style, then stick with the old method.

However, if an element is already sufficiently styled, then you probably don't need an additional class or ID at all (nor an advanced selector, for that matter).

Columns

Support: Google Chrome 7.0+, Firefox 2.0+, Safari 3.0+, Opera 11.10+

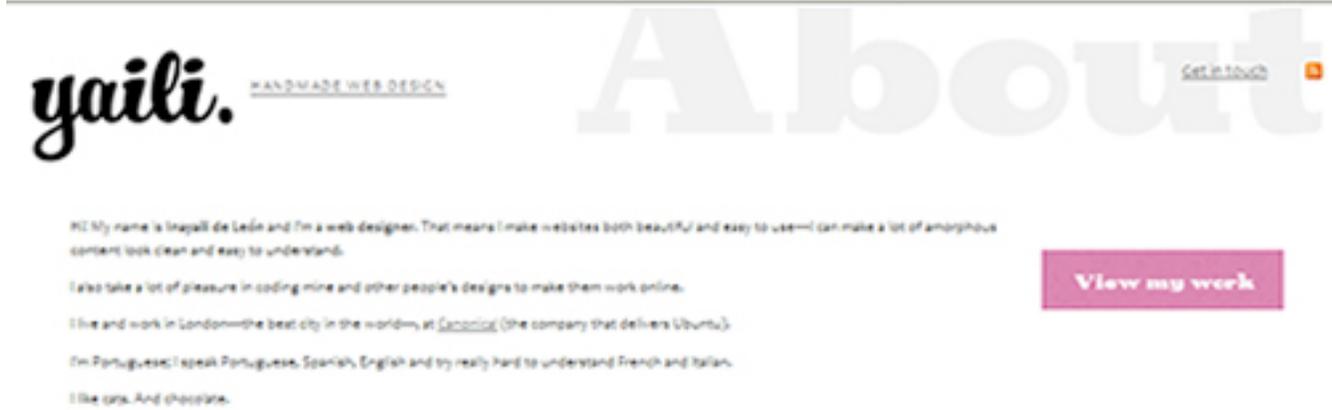
Property: column-count

Vendor prefixes: -webkit-column-count, -moz-column-count

Example usage (splitting content into three columns):

```
1 | .somediv {  
2 |   -moz-column-count: 3;  
3 |   -webkit-column-count: 3;  
4 |   column-count: 3; }
```

Fallback behavior: in the absence of support for multiple columns, the browser spreads the content across the full width that the columns would have taken up.



Multiple columns and their fallback on [Inayaili de León's website](#).

This property gives you a nice easy way to spread content across multiple columns. The technique is standard in print, and on the Web it makes it easy to read content without needing to scroll. But you didn't need me to tell you that, did you?

Because the property's main purpose is to allow users to read horizontally without scrolling, first make sure that your columns aren't too tall. Having to scroll up and down to read columns not only defeats their purpose but actually makes the content harder to read.

There are some JavaScript solutions for multiple columns. For older browsers, though, there's generally no need to stick with a multi-column layout; rather, you could consider sensible alternatives for fallbacks.

In the absence of CSS3 support, the browser will flow content across the full width of the container. You'll want to be careful about legibility. It can be very hard to read content that spans the width of an area that is intended to be broken into three columns. In this case, you'll want to set a suitable line length. There are a few ways to do so: increase the margins, change the font size or decrease the element's width. Floating elements such as images and block quotes out of the flow of text can help to fill up any leftover space in the single column.

