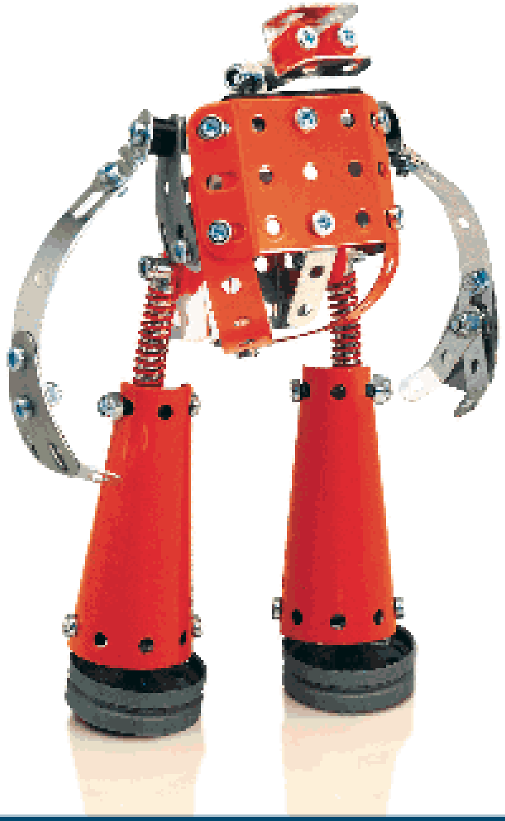
sitepoint®

HTML5 & CSS3 FOR THE REAL WORLD

BY ALEXIS GOLDSTEIN
LOUIS LAZARIS
ESTELLE WEYL



POWERFUL HTML5 AND CSS3 TECHNIQUES YOU CAN USE TODAY!

Something *seriously* cool has happened in web development!

Hi and welcome to the sample chapters of *HTML5 & CSS3 for the Real World*.

You might assume that the person writing this foreword is a designer or a front-end developer, at the forefront of these new technologies. Well, that's not the case. Let me reveal something about myself … I'm not a techie and I know nothing about code.

Yes, I work at SitePoint and have copied and pasted my way through creating a few HTML emails, but that's about as far as my skills go. So, you're thinking, "Why are you writing this if you don't know anything about HTML5 & CSS3?"

Well, let me tell you what I do know.

Last year, I started noticing lots of talk about these new technologies. I really didn't pay much attention. But the chatter continued, and shortly after we decided to start work on an HTML5 & CSS3 book due to the huge demand from our customers.

Now, it's not unusual for the people I work with to get excited about new books. But there was so much enthusiasm for this upcoming book and its topic, I had to find out why.

Over the next few months, I got involved in conversations around the office. When people spoke about all the cool new things you can do using HTML5 & CSS3, I listened. Most of it made no sense and my eyes glazed over, but there were a few times that I found myself thinking, "Hey, that sounds pretty awesome."

Then, last week, I asked our Sys Admin to do something for me and he needed to watch some instructional videos to get the job done, but they were not working. It had something to do with Flash.

The comment that left my mouth afterwards surprised me just as much as it surprised him—"Well, that won't be a concern for much longer since HTML5 allows for native video."

I still can't believe I said it. I was so proud of myself for knowing that. I've worked at SitePoint for a few years now and have watched new technologies come and go, but have yet to see anything quite like this.

HTML5 and CSS3 will change the way our designers and developers work for the better. One of our designers explained it like this: "We've been locked in, trying to get petty tasks done with so much effort. HTML5 & CSS3 allows me to finally do cool stuff and have some fun."

Well, that sounds pretty amazing to me. Read on and make up your own mind.

Regards,
Melinda Szasz

What's in This Excerpt

What's in the Rest of the Book

Chapter 3: More HTML5 Semantics

Continuing on from the previous chapter, we turn our attention to the new way in which HTML5 constructs document outlines. Then we look at a plethora of other semantic elements that let you be a little more expressive with your markup.

Chapter 4: HTML5 Forms

Some of the most useful and currently applicable features in HTML5 pertain to forms. A number of browsers now support native validation on email types like emails and URLs, and some browsers even support native date pickers, sliders, and spinner boxes. It's almost enough to make you enjoy coding forms! This chapter covers everything you need to know to be up to speed writing HTML5 forms, and provides scripted fallbacks for older browsers.

Chapter 5: HTML5 Audio and Video

HTML5 is often touted as a contender for the online multimedia content crown, long held by Flash. The new `audio` and `video` elements are the reason—they provide native, scriptable containers for your media without relying on a third-party plugin like Flash. In this chapter, you'll learn all the ins and outs of putting these new elements to work.

Chapter 6: Introducing CSS3

Now that we've covered just about all of HTML5, it's time to move onto its close relative CSS3. We'll start our tour of CSS3 by looking at some of the new selectors that let you target elements on the page with unprecedented flexibility. Then we'll follow up with a look at some new ways of specifying color in CSS3, including transparency. We'll close the chapter with a few quick wins—cool CSS3 features that can be added to your site with a minimum of work: text shadows, drop shadows, and rounded corners.

Chapter 8: CSS3 Transforms and Transitions

Animation has long been seen as the purview of JavaScript, but CSS3 lets you offload some of the heavy lifting to the browser. Transforms let you rotate, flip, skew, and otherwise throw your elements around. Transitions can add some subtlety to the otherwise jarring all-on or all-off state changes we see on our sites. We wrap up this chapter with a glimpse of the future; while CSS keyframe

Foreword

Heard of Sjoerd Visscher? I'd venture to guess you haven't, but what he considered a minor discovery is at the foundation of our ability to use HTML5 today.

Back in 2002, in The Hague, Netherlands, Mr. Visscher was attempting to improve the performance of his XSL output. He switched from `createElement` calls to setting the `innerHTML` property, and then realized that all the unknown, non-HTML elements were no longer able to be styled by CSS.

Fast forward to 2008, and HTML5 is gaining momentum. New elements have been specified, but in practice Internet Explorer versions 6-8 pose a problem, as they fail to recognize unknown elements; the new elements are unable to hold children and CSS has no effect on them. This depressing fact was posing quite a hindrance to HTML5 adoption.

Now, half a decade after his discovery, Sjoerd innocently mentions this trick in a comment on the blog of the W3C HTML Working Group co-chair, Sam Ruby: "BTW, if you want CSS rules to apply to unknown elements in IE, you just have to do `document.createElement(elementName)`. This somehow lets the CSS engine know that elements with that name exist."

Ian Hickson, lead editor of the HTML5 spec, was as surprised as the rest of the Web. Having never heard of this trick before, he was happy to report: "This piece of information makes building an HTML5 compatibility shim for IE7 far easier than had previously been assumed."

A day later, John Resig wrote the post that coined the term "HTML5 shiv." Here's a quick timeline of what followed:

- January 2009: Remy Sharp creates the first distributable script for enabling HTML5 element use in IE.
- June 2009: Faruk Ateş includes the HTML5 shiv in Modernizr's initial release.
- February 2010: A ragtag team of superstar JavaScript developers including Remy, Kangax, John-David Dalton, and PorneL collaborate and drop the file size of the script.

- March 2010: Mathias Bynens and others notice that the shiv doesn't affect pages printed from IE. It was a sad day. I issue an informal challenge to developers to find a solution.

- April 2010: Jonathan Neal answers that challenge with the IE Print Protector (IEPP), which captured the scope of the HTML5 shiv but added in support for printing the elements as well.

- April 2010: Remy replaces the legacy HTML5 shiv solution with the new IEPP.

- February 2011: Alexander Farkas carries the torch, moving the IEPP project to GitHub, adding a test suite, fixing bugs, and improving performance.

- April 2011: IEPP v2 comes out. Modernizr and the HTML5 shiv inherit the latest code while developers everywhere continue to use HTML5 elements in a cross-browser fashion without worry.

The tale of the HTML5 shiv is just one example of community contribution that helps to progress the open web movement. It's not just the W3C or the browsers who directly affect how we work on the Web, but people like you and me. I hope this book encourages you to contribute in a similar manner; the best way to further your craft is to actively share what you learn.

Adopting HTML5 and CSS3 today is easier than ever, and seriously fun. This book presents a wealth of practical information that gives you what you need to know to take advantage of HTML5 now. The authors—Alexis, Louis, and Estelle—are well-respected web developers who present a realistic learning curve for you to absorb the best practices of HTML5 development easily.

I trust this book is able to serve you well, and that you'll be as excited about the next generation of the Web as I am.

—

Paul Irish
jQuery Dev Relations,
Lead Developer of Modernizr and HTML5 Boilerplate
April 2011

Chapter 1

Introducing HTML5 and CSS3

This chapter gives a basic overview of how we arrived where we are today, why HTML5 and CSS3 are so important to modern websites and web apps, and how using these technologies will be invaluable to your future as a web professional.