Gradients

Support: Google Chrome 7.0+ for `-webkit-gradient`, Google 10+ for `-webkit-linear-gradient` and `-webkit-radial-gradient`, Firefox 3.6+, Safari

Property: `linear-gradient`, `radial-gradient`

Vendor prefixes: `-webkit-gradient`, `-webkit-linear-gradient`, `-webkit-radial-gradient`, `-moz-linear-gradient`, `moz-radial-gradient`

Example usage (a linear white-to-black gradient running from top to bottom — notice the lack of `-linear-` in the Webkit declaration):

```
1 .somediv {
2   background-image: -webkit-gradient(linear, 0% 0%, 0%
3     100%,
4     from(#ffffff), to(#000000));
5   background-image: -webkit-linear-gradient(0% 0%, 0%
6     100%,
7     from(#ffffff), to(#000000));
8   background-image: -moz-linear-gradient(0% 0% 270deg,
9     #ffffff, #000000);
```

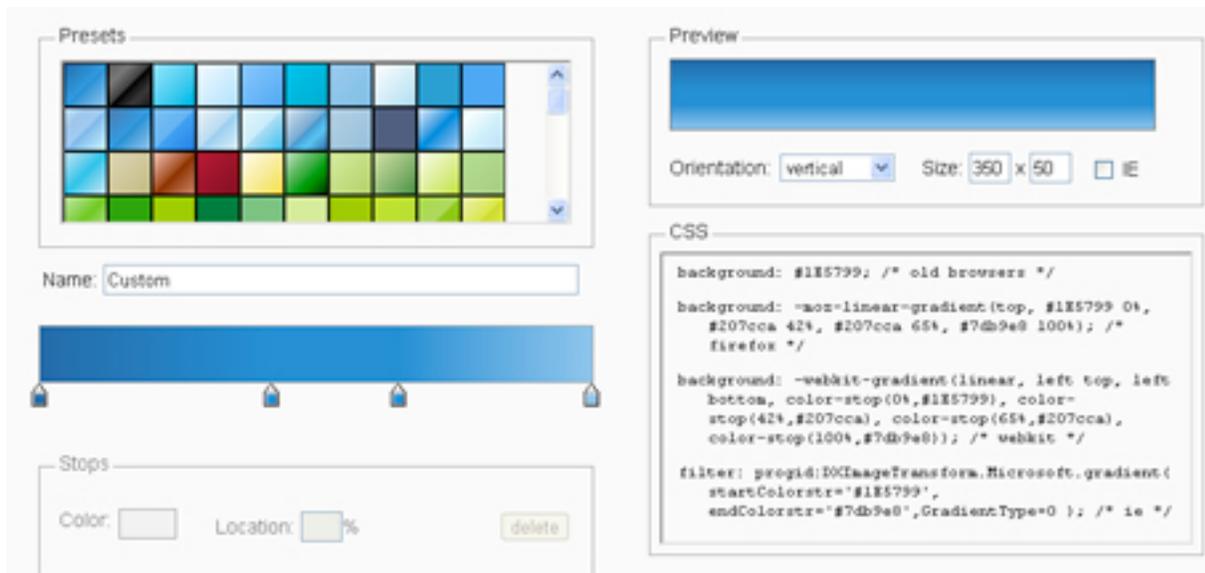
```
8  background-image: linear-gradient(0% 0% 270deg,  
9  #ffffff, #000000); }  
  
1  .somediv {  
2  background-image: -moz-radial-gradient(50% 50%,  
   farthest-side,  
3  #ffffff, #000000);  
4  background-image: -webkit-gradient(radial, 50% 50%,  
   0, 50% 50%, 350,  
5  from(#ffffff), to(#000000));  
6  background-image: -webkit-radial-gradient(50% 50%, 0,  
   50% 50%, 350,  
7  from(#ffffff), to(#000000));  
8  background-image: radial-gradient(50% 50%, farthest-  
   side,  
9  #ffffff, #000000); }
```

A radial gradient running from white in the center to black on the outside:

```
1  .somediv {  
2  background-image: -moz-radial-gradient(50% 50%,  
   farthest-side,  
3  #ffffff, #000000);  
4  background-image: -webkit-gradient(radial, 50% 50%,  
   0, 50% 50%, 350,  
5  from(#ffffff), to(#000000));  
6  background-image: -webkit-radial-gradient(50% 50%, 0,  
   50% 50%, 350,  
7  from(#ffffff), to(#000000));  
8  background-image: radial-gradient(50% 50%, farthest-  
   side,
```

```
9 | #ffffff, #000000); }
```

Fallback behavior: the browser will show the same behavior as it would for a missing image file (i.e. either the background or default color).



ColorZilla's [Ultimate CSS Gradient Generator](#) offers a familiar interface for generating gradients.

Ah, the good ol' Web 2.0 look, but using nothing but CSS. Thankfully, gradients have come a long way from being used for glossy buttons, and this CSS3 property is the latest step in that evolution.

Gradients are applied the way background images are, and there are a few ways to do it: hex codes, RGBA and HSLa.

Be careful when applying a background with a height of 100% to an element such as the body. In some browsers, this will limit the gradient to the edge of the visible page, and so you'll lose the gradient as you scroll down (and if you haven't specified a color, then the background will be

white). You can get around this by setting the `background-position` to `fixed`, which ensures that the background doesn't move as you scroll.

Specifying a background color not only is a good fallback practice but can prevent unforeseen problems. As a general rule, set it either to one end of the gradient or to a color in the middle of the range.

Legibility is also a consideration. Making text readable against a solid background color is easy. But if a gradient is dramatic (for example, from very light to very dark), then choose a text color that will work over the range of the gradient.

Radial gradients are a bit different, and getting used to the origins and spreads can take a bit of playing around. But the same principles apply. Note that Webkit browsers are switching from the `-webkit-gradient` property to `-webkit-linear-gradient` and `-webkit-radial-gradient`. To be safe, include both properties, but (as we have learned) put the old property before the new one.

These issues aren't new; we've been using gradients for ages. If you really need one, then the obvious fallback is simply to use an image. While it won't adapt to the dimensions of the container, you will be able to set its exact dimensions as you see fit.

Multiple Backgrounds

Support: Google Chrome 7.0+, Firefox 3.6+, Safari 2.0+, IE 9

Property: `background`

Example usage (multiple backgrounds separated by a comma, the first on top, the second behind it, the third behind them, and so on):

```
1 | .somediv {  
2 |   background: url('background1.jpg') top left no-  
   repeat,  
3 |   url('background2.jpg') bottom left repeat-y,  
4 |   url('background3.jpg') top right no-repeat; }
```

Fallback behavior: an unsupported browser will show only one image, the one on top (i.e. the first in the declaration).



The fantastic [Lost World's Fairs](#) website shows multiple backgrounds in its header and a solid color as a fallback.

Being able to set multiple background images is very useful. You can layer images on top of one another. And because CSS gradients can be applied as backgrounds, you can layer multiple images and gradients with ease.

You can also position background elements within dynamically sized containers. For example, you could have an image appear 25% down the container and then another at 75%, both of which move with any dynamic content.