Of course, if you'd prefer to just get into the meat of the project that we'll be building, and start learning how to use all the new bells and whistles that HTML5 and CSS3 bring to the table, you can always skip ahead to Chapter 2 and come back later.

What is HTML5?

What we understand today as HTML5 has had a relatively turbulent history. You probably already know that HTML is the predominant markup language used to describe content, or data, on the World Wide Web. HTML5 is the latest iteration of that markup language, and includes new features, improvements to existing features, and scripting-based APIs.

That said, HTML5 is not a reformulation of previous versions of the language—it includes all valid elements from both HTML4 and XHTML 1.0. Furthermore, it's been designed with some primary principles in mind to ensure it works on just about every platform, is compatible with older browsers, and handles errors gracefully. A summary of the design principles that guided the creation of HTML5 can be found on the W3C's HTML Design Principles page[1].

First and foremost, HTML5 includes redefinitions of existing markup elements, and new elements that allow web designers to be more expressive in the semantics of their markup. Why litter your page with `div`s when you can have `article`s, `section`s, `header`s, `footer`s, and more?

The term "HTML5" has additionally been used to refer to a number of other new technologies and APIs. Some of these include drawing with the `<canvas>` element, offline storage, the new `<video>` and `<audio>` elements, drag-and-drop functionality, Microdata, embedded fonts, and others. In this book, we'll be covering a number of those technologies, and more.

What's an API?

API stands for Application Programming Interface. Think of an API the same way you think of a graphical user interface—except that instead of being an interface for humans, it's an interface for your code. An API provides your code with a set of "buttons" (predefined methods) that it can press to elicit the desired behavior from the system, software library, or browser.

API-based commands are a way of abstracting the more complex stuff that's done in the background (or sometimes by third-party software). Some of the HTML5-related APIs will be introduced and discussed in later sections of this book.

Overall, you shouldn't be intimidated if you've had little experience with JavaScript or any scripting-related APIs. While it would certainly be beneficial to have some experience with JavaScript, it isn't mandatory.

Whatever the case, we'll walk you through the scripting parts of our book gradually, to ensure you're not left scratching your head!

It should also be noted that some of the technologies that were once part of HTML5 have been separated from the specification, so technically, they no longer fall under the "HTML5" umbrella. Certain other technologies were *never* part of HTML5, yet have at times been lumped in under the same label. This has instigated the use of

[1] http://www.w3.org/TR/html-design-principles/

broad, all-encompassing expressions such as "HTML5 and related technologies." Bruce Lawson even half-jokingly proposed the term "NEWT" (New Exciting Web Technologies)[2] as an alternative.

However, in the interest of brevity—and also at the risk of inciting heated arguments—we'll generally refer to these technologies collectively as "HTML5."

How did we get here?

The web design industry has evolved in a relatively short time period. Twelve years ago, a website that included images and an eye-catching design was considered "top of the line" in terms of web content.

Now, the landscape is quite different. Simple, performance-driven, Ajax-based web apps that rely on client-side scripting for critical functionality are becoming more and more common. Websites today often resemble standalone software applications, and an increasing number of developers are viewing them as such.

Along the way, web markup evolved. HTML4 eventually gave way to XHTML, which is really just HTML 4 with strict XML-style syntax. Currently, both HTML 4 and XHTML are in general use, but HTML5 is gaining headway.

HTML5 originally began as two different specifications: Web Forms 2.0 and Web Apps 1.0. Both were a result of the changed web landscape, and the need for faster, more efficient, maintainable web applications. Forms and app-like functionality are at the heart of web apps, so this was the natural direction for the HTML5 spec to take. Eventually, the two specs were merged to form what we now call HTML5.

During the time that HTML5 was in development, so was XHTML 2.0. That project has since been abandoned to allow focus on HTML5.

Would the real HTML5 spec please stand up?

Because the HTML5 specification is being developed by two different bodies (the WHATWG and the W3C), there are two different versions of the spec. The W3C (or World Wide Web Consortium) you're probably familiar with: it's the organization that maintains the original HTML and CSS specifications, as well as a host of other

[2] http://www.brucelawson.co.uk/2010/meet-newt-new-exciting-web-technologies/

web-related standards, such as SVG (scalable vector graphics) and WCAG (web content accessibility guidelines.)

The WHATWG (aka the Web Hypertext Application Technology Working Group), on the other hand, might be new to you. It was formed by a group of people from Apple, Mozilla, and Opera after a 2004 W3C meeting left them disheartened. They felt that the W3C was ignoring the needs of browser makers and users by focusing on XHTML 2.0, instead of working on a backwards-compatible HTML standard. So they went off on their own and developed the Web Apps and Web Forms specifications discussed above, which were then merged into a spec they called HTML5. On seeing this, the W3C eventually gave in and created its own HTML5 specification based on the WHATWG's spec.

This can seem a little confusing. Yes, there are some politics behind the scenes that we, as designers and developers, have no control over. But should it worry us that there are two versions of the spec? In short, no.

The WHATWG's version of the specification can be found at http://www.whatwg.org/html/, and has recently been renamed "HTML" (dropping the "5"). It's now called a "living standard," meaning that it will be in constant development and will no longer be referred to using incrementing version numbers.[3]

The WHATWG version contains information covering HTML-only features, including what's new in HTML5. Additionally, there are separate specifications being developed by the WHATWG that cover the related technologies. These specifications include Microdata, Canvas 2D Context, Web Workers, Web Storage, and others.[4]

The W3C's version of the spec can be found at http://dev.w3.org/html5/spec/, and the separate specifications for the other technologies can be accessed through http://dev.w3.org/html5/.

So what's the difference between the W3C spec and that of WHATWG? Briefly, the WHATWG version is a little more informal and experimental (and, some might argue, more forward-thinking). But overall, they're very similar, so either one can be used as a basis for studying new HTML5 elements and related technologies.

[3] See http://blog.whatwg.org/html-is-the-new-html5/ for an explanation of this change.
[4] For details, see http://wiki.whatwg.org/wiki/FAQ#What_are_the_various_versions_of_the_spec.3F.

Why should I care about HTML5?

As mentioned, at the core of HTML5 are a number of new semantic elements, as well as several related technologies and APIs. These additions and changes to the language have been introduced with the goal of web pages being easier to code, use, and access.

These new semantic elements, along with other standards like WAI-ARIA and Microdata (which we cover in Appendix B and Appendix C respectively), help make our documents more accessible to both humans and machines—resulting in benefits for both accessibility and search engine optimization.

The semantic elements, in particular, have been designed with the dynamic web in mind, with a particular focus on making pages more modular and portable. We'll go into more detail on this in later chapters.

Finally, the APIs associated with HTML5 help improve on a number of techniques that web developers have been using for years. Many common tasks are now simplified, putting more power in developers' hands. Furthermore, the introduction of HTML5-based audio and video means there will be less dependence on third-party software and plugins when publishing rich media content on the Web.

Overall, there is good reason to start looking into HTML5's new features and APIs, and we'll discuss more of those reasons as we go through this book.

What is CSS3?

Another separate—but no less important—part of creating web pages is Cascading Style Sheets (CSS). As you probably know, CSS is a style language that describes how HTML markup is presented or styled. CSS3 is the latest version of the CSS specification. The term "CSS3" is not just a reference to the new features in CSS, but the third level in the progress of the CSS specification.[5]

CSS3 contains just about everything that's included in CSS2.1 (the previous version of the spec). It also adds new features to help developers solve a number of problems without the need for non-semantic markup, complex scripting, or extra images.

[5] http://www.w3.org/Style/CSS/current-work.en.html

New features in CSS3 include support for additional selectors, drop shadows, rounded corners, multiple backgrounds, animation, transparency, and much more.

CSS3 is distinct from HTML5. In this publication, we'll be using the term CSS3 to refer to the third level of the CSS specification, with a particular focus on what's new in CSS3. Thus, CSS3 is separate from HTML5 and its related APIs.

Why should I care about CSS3?

Later in this book, we'll look in greater detail at what's new in CSS3. In the meantime, we'll give you a taste of why CSS3's new techniques are so exciting to web designers.

Some design techniques find their way into almost every project. Drop shadows, gradients, and rounded corners are three good examples. We see them everywhere. When used appropriately, and in harmony with a site's overall theme and purpose, these enhancements can make a design flourish.

Perhaps you're thinking: we've been creating these design elements using CSS for years now. But have we?

In the past, in order to create gradients, shadows, and rounded corners, web designers have had to resort to a number of tricky techniques. Sometimes extra HTML elements were required. In cases where the HTML is kept fairly clean, scripting hacks were required. In the case of gradients, the use of extra images was inevitable. We put up with these workarounds, because there was no other way of accomplishing those designs.

CSS3 allows you to include these and other design elements in a forward-thinking manner that leads to so many benefits: clean markup that is accessible to humans and machines, maintainable code, fewer extraneous images, and faster loading pages.

A Note on Vendor Prefixes

In order to use many of the new CSS3 features today, you'll be required to include quite a few extra lines of code. This is because browser vendors have implemented many of the new features in CSS3 using their own "prefixed" versions of a property. For example, to transform an element in Firefox, you need to use the `-moz-transform` property; to do the same in WebKit-based browsers such as Safari and Google Chrome, you have to use `-webkit-transform`. In some cases, you'll need up to four lines of code for a single CSS property. This can seem to nullify some of the benefits gained from avoiding hacks, images, and nonsemantic markup.

But browser vendors have implemented features this way for a good reason: the specifications are yet to be final, and early implementations tend to be buggy. So, for the moment, you provide values to current implementations using the vendor prefixes, and *also* provide a perennial version of each property using an unprefixed declaration. As the specs become finalized and the implementations refined, browser prefixes will eventually be dropped.

Even though it may seem like a lot of work to maintain code with all these prefixes, the benefits of using CSS3 today still outweigh the drawbacks. Despite having to change a number of prefixed properties just to alter one design element, maintaining a CSS3-based design is still easier than, say, making changes to background images through a graphics program, or dealing with the drawbacks of extra markup and hacky scripts. And, as we have mentioned, your code is much less likely to become outdated or obsolete.

What do we mean by the "real world"?

In the real world, we don't create a website and then move on to the next project while leaving previous work behind. We create web applications and we update them, fine-tune them, test them for potential performance problems, and continually tweak their design, layout, and content.

In other words, in the real world, we don't write code that we have no intention of revisiting. We write code using the most reliable, maintainable, and effective methods available, with every intention of returning to work on that code again to make any necessary improvements or alterations. This is evident not only in websites and web apps that we build and maintain on our own, but also in those we create and maintain for our clients.

We need to continually search out new and better ways to write our code. HTML5 and CSS3 are a big step in that direction.

The Varied Browser Market

Although HTML5 is still in development, and does present significant changes in the way content is marked up, it's worth noting that those changes won't cause older browsers to choke, or result in layout problems or page errors.

What this means is that you could take any of your current projects containing valid HTML4 or XHTML markup, change the doctype to HTML5 (which we'll cover in Chapter 2), and the page will still validate and appear the same as it did before. The changes and additions in HTML5 have been implemented into the language in such a way so as to ensure backwards compatibility with older browsers—even IE6!

But that's just the markup. What about all the other features of HTML5, CSS3, and related technologies? According to one set of statistics,[6] about 47% of users are on a version of Internet Explorer that has no support for most of these new features.

As a result, developers have come up with various solutions to provide the equivalent experience to those users, all while embracing the exciting new possibilities offered by HTML5 and CSS3. Sometimes this is as simple as providing fallback content, like a Flash video player to browsers without native video support. At other times, though, it's been necessary to use scripting to mimic support for new features. These "gap-filling" techniques are referred to as **polyfills**. Relying on scripts to emulate native features isn't always the best approach when building high-performance web apps, but it's a necessary growing pain as we evolve to include new enhancements and features, such as the ones we'll be discussing in this book.

So, while we'll be recommending fallback options and polyfills to plug the gaps in browser incompatibilities, we'll also try to do our best in warning you of potential drawbacks and pitfalls associated with using these options.

Of course, it's worth noting that sometimes no fallbacks or polyfills are required at all: for example, when using CSS3 to create rounded corners on boxes in your design, there's often no harm in users of older browsers seeing square boxes instead. The

[6] http://gs.statcounter.com/#browser_version-ww-monthly-201011-201101-bar

functionality of the site isn't degraded, and those users will be none the wiser about what they're missing.

With all this talk of limited browser support, you might be feeling discouraged. Don't be! The good news is that more than 40% of worldwide users are on a browser that *does* offer support for a lot of the new stuff we'll discuss in this book. And this support is growing all the time, with new browser versions (such as Internet Explorer 9) continuing to add support for many of these new features and technologies.

As we progress through the lessons, we'll be sure to inform you where support is lacking, so you'll know how much of what you create will be visible to your audience in all its HTML5 and CSS3 glory. We'll also discuss ways you can ensure that nonsupporting browsers have an acceptable experience, even without all the bells and whistles that come with HTML5 and CSS3.

The Growing Mobile Market

Another compelling reason to start learning and using HTML5 and CSS3 today is the exploding mobile market.

According to StatCounter, in 2009, just over 1% of all web usage was mobile.[7] In less than two years, that number has quadrupled to over 4%.[8] Some reports have those numbers even higher, depending on the kind of analysis being done. Whatever the case, it's clear that the mobile market is growing at an amazing rate.

4% of total usage may seem small, and in all fairness, it is. But it's the growth rate that makes that number so significant—400% in two years! So what does this mean for those learning HTML5 and CSS3?

HTML5, CSS3, and related cutting-edge technologies are very well supported in many mobile web browsers. For example, mobile Safari on iOS devices like the iPhone and iPad, Opera Mini and Opera Mobile, as well as the Android operating system's web browser all provide strong levels of HTML5 and CSS3 support. New features and technologies supported by some of those browsers include CSS3 colors and opacity, the Canvas API, Web Storage, SVG, CSS3 rounded corners, Offline Web Apps, and more.

[7] http://gs.statcounter.com/#mobile_vs_desktop-ww-monthly-200901-200912-bar
[8] http://gs.statcounter.com/#mobile_vs_desktop-ww-monthly-201011-201101-bar

In fact, some of the new technologies we'll be introducing in this book have been specifically designed with mobile devices in mind. Technologies like Offline Web Apps and Web Storage have been designed, in part, because of the growing number of people accessing web pages with mobile devices. Such devices can often have limitations with online data usage, and thus benefit greatly from the ability to access web applications offline.

We'll be touching on those subjects in Chapter 10, as well as others throughout the course of the book that will provide the tools you need to create web pages for a variety of devices and platforms.

On to the Real Stuff

It's unrealistic to push ahead into new technologies and expect to author pages and apps for only one level of browser. In the real world, and in a world where we desire HTML5 and CSS3 to make further inroads, we need to be prepared to develop pages that work across a varied landscape. That landscape includes modern browsers, older versions of Internet Explorer, and an exploding market of mobile devices.

Yes, in some ways, supplying a different set of instructions for different user agents resembles the early days of the Web with its messy browser sniffing and code forking. But this time around, the new code is future-proof, so that when the older browsers fall out of general use, all you need to do is remove the fallbacks and polyfills, leaving only the code base that's aimed at modern browsers.

HTML5 and CSS3 are the leading technologies ushering in a much more exciting world of web page authoring. Because all modern browsers (including IE9) provide significant levels of support for a number of HTML5 and CSS3 features, creating powerful, easy-to-maintain, future-proof web pages is more accessible to web developers than ever before.

As the market share of older browsers declines, the skills you gain today in understanding HTML5 and CSS3 will become that much more valuable. By learning these technologies today, you're preparing for a bright future in web design. So, enough about the "why," let's start digging into the "how"!

Chapter 2

Markup, HTML5 Style

Now that we've given you a bit of a history primer, along with some compelling reasons to learn HTML5 and start using it in your projects today, it's time to introduce you to the sample site that we'll be progressively building in this book.

After we briefly cover what we'll be building, we'll discuss some HTML5 syntax basics, along with some suggestions for best practice coding. We'll follow that with some important info on cross-browser compatibility, and the basics of page structure in HTML5. Lastly, we'll introduce some specific HTML5 elements and see how they'll fit into our layout.

So let's get into it!