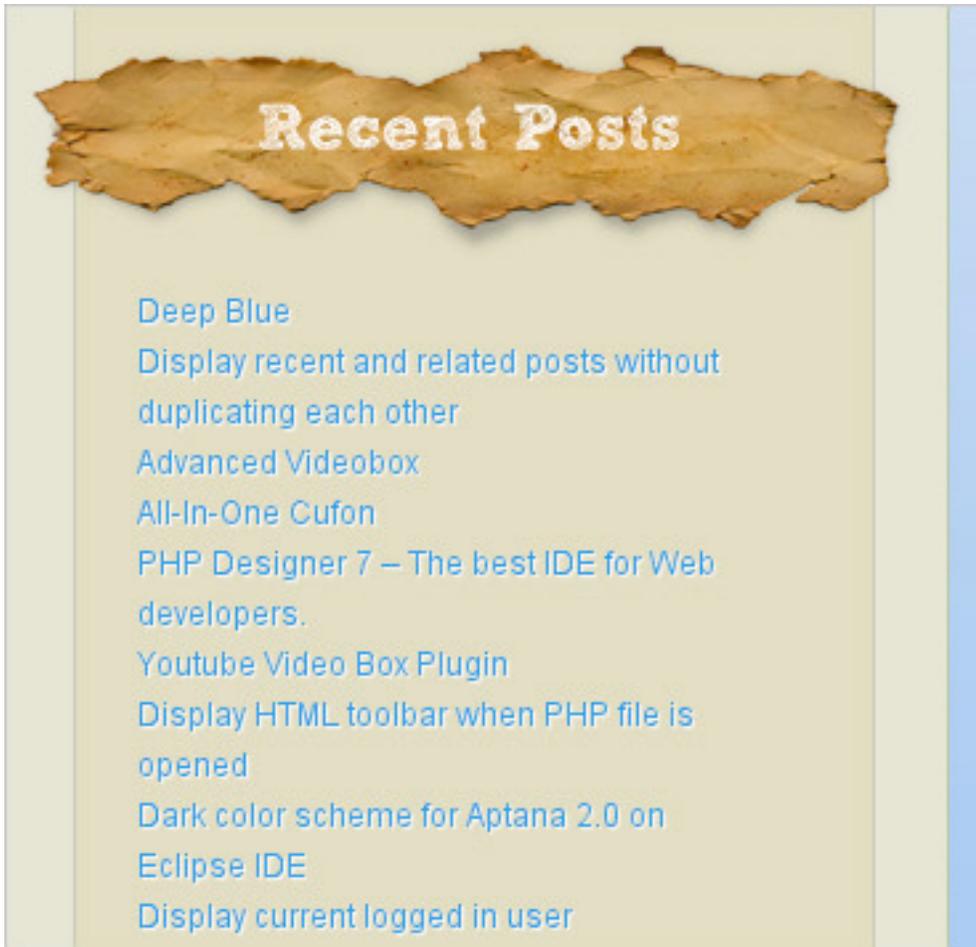
If multiple backgrounds are essential to your website, you could insert additional elements and images into the DOM using JavaScript. But again, is this worth doing? A single well-chosen background image might work best. It could be a matter of picking the most important image or blending the images into a composite (even if this makes for a less dynamic background).

Use Only Where Needed

It's worth repeating that CSS3 is not a necessity. Just because you can use CSS3 properties, doesn't mean your website would be any worse off without them. Applying these effects is now so simple, and so getting carried away becomes all too easy. Do you really need to round every corner or use multiple backgrounds everywhere? Just as a film can work without 3-D, so should your design be able to work without CSS3 splashed everywhere indiscriminately. The technology is simply a tool to make our lives easier and help us create better designs.

It is a testament to those who are already using CSS3 that there are very few instances of its misuse at the moment. The websites that do seem to misuse it suggest that their designers either used CSS3 for its own sake or didn't consider its implications on certain platforms.

In "[Web Design Trends 2010: Real-Life Metaphors and CSS3 Adaptation](#)," Smashing Magazine's Vitaly Friedman notes a number of misuses of the `text-shadow` property.



A less-than-ideal use of CSS3 on SramekDesign.com.

The `text-shadow` property has certainly become popular. One-pixel white shadows are popping up in text everywhere for no apparent reason. As Vitaly says:

... before adding a CSS3 feature to your website, make sure it is actually an enhancement, added for the purpose of aesthetics and usability, and not aesthetics at the cost of usability.

As you become familiar with CSS3's new properties, you will learn to recognize when and where problems occur and where the properties aren't really necessary.

Using CSS3

CSS3 is the future of the Web, no argument about that. So, versing yourself right now in the language of the future makes sense. While IE is still the single most popular browser, it now has less than half of the market share, meaning that the majority of people no longer use it and it can no longer be used as an excuse not to explore new possibilities and opportunities.