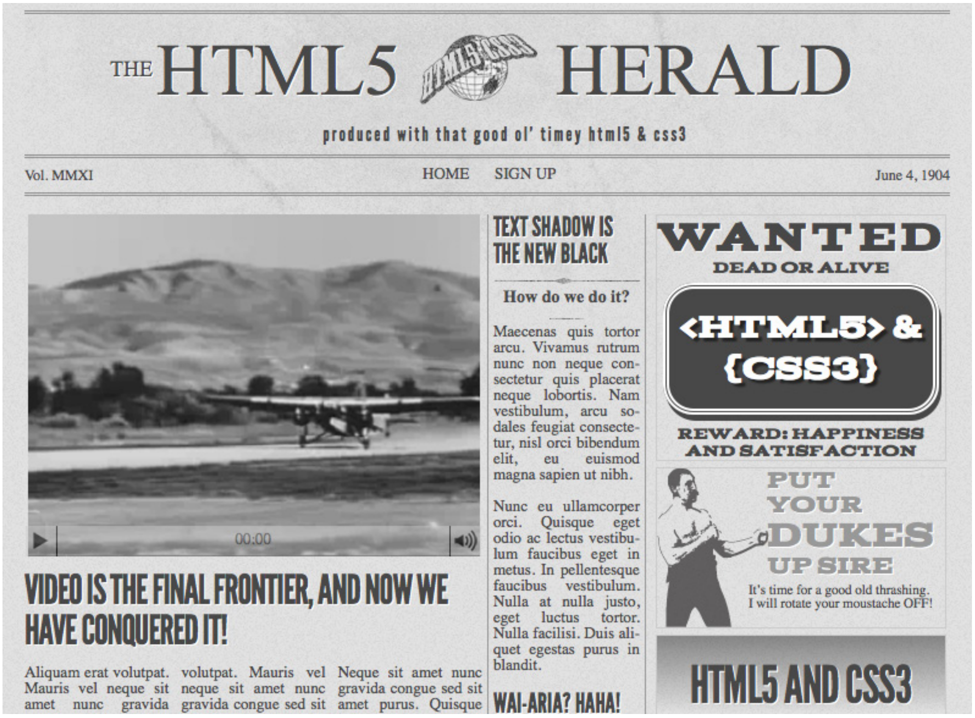
Introducing *The HTML5 Herald*

For the purpose of this book, we've put together a sample website project that we'll be building from scratch.

The website is already built—check it out now at http://thehtml5herald.com/. It's an old-time newspaper-style website called *The HTML5 Herald*. The home page of the site contains some media in the form of video, images, articles, and advertisements. There's also another page comprising a registration form.

Go ahead and view the source, and try some of the functionality if you like. As we proceed through the book, we'll be working through the code that went into making the site. We'll avoid discussing every detail of the CSS involved, as most of it should already be familiar to you: float layouts, absolute and relative positioning, basic font styling, and the like. We'll primarily focus on the new HTML5 elements, along with the APIs, plus all the new CSS3 techniques being used to add styles and interactivity to the various elements.

Figure 2.1 shows a bit of what the finished product looks like.


THE HTML5 HERALD

produced with that good ol' timey html5 & css3

Vol. MMXI HOME SIGN UP June 4, 1904

00:00

VIDEO IS THE FINAL FRONTIER, AND NOW WE HAVE CONQUERED IT!

Aliquam erat volutpat. Mauris vel neque sit amet nunc gravida

volutpat. Mauris vel neque sit amet nunc gravida congue sed sit

Neque sit amet nunc gravida congue sed sit amet purus. Quisque

TEXT SHADOW IS THE NEW BLACK

How do we do it?

Maecenas quis tortor arcu. Vivamus rutrum nunc non neque consectetur quis placerat neque lobortis. Nam vestibulum, arcu sodales feugiat consectetur, nisl orci bibendum elit, eu euismod magna sapien ut nibh.

Nunc eu ullamcorper orci. Quisque eget odio ac lectus vestibulum faucibus eget in metus. In pellentesque faucibus vestibulum. Nulla at nulla justo, eget luctus tortor. Nulla facilisi. Duis aliquet egestas purus in blandit.

WAI-ARIA? HAHA!

WANTED

DEAD OR ALIVE

<HTML5> & {CSS3}

REWARD: HAPPINESS AND SATISFACTION

PUT YOUR DUKES UP SIRE

It's time for a good old thrashing.
I will rotate your moustache OFF!

HTML5 AND CSS3

Figure 2.1. The front page of *The HTML5 Herald*

While we build the site, we'll do our best to explain the new HTML5 elements, APIs, and CSS3 features, and we'll try to recommend some best practices. Of course, many of these technologies are still new and in development, so we'll try not to be too dogmatic about what you can and can't do.

A Basic HTML5 Template

As you learn HTML5 and add new techniques to your toolbox, you're likely going to want to build yourself a blueprint, or boilerplate, from which you can begin all your HTML5-based projects. In fact, you've probably already done something similar for your existing XHTML or HTML 4.0 projects. We encourage this, and you may also consider using one of the many online sources that provide a basic HTML5 starting point for you.[1]

In this project, however, we want to build our code from scratch and explain each piece as we go along. Of course, it would be impossible for even the most fantastical and unwieldy sample site we could dream up to include *every* new element or technique, so we'll also explain some new features that don't fit into the project. This way, you'll be familiar with a wide set of options when deciding how to build your HTML5 and CSS3 websites and web apps, so you'll be able to use this book as a quick reference for a number of techniques.

Let's start simple, with a bare-bones HTML5 page:

index.html *(excerpt)*

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">

  <title>The HTML5 Herald</title>
  <meta name="description" content="The HTML5 Herald">
  <meta name="author" content="SitePoint">

  <link rel="stylesheet" href="css/styles.css?v=1.0">

  <!--[if lt IE 9]>
  <script src="http://html5shiv.googlecode.com/svn/trunk/html5.js">
➥</script>
  <![endif]-->
</head>
<body>
```

[1] A few you might want to look into can be found at http://www.html5boilerplate.com/ and http://html5reset.org/.

```
  <script src="js/scripts.js"></script>
</body>
</html>
```

Look closely at the above markup. If you're making the transition to HTML5 from XHTML or HTML 4, then you'll immediately notice quite a few areas in which HTML5 differs.

The Doctype

First, we have the Document Type Declaration, or **doctype**. This is simply a way to tell the browser—or any other parsers—what type of document they're looking at. In the case of HTML files, it means the specific version and flavor of HTML. The doctype should always be the first item at the top of all your HTML files. In the past, the doctype declaration was an ugly and hard-to-remember mess. For XHTML 1.0 Strict:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

And for HTML4 Transitional:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
```

Over the years, code editing software began to provide HTML templates with the doctype already included, or else they offered a way to automatically insert one. And naturally, a quick web search will easily bring up the code to insert whatever doctype you require.

Although having that long string of text at the top of our documents hasn't really hurt us (other than forcing our sites' viewers to download a few extra bytes), HTML5 has done away with that indecipherable eyesore. Now all you need is this:

```
 <!doctype html>
```

Simple, and to the point. You'll notice that the "5" is conspicuously missing from the declaration. Although the current iteration of web markup is known as "HTML5," it really is just an evolution of previous HTML standards—and future specifications will simply be a development of what we have today. Because browsers have to support all existing content on the Web, there's no reliance on the doctype to tell them which features should be supported in a given document.

The `html` Element

Next up in any HTML document is the `html` element, which has not changed significantly with HTML5. In our example, we've included the `lang` attribute with a value of `en`, which specifies that the document is in English. In XHTML-based syntax, you'd be required to include an `xmlns` attribute. In HTML5, this is no longer needed, and even the `lang` attribute is unnecessary for the document to validate or function correctly.

So here's what we have so far, including the closing `</html>` tag:

```
<!doctype html>
<html lang="en">

</html>
```

The `head` Element

The next part of our page is the `<head>` section. The first line inside the `head` is the one that defines the character encoding for the document. This is another element that's been simplified. Here's how you used to do this:

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
```

HTML5 improves on this by reducing the character encoding `<meta>` tag to the bare minimum:

```
<meta charset="utf-8">
```

In nearly all cases, `utf-8` is the value you'll be using in your documents. A full explanation of character encoding is beyond the scope of this chapter, and it probably won't be that interesting to you, either. Nonetheless, if you want to delve a little deeper, you can read up on the topic on the W3C's site.[2]

[2] http://www.w3.org/TR/html-markup/syntax.html#character-encoding

Get In Early

To ensure that all browsers read the character encoding correctly, the entire character encoding declaration must be included somewhere within the first 512 characters of your document. It should also appear before any content-based elements (like the `<title>` element that follows it in our example site).

There's much more we could write about this subject, but we want to keep you awake—so we'll spare you those details! For now, we're content to accept this simplified declaration and move on to the next part of our document:

```
<title>The HTML5 Herald</title>
<meta name="description" content="The HTML5 Herald">
<meta name="author" content="SitePoint">

<link rel="stylesheet" href="css/styles.css?v=1.0">
```

In these lines, HTML5 barely differs from previous syntaxes. The page title is declared the same as it always was, and the `<meta>` tags we've included are merely optional examples to indicate where these would be placed; you could put as many `meta` elements here as you like.

The key part of this chunk of markup is the stylesheet, which is included using the customary `link` element. At first glance, you probably didn't notice anything different. But customarily, `link` elements would include a `type` attribute with a value of `text/css`. Interestingly, this was never required in XHTML or HTML 4—even when using the Strict doctypes. HTML5-based syntax encourages you to drop the `type` attribute completely, since all browsers recognize the content type of linked stylesheets without requiring the extra attribute.

Leveling the Playing Field

The next element in our markup requires a bit of background information before it can be introduced.

HTML5 includes a number of new elements, such as `article` and `section`, which we'll be covering later on. You might think this would be a major problem for older browsers, but you'd be wrong. This is because the majority of browsers don't actually care what tags you use. If you had an HTML document with a `<recipe>` tag (or even a `<ziggy>` tag) in it, and your CSS attached some styles to that element, nearly every browser would proceed as if this were totally normal, applying your styling without complaint.

Of course, this hypothetical document would fail to validate, but it *would* render correctly in *almost* all browsers—the exception being Internet Explorer. Prior to version 9, IE prevented unrecognized elements from receiving styling. These mystery elements were seen by the rendering engine as "unknown elements," so you were unable to change the way they looked or behaved. This includes not only our imagined elements, but also any elements which had yet to be defined at the time those browser versions were developed. That means (you guessed it) the new HTML5 elements.

At the time of writing, Internet Explorer 9 has only just been released (and adoption will be slow), so this is a bit of a problem. We want to start using the shiny new tags, but if we're unable to attach any CSS rules to them, our designs will fall apart.

Fortunately, there's a solution: a very simple piece of JavaScript, originally developed by John Resig, can magically make the new HTML5 elements visible to older versions of IE.

We've included this so-called "HTML5 shiv"[3] in our markup as a `<script>` tag surrounded by **conditional comments**. Conditional comments are a proprietary feature implemented by Microsoft in Internet Explorer. They provide you with the ability to target specific versions of that browser with scripts or styles.[4] This conditional comment is telling the browser that the enclosed markup should only appear to users viewing the page with Internet Explorer prior to version 9:

```
<!--[if lt IE 9]>
<script src="http://html5shiv.googlecode.com/svn/trunk/html5.js">
➥</script>
<![endif]-->
```

It should be noted that if you're using a JavaScript library that deals with HTML5 features or the new APIs, it's possible that it will already have the HTML5 enabling

[3] You might be more familiar with its alternative name: the HTML5 shim. Whilst there are identical code snippets out there that go by both names, we'll be referring to all instances as the HTML5 shiv, its original name.

[4] For more information see http://reference.sitepoint.com/css/conditionalcomments

script present; in this case, you could remove reference to Remy Sharp's script. One example of this would be Modernizr,[5] a JavaScript library that detects modern HTML and CSS features—and which we cover in Appendix A. Modernizr includes code that enables the HTML5 elements in older versions of IE, so Remy's script would be redundant.

What about users on IE 6–8 who have JavaScript disabled?

Of course, there's still a group of users who won't benefit from Remy's HTML5 shiv: those who have, for one reason or another, disabled JavaScript. As web designers, we're constantly told that the content of our sites should be fully accessible to all users, even those without JavaScript. When between 40% and 75% of your audience uses Internet Explorer, this can seem like a serious concern.

But it's not as bad as it seems. A number of studies have shown that the number of users that have JavaScript disabled is low enough to be of little concern.

In one study[6] conducted on the Yahoo network, published in October 2010, users with JavaScript disabled amounted to around 1% of total traffic worldwide. Another study[7] indicated a similar number across a billion visitors. In both studies, the US had the highest number of visitors with JavaScript disabled in comparison to other parts of the world.

There *are* ways to use HTML5's new elements without requiring JavaScript for the elements to appear styled in nonsupporting browsers. Unfortunately, those methods are rather impractical and have other drawbacks.

If you're still concerned about these users, it might be worth considering a hybrid approach; for example, use the new HTML5 elements where the lack of styles won't be overly problematic, while relying on traditional elements like `div`s for key layout containers.

The Rest is History

Looking at the rest of our starting template, we have the usual `body` element along with its closing tag and the closing `</html>` tag. We also have a reference to a JavaScript file inside a `script` element.

[5] http://www.modernizr.com/

[6] http://developer.yahoo.com/blogs/ydn/posts/2010/10/how-many-users-have-javascript-disabled/

[7] http://visualrevenue.com/blog/2007/08/eu-and-us-javascript-disabled-index.html

Much like the `link` element discussed earlier, the `<script>` tag does not require that you declare a `type` attribute. In XHTML, to validate a page that contains external scripts, your `<script>` tag should look like this:

```
<script src="js/scripts.js" type="text/javascript"></script>
```

Since JavaScript is, for all practical purposes, the only real scripting language used on the Web, and since all browsers will assume that you're using JavaScript even when you don't explicitly declare that fact, the `type` attribute is unnecessary in HTML5 documents:

```
<script src="js/scripts.js"></script>
```

We've put the `script` element at the bottom of our page to conform to best practices for embedding JavaScript. This has to do with the page loading speed; when a browser encounters a script, it will pause downloading and rendering the rest of the page while it parses the script. This results in the page appearing to load much more slowly when large scripts are included at the top of the page before any content. This is why most scripts should be placed at the very bottom of the page, so that they'll only be parsed after the rest of the page has loaded.

In some cases (like the HTML5 shiv) the script may *need* to be placed in the `head` of your document, because you want it to take effect before the browser starts rendering the page.

HTML5 FAQ

After this quick introduction to HTML5 markup, you probably have a bunch of questions swirling inside your head. Here are some answers to a few of the likely ones.

Why do these changes still work in older browsers?

This is what a lot of developers seem to have trouble accepting. To understand why this isn't a problem, we can compare HTML5 to some of the new features added in CSS3, which we'll be discussing in later chapters.

In CSS, when a new feature is added (for example, the `border-radius` property that adds rounded corners to elements), it also has to be added to browsers' rendering engines, so older browsers won't recognize it. So if a user is viewing the page on a browser with no support for `border-radius`, the rounded corners will appear square. Other CSS3 features behave similarly, causing the experience to be degraded to some degree.

Many developers expect that HTML5 will work in a similar way. While this might be true for some of the advanced features and APIs we'll be considering later in the book, it's not the case with the changes we've covered so far; that is, the simpler syntax, the reduced redundancies, and the new doctype.

HTML5's syntax was defined after a careful study of what older browsers can and can't handle. For example, the 15 characters that comprise the doctype declaration in HTML5 are the minimum characters required to get every browser to display a page in standards mode.

Likewise, while XHTML required a lengthier character-encoding declaration and an extra attribute on the `html` element for the purpose of validation, browsers never required them in order to display a page correctly. Again, the behavior of older browsers was carefully examined, and it was determined that the character encoding could be simplified and the `xmlns` attribute be removed—and browsers would still see the page the same way.

The simplified `script` and `link` elements also fall into this category of "simplifying without breaking older pages." The same goes for the Boolean attributes we saw above; browsers have always ignored the values of attributes like `checked` and `disabled`, so why insist on providing them?

Thus, as mentioned in Chapter 1, you shouldn't be afraid to use HTML5 today. The language was designed with backwards compatibility in mind, with the goal of trying to support as much existing content as possible.

Unlike changes to CSS3 and JavaScript, where additions are only supported when browser makers actually implement them, there's no need to wait for new browser versions to be released before using HTML5's syntax. And when it comes to using the new semantic elements, a small snippet of JavaScript is all that's required to bring older browsers into line.

What is standards mode?

When standards-based web design was in its infancy, browser makers were faced with a problem: supporting emerging standards would, in many cases, break backwards compatibility with existing web pages that were designed to older, nonstandard browser implementations. Browser makers needed a signal indicating that a given page was meant to be rendered according to the standards. They found such a signal in the doctype: new, standards-compliant pages included a correctly formatted doctype, while older, nonstandard pages generally didn't.

Using the doctype as a signal, browsers could switch between **standards mode** (in which they try to follow standards to the letter in the way they render elements) and **quirks mode** (where they attempt to mimic the "quirky" rendering capabilities of older browsers to ensure that the page renders how it was intended).

It's safe to say that in the current development landscape, nearly every web page has a proper doctype, and thus will render in standards mode; it's therefore unlikely that you'll ever have to deal with a page being rendered in quirks mode. Of course, if a user is viewing a web page using a very old browser (like IE4), the page will be rendered using that browser's rendering mode. This is what quirks mode mimics, and it will do so regardless of the doctype being used.

Although the XHTML and older HTML doctypes include information about the exact version of the specification they refer to, browsers have never actually made use of that information. As long as a seemingly correct doctype is present, they'll render the page in standards mode. Consequently, HTML5's doctype has been stripped down to the bare minimum required to trigger standards mode in any browser.

Further information, along with a chart that outlines what will cause different browsers to render in quirks mode, can be found on Wikipedia.[8] You can also read a good overview of standards and quirks mode on SitePoint's CSS reference.[9]

Shouldn't all tags be closed?

In XHTML-based syntax, all elements need to be closed—either with a corresponding closing tag (like `</html>`) or in the case of **void** elements, a forward slash at the end

[8] http://en.wikipedia.org/wiki/Quirks_mode/

[9] http://reference.sitepoint.com/css/doctypesniffing/

of the tag. The latter are elements that *can't* contain child elements (such as `input`, `img`, or `link`).

You can still use that style of syntax in HTML5—and you might prefer it for consistency and maintainability reasons—but it's no longer required to add a trailing slash to void elements for validation. Continuing with the theme of "cutting the fat," HTML5 allows you to omit the trailing slash from such elements, arguably leaving your markup cleaner and less cluttered.

It's worth noting that in HTML5, most elements that *can* contain nested elements —but simply happen to be empty—still need to be paired with a corresponding closing tag. There are exceptions to this rule, but it's simpler to assume that it's universal.

What about other XHTML-based syntax customs?

While we're on the subject, omitting closing slashes is just one aspect of HTML5-based syntax that differs from XHTML. In fact, syntax style issues are completely ignored by the HTML5 validator, which will only throw errors for code mistakes that threaten to disrupt your document in some way.

What this means is that through the eyes of the validator, the following five lines of markup are identical:

```
<link rel="stylesheet" href="css/styles.css" />
<link rel="stylesheet" href="css/styles.css">
<LINK REL="stylesheet" HREF="css/styles.css">
<Link Rel="stylesheet" Href="css/styles.css">
<link rel=stylesheet href=css/styles.css>
```

In HTML5, you can use lowercase, uppercase, or mixed-case tag names or attributes, as well as quoted or unquoted attribute values (as long as those values don't contain spaces or other reserved characters)—and it will all validate just fine.

In XHTML, all attributes have to have values, even if those values are redundant. For example, you'd often see markup like this:

```
<input type="text" disabled="disabled" />
```

In HTML5, attributes that are either "on" or "off" (called **Boolean attributes**) can simply be specified with no value. So, the above `input` element could now be written as follows:

```
<input type="text" disabled>
```

Hence, HTML5 has much looser requirements for validation, at least as far as syntax is concerned. Does this mean you should just go nuts and use whatever syntax you want on any given element? No, we certainly don't recommend that.

We encourage developers to choose a syntax style and stick to it—especially if you are working in a team, where code maintenance and readability are crucial. We also recommend (though this is certainly not required) that you choose a minimalist coding style while staying consistent.

Here are some guidelines that you can consider using:

- Use lowercase for all elements and attributes, as you would in XHTML.
- Despite some elements not requiring closing tags, we recommend that all elements that contain content be closed (as in `<p>Text</p>`).
- Although you can leave attribute values unquoted, it's highly likely that you'll have attributes that require quotes (for example, when declaring multiple classes separated by spaces, or when appending a query string value to a URL). As a result, we suggest that you always use quotes for the sake of consistency.
- Omit the trailing slash from elements that have no content (like `meta` or `input`).
- Avoid providing redundant values for Boolean attributes (for instance, use `<input type="checkbox" checked>` rather than `<input type="checkbox" checked="checked">`).

Again, the recommendations above are by no means universally accepted. But we believe they're reasonable syntax suggestions for achieving clean, easy-to-read maintainable markup.

If you run amok with your code style, including too much that's unnecessary, you run the risk of negating the strides taken by the creators of HTML5 in trying to simplify the language.

Defining the Page's Structure

Now that we've broken down the basics of our template, let's start adding some meat to the bones, and give our page some basic structure.

Later in the book, we're going to specifically deal with adding CSS3 features and other HTML5 goodness; for now, we'll consider what elements we want to use in building our site's overall layout. We'll be covering a lot in this section, and throughout the coming chapters, about **semantics**. When we use this term, we're referring to the way a given HTML element describes the meaning of its content. Because HTML5 includes a wider array of semantic elements, you might find yourself spending a bit more time thinking about your content's structure and meaning than you've done in the past with HTML 4 or XHTML. That's great! Understanding what your content *means* is what writing good markup is all about.

If you look back at the screenshot of *The HTML5 Herald* (or view the site online), you'll see that it's divided up as follows:

- header section with a logo and title
- navigation bar
- body content divided into three columns
- articles and ad blocks within the columns
- footer containing some author and copyright information

Before we decide which elements are appropriate for these different parts of our page, let's consider some of our options. First of all, we'll introduce you to some of the new HTML5 semantic elements that could be used to help divide our page up and add more meaning to our document's structure.

The `header` Element

Naturally, the first element we'll look at is the `header` element. The WHATWG spec describes it succinctly as "a group of introductory or navigational aids."[10] Essentially, this means that whatever content you were accustomed to including inside of `<div id="header">`, you would now include in the `header`.

[10] http://www.whatwg.org/specs/web-apps/current-work/multipage/sections.html#the-header-element

But there's a catch that differentiates `header` from the customary `div` element that's often used for a site's header: there's no restriction to using it just once per page. Instead, you can include a new `header` element to introduce each section of your content. When we use the word "section" here, we're not limiting ourselves to the actual `section` element described below; technically, we're referring to what HTML5 calls "sectioning content." This will be covered in greater detail in the next chapter; for now, you can safely understand it to mean any chunk of content that might need its own header.

A `header` element can be used to include introductory content or navigational aids that are specific to any single section of a page, or that apply to the entire page—or both.

While a `header` element will frequently be placed at the top of a page or section, its definition is independent from its position. Your site's layout might call for the title of an article or blog post to be off to the left, right, or even below the content; regardless, you can still use `header` to describe this content.

The `section` Element

The next element you should become familiar with is HTML5's `section` element. The WHATWG spec defines `section` as follows:[11]

> The `section` element represents a generic section of a document or application. A section, in this context, is a thematic grouping of content, typically with a heading.

It further explains that a `section` shouldn't be used as a generic container that exists for styling or scripting purposes only. If you're unable to use `section` as a generic container—for example, in order to achieve your desired CSS layout—then what *should* you use? Our old friend, the `div`—which is semantically meaningless.

Going back to the definition from the spec, the `section` element's content should be "thematic," so it would be incorrect to use it in a generic way to wrap unrelated pieces of content.

[11] http://www.whatwg.org/specs/web-apps/current-work/multipage/sections.html#the-section-element

Some examples of acceptable uses for `section` elements include:

- individual sections of a tabbed interface
- segments of an "About" page; for example, a company's "About" page might include sections on the company's history, its mission statement, and its team
- different parts of a lengthy "terms of service" page
- various sections of an online news site; for example, articles could be grouped into `section`s covering sports, world affairs, and economic news

Semantics!

Every time new semantic markup is made available to web designers, there will be debate over what constitutes correct use of these elements, what the spec's intention was, and so on. You may remember discussions about the appropriate use of the `dl` element in previous HTML specifications. Unsurprisingly, therefore, HTML5 has not been immune to this phenomenon—particularly when it comes to the `section` element.

Even Bruce Lawson, a well-respected authority on HTML5, has admitted to using `section` incorrectly in the past. For a bit of clarity, Bruce's post[12] explaining his error is well worth the read. In short:

- `section` is *generic*, so if a more specific semantic element is appropriate (like `article`, `aside`, or `nav`), use that instead.
- `section` *has semantic meaning*; it implies that the content it contains is related in some way. If you're unable to succinctly describe all the content you're trying to put in a `section` using just a few words, it's likely you need a semantically neutral container instead: the humble `div`.

That said, as is always the case with semantics, it's open to interpretation in some instances. If you feel you can make a case for why you're using a given element rather than another, go for it. In the unlikely event that anyone ever calls you on it, the resulting discussion can be both entertaining and enriching for everyone involved, and might even contribute to the wider community's understanding of the specification.

[12] http://html5doctor.com/the-section-element/

Keep in mind, also, that you're permitted to nest `section` elements inside existing `section` elements, if it's appropriate. For example, for an online news website, the world news `section` might be further subdivided into a `section` for each major global region.

The `article` Element

The `article` element is similar to the `section` element, but there are some notable differences. Here's the definition according to WHATWG:[13]

> The article element represents a self-contained composition in a document, page, application, or site and that is, in principle, independently distributable or reusable, e.g. in syndication.

The key terms in that definition are *self-contained composition* and *independently distributable*. Whereas a `section` can contain any content that can be grouped thematically, an `article` must be a single piece of content that can stand on its own. This distinction can be hard to wrap your head around—so when in doubt, try the test of syndication: if a piece of content can be republished on another site without being modified, or pushed out as an update via RSS, or on social media sites like Twitter or Facebook, it has the makings of an `article`.

Ultimately, it's up to you to decide what constitutes an `article`, but here are some suggestions:

- forum posts
- magazine or newspaper articles
- blog entries
- user-submitted comments