To use CSS3 means to embrace the principle that websites need not look the same in every browser, but rather should be tailored to the user's browsing preferences via sensible fallbacks. It isn't daunting or inaccessible, and the best way to learn is by diving in.

The Authors

Christian Krammer

Christian Krammer is Web designer at one of Austria's biggest newspapers called "[Kleine Zeitung](#)" where he worked for nearly ten years. He also is the proud owner of [css3files.com](#), a comprehensive website about CSS3. A large range of properties is explained there for you to learn from and reference. You can follow Christian on Twitter [@edge0703](#).

Dave Sparks

Dave Sparks is a Web designer and developer who has dabbled on the Web for over 10 years with more than six years of commercial experience. He is a part-timer who freelances and does work for [Armitage Online](#). He can be found writing about various topics at [Kamikazemusic.com](#) and tweeting as [@dsparks83](#). He also runs long distances, drinks lots of tea and spends time flying planes in his day job.

Harry Roberts

Harry Roberts is a Senior UI Developer for [Sky.com](#) and type nerd from the UK. Enthusiastic, passionate and often outspoken, he is a writer, designer and member of Smashing Magazine's Experts Panel. He tweets at [@csswizardry](#).

Kayla Knight

[Kayla Knight](#) is a full-time freelance Web designer and developer, and likes to blog a lot too. She also created and runs [Freelance Mingle](#), a social network for freelancers.

Inayaili de Leon

[Inayaili de León](#) is a London-based Portuguese web designer. When she's not designing sites or coding HTML and CSS, she is usually writing about it on her own [web design blog](#) or as a guest author on various sites. She works at Canonical and is pretty much incapable of going a day without at least 30 Twitter updates. Cats, chocolate, tea, pizza and pancakes are amongst the things that make her less dangerous on Monday morning commutes.

Louis Lazaris

Louis Lazaris is a freelance Web developer based in Toronto, Canada. He blogs about front-end code on [Impressive Webs](#) and is a coauthor of [HTML5 and CSS3 for the Real World](#), published by SitePoint. You can [follow Louis on Twitter](#) or contact him through his website.

Rachel Andrew

Rachel Andrew is a front and back-end Web developer and Director of [edgeofmyseat.com](#), a UK Web development consultancy and the creators of the small content management system, [Perch](#). She is the author of a number of Web design and development books including CSS Anthology: 101 Essential Tips, Tricks and Hacks (3rd edition), published by SitePoint and also writes on her blog [rachelandrew.co.uk](#). Rachel tries to encourage a common sense application of best practice and standards adoption in her own work and when writing about the web.

Richard Shepherd

Richard Shepherd ([@richardshepherd](#)) is a UK based Web designer and front-end developer. He loves to play with HTML5, CSS3, jQuery and WordPress, and currently works full-time bringing [VoucherCodes.co.uk](#) to life. He has an awesomeness factor of 8, and you can also find him at [www.richardshepherd.com](#).

Trent Walton

Trent Walton is founder and 1/3 of [Paravel Inc.](#), a custom Web design and development shop based out of the Texas Hill Country. When he's not working on client projects, he's probably writing & designing articles for [his blog](#), or contributing ideas for the next edition of [TheManyFacesOf.com](#).

Vitaly Friedman

Vitaly Friedman is editor-in-chief of [Smashing Magazine](#), an online magazine dedicated to designers and developers.

Zoe Mickley Gillenwater

Zoe Mickley Gillenwater is the author of the books [Stunning CSS3: A Project-based Guide to the Latest in CSS](#) and [Flexible Web Design: Creating Liquid and Elastic Layouts with CSS](#). She currently works on AT&T's design standards web team. Zoe is also a member of the Web Standards Project (WaSP) Adobe Task Force and was previously a moderator of the popular [css-discuss mailing list](#). Find out more about Zoe on her [blog and portfolio site](#).