Finally, just like `section` elements, `article` elements can be nested inside other `article` elements. You can also nest a `section` inside an `article`, and vice versa.

The `nav` Element

It's safe to assume that this element will appear in virtually every project. `nav` represents exactly what it implies: a group of navigation links. Although the most common use for `nav` will be for wrapping an unordered list of links, there are other

[13] http://www.whatwg.org/specs/web-apps/current-work/multipage/sections.html#the-article-element

options. You could also wrap the `nav` element around a paragraph of text that contained the major navigation links for a page or section of a page.

In either case, the `nav` element should be reserved for navigation that is of primary importance. So, it's recommended that you avoid using `nav` for a brief list of links in a footer, for example.

nav and Accessibility

A design pattern you may have seen implemented on many sites is the "skip navigation" link. The idea is to allow users of screen readers to quickly skip past your site's main navigation if they've already heard it—after all, there's no point listening to a large site's entire navigation menu every time you click through to a new page!

The `nav` element has the potential to eliminate this need; if a screen reader sees a `nav` element, it could allow its users to skip over the navigation without requiring an additional link. The specification states:

> User agents (such as screen readers) that are targeted at users who can benefit from navigation information being omitted in the initial rendering, or who can benefit from navigation information being immediately available, can use this element as a way to determine what content on the page to initially skip and/or provide on request.

Current screen readers fail to recognize `nav`, but this doesn't mean you shouldn't use it. Assistive technology will continue to evolve, and it's likely your page will be on the Web well into the future. By building to the standards now, you ensure that as screen readers improve, your page will become more accessible over time.

What's a user agent?

You'll encounter the term **user agent** a lot when browsing through specifications. Really, it's just a fancy term for a browser—a software "agent" that a user employs to access the content of a page. The reason the specs don't simply say "browser" is that user agents can include screen readers, or any other technological means to read a web page.

You can use `nav` more than once on a given page. If you have a primary navigation bar for the site, this would call for a `nav` element.

Additionally, if you had a secondary set of links pointing to different parts of the current page (using in-page anchors), this too could be wrapped in a `nav` element.

As with `section`, there's been some debate over what constitutes acceptable use of `nav` and why it isn't recommended in some circumstances (such as inside a `footer`). Some developers believe this element is appropriate for pagination or breadcrumb links, or for a search form that constitutes a primary means of navigating a site (as is the case on Google).

This decision will ultimately be up to you, the developer. Ian Hickson, the primary editor of WHATWG's HTML5 specification, responded to the question directly: "use [it] whenever you would have used class=nav".[14]

The `aside` Element

This element represents a part of the page that's "tangentially related to the content around the `aside` element, and which could be considered separate from that content."[15]

The `aside` element could be used to wrap a portion of content that is tangential to:

- a specific standalone piece of content (such as an `article` or `section`)
- an entire page or document, as is customarily done when adding a "sidebar" to a page or website

The `aside` element should never be used to wrap sections of the page that are part of the primary content; in other words, it's not meant to be parenthetical. The `aside` content could stand on its own, but it should still be part of a larger whole.

Some possible uses for `aside` include a sidebar, a secondary lists of links, or a block of advertising. It should also be noted that the `aside` element (as in the case of `header`) is not defined by its position on the page. It could be on the "side," or it could be elsewhere. It's the content itself, and its relation to other elements, that defines it.

[14] See http://html5doctor.com/nav-element/#comment-213
[15] http://dev.w3.org/html5/spec/Overview.html#the-aside-element

The `footer` Element

The final element we'll discuss in this chapter is the `footer` element. As with `header`, you can have multiple `footer`s on a single page, and you should use `footer` to wrap the section of your page that you would normally wrap inside of `<div id="footer">`.

A `footer` element, according to the spec, represents a footer for the section of content that is its nearest ancestor. The "section" of content could be the entire document, or it could be a `section`, `article`, or `aside` element.

Often a footer will contain copyright information, lists of related links, author information, and similar content that you normally think of as coming at the end of a block of content. However, much like `aside` and `header`, a `footer` element is not defined in terms of its position on the page; hence, it does not have to appear at the end of a section, or at the bottom of a page. Most likely it will, but this is not required. For example, information about the author of a blog post might be displayed above the post instead of below it, and still be considered `footer` information.

How did HTML5's creators decide which new elements to include?

You might wonder how the creators of the language came up with new semantic elements. After all, you could feasibly have dozens more semantic elements—why not have a `comment` element for user-submitted comments, or an `ad` element specifically for advertisements?

The creators of HTML5 ran tests to search through millions of web pages to see what kinds of elements were most commonly being used. The elements were decided based on the `id` and `class` attributes of the elements being examined. The results helped guide the introduction of a number of new HTML semantic elements.

Thus, instead of introducing new techniques that might be rejected or go unused, the editors of HTML5 are endeavoring to include elements that work in harmony with what web page authors are already doing. In other words, if it's common for most web pages to include a `div` element with an `id` of `header`, it makes sense to include a new element called `header`.

Structuring *The HTML5 Herald*

Now that we've covered the basics of page structure and the elements in HTML5 that will assist in this area, it's time to start building the parts of our page that will hold the content.

Let's start from the top, with a `header` element. It makes sense to include the logo and title of the paper in here, as well as the tagline. We can also add a `nav` element for the site navigation.

After the `header`, the main content of our site is divided into three columns. While you might be tempted to use `section` elements for these, stop and think about the content. If each column contained a separate "section" of information (like a sports section and an entertainment section), that would make sense. As it is, though, the separation into columns is really only a visual arrangement—so we'll use a plain old `div` for each column.

Inside those `div`s, we have newspaper articles; these, of course, are perfect candidates for the `article` element.

The column on the far right, though, contains three ads in addition to an article. We'll use an `aside` element to wrap the ads, with each ad placed inside an `article` element. This may seem odd, but look back at the description of `article`: "a self-contained composition [...] that is, in principle, independently distributable or re-usable." An ad fits the bill almost perfectly, as it's usually intended to be reproduced across a number of websites without modification.

Next up, we'll add another `article` element for the final article that appears below the ads. That final article will *not* be included in the `aside` element that holds the three ads. To belong in the `aside`, the `article` would need to be tangentially related to the page's content. This isn't the case: the `article` is part of the page's main content, so it would be wrong to include it in the `aside`.

Now the third column consists of two elements: an `aside` and an `article`, stacked one on top of the other. To help hold them together and make them easier to style, we'll wrap them in a `div`. We're not using a `section`, or any other semantic markup, because that would imply that the `article` and the `aside` were somehow topically related. They're not—it's just a feature of our design that they happen to be in the same column together.

We'll also have the entire upper section below the header wrapped in a generic `div` for styling purposes.

Finally, we have a `footer` in its traditional location, at the bottom of the page. Because it contains a few different chunks of content, each of which forms a self-contained and topically related unit, we've split them out into `section` elements. The author information will form one `section`, with each author sitting in their own nested `section`. Then there's another `section` for the copyright and additional information.

Let's add the new elements to our page, so that we can see where our document stands:

index.html ***(excerpt)***

```
<body>

<header>
  <nav></nav>
</header>

<div id="main">
  <div id="primary">
    <article></article>
    ⋮
  </div>
  <div id="secondary">
    <article></article>
    ⋮
  </div>
  <div id="tertiary">
    <aside>
      <article></article>
      ⋮
    </aside>
    <article></article>
  </div>
</div><!-- #main -->

<footer>
  <section id="authors">
    <section></section>
  </section>
  <section id="copyright"></section>
</footer>

<script src="js/scripts.js"></script>
</body>
```

Figure 2.2 shows a screenshot that displays our page with some labels indicating the major structural elements we've used.

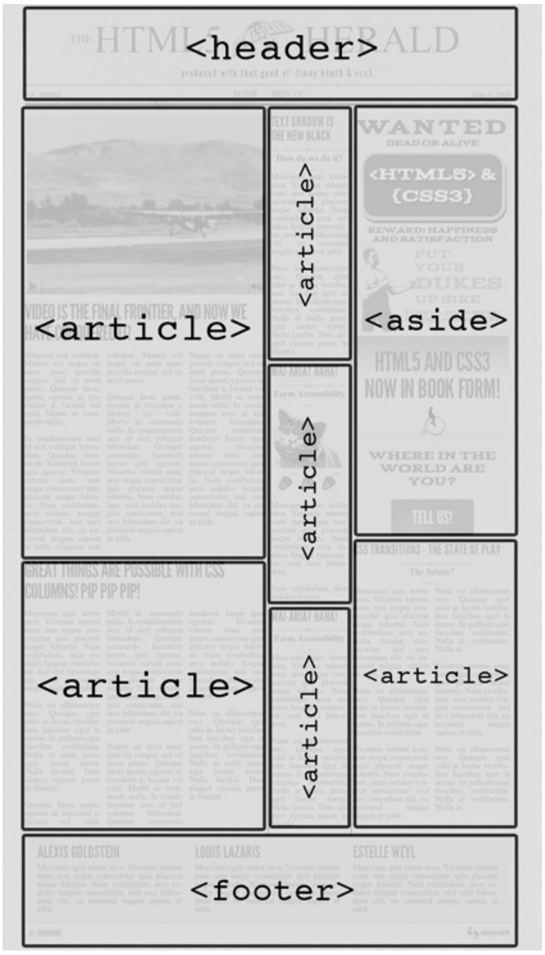


Figure 2.2. *The HTML5 Herald*, broken into structural HTML5 elements

We now have a structure that can serve as a solid basis for the content.

Something seriously cool has happened in web development!

Wrapping Things Up

That's it for this chapter. We've learned some of the basics of content structure in HTML5, and we've started to build our sample project using the knowledge we've gained.

In the next chapter, we'll have a more in-depth look at how HTML5 deals with different types of content. Then, we'll continue to add semantics to our page when we deal with more new HTML elements.

Chapter 7

CSS3 Gradients and Multiple Backgrounds

In Chapter 6, we learned a few ways to add decorative styling features—like shadows and rounded corners—to our pages without the use of additional markup or images. The next most common feature frequently added to websites that used to require images is gradients. CSS3 provides us with the ability to create native radial and linear gradients, as well as include multiple background images on any element. With CSS3, there's no need to create the multitudes of JPEGs of years past, or add nonsemantic hooks to our markup.

Browser support for gradients and multiple backgrounds is still evolving, but as you'll see in this chapter, it's possible to develop in a way that supports the latest versions of all major browsers—including IE9.

We'll start by looking at CSS3 gradients—but first, what *are* gradients? **Gradients** are smooth transitions between two or more specified colors. In creating gradients, you can specify multiple in-between color values, called **color stops**. Each color stop is made up of a color and a position; the browser fades the color from each stop to the next to create a smooth gradient. Gradients can be utilized anywhere a background image can be used. This means that in your CSS, a gradient can be theoretically employed anywhere a `url()` value can be used, such as `background-image`, `border-image`, and even `list-style-type`, though for now the most consistent support is for background images.

By using CSS gradients to replace images, you avoid forcing your users to download extra images, support for flexible layouts is improved, and zooming is no longer pixelated the way it can be with images.

There are two types of gradients currently available in CSS3: linear and radial. Let's go over them in turn.

Linear Gradients

Linear gradients are those where colors transition across a straight line: from top to bottom, left to right, or along any arbitrary axis. If you've spent any time with image-editing tools like Photoshop and Fireworks, you should be familiar with linear gradients—but as a refresher, Figure 7.1 shows some examples.

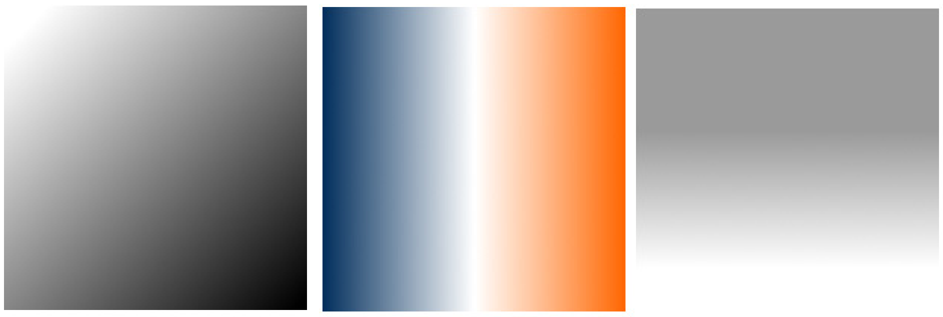


Figure 7.1. Linear gradient examples

Similar to image-editing programs, to create a linear gradient you specify a direction, the starting color, the end color, and any color stops you want to add along that line. The browser takes care of the rest, filling the entire element by painting lines of color perpendicular to the line of the gradient. It produces a smooth fade from one color to the next, progressing in the direction you specify.

When it comes to browsers and linear gradients, things get a little messy. WebKit first introduced gradients several years ago using a particular and, many argued, convoluted syntax. After that, Mozilla implemented gradients using a simpler and more straightforward syntax. Then, in January of 2011, the W3C included a proposed syntax in CSS3. The new syntax is very similar to Firefox's existing implementation—in fact, it's close enough that any gradient written with the new syntax will work just fine in Firefox. The W3C syntax has also been adopted by WebKit, though, at the time of writing it's only in the nightly builds, and yet to make its way into Chrome and Safari; they are still using the old-style syntax. For backwards compatibility reasons, those browsers will continue to support the old syntax even once the standard form is implemented. And Opera, with the release of version 11.10, supports the new W3C standard for linear gradients. All the current implementations use vendor prefixes (`-webkit-`, `-moz-`, and `-o-`).

WebKit Nightly Builds

The WebKit engine that sits at the heart of Chrome and Safari exists independently as an open source project at http://www.webkit.org/. However, new features implemented in WebKit take some time to be released in Chrome or Safari. In the meantime, it's still possible to test these features by installing one of the **nightly builds**, so-called because they're built and released on a daily basis, incorporating new features or code changes from a day's work by the community. Because they're frequently released while in development, they can contain incomplete features or bugs, and will often be unstable. Still, they're great if you want to test new features (like the W3C gradient syntax) that are yet to make it into Chrome or Safari. Visit http://nightly.webkit.org/ to obtain nightly builds of WebKit for Mac or Windows.

That still leaves us with the question of how to handle gradients in IE and older versions of Opera. Fortunately, IE9 and Opera 11.01 and earlier support SVG backgrounds—and it's fairly simple to create gradients in SVG. (We'll be covering SVG in more detail in Chapter 11.) And finally, all versions of IE support a proprietary filter that enables the creation of basic linear gradients.

Confused? Don't be. While gradients are important to understand, it's unnecessary to memorize all the browser syntaxes. We'll cover the new syntax, as well as the soon-to-be-forgotten old-style WebKit syntax, and then we'll let you in on a little secret: there are tools that will create all the required styles for you, so there's no need to remember all the specifics of each syntax. Let's get started.

There's one linear gradient in *The HTML5 Herald*, in the second advertisement block shown in Figure 7.2 (which happens to be advertising this very book!). You'll note that the gradient starts off dark at the top, lightens, then darkens again as if to create a road under the cyclist, before lightening again.

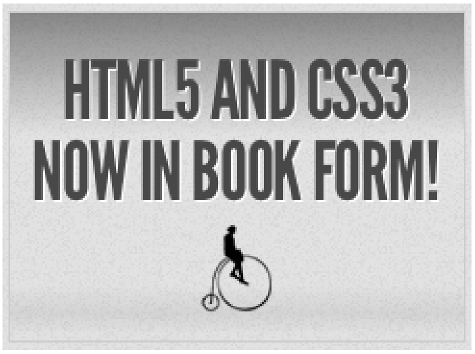


Figure 7.2. A linear gradient in *The HTML5 Herald*

To create a cross-browser gradient for our ad, we'll start with the new standard syntax. It's the simplest and easiest to understand, and likely to be the only one you'll need to use in a few years' time. After that, we'll look at how the older WebKit and Firefox syntaxes differ from it.

The W3C Syntax

Here's the basic syntax for linear gradients:

```
background-image: linear-gradient( … );
```

Inside those parentheses, you specify the direction of the gradient, and then provide some color stops. For the direction, you can provide either the angle along which the gradient should proceed, or the side or corner from which it should start—in which case it will proceed towards the opposite side or corner. For angles, you use values in degrees (deg). `0deg` points to the right, `90deg` is up, and so on counter-clockwise. For a side or corner, use the `top`, `bottom`, `left`, and `right` keywords. After specifying the direction, provide your color stops; these are made up of a color and a percentage or length specifying how far along the gradient that stop is located.

That's a lot to take in, so let's look at some gradient examples. For the sake of illustration we'll use a gradient with just two color stops: `#FFF` (white) to `#000` (black).

To have the gradient go from top to bottom of an element, as shown in Figure 7.3, you could specify any of the following (our examples are using the `-moz-` prefix, but remember that `-webkit-` and `-o-` support the same syntax):

```
background-image: -moz-linear-gradient(270deg, #FFF 0%, #000 100%);
background-image: -moz-linear-gradient(top, #FFF 0%, #000 100%);
background-image: -moz-linear-gradient(#FFF 0%, #000 100%);
```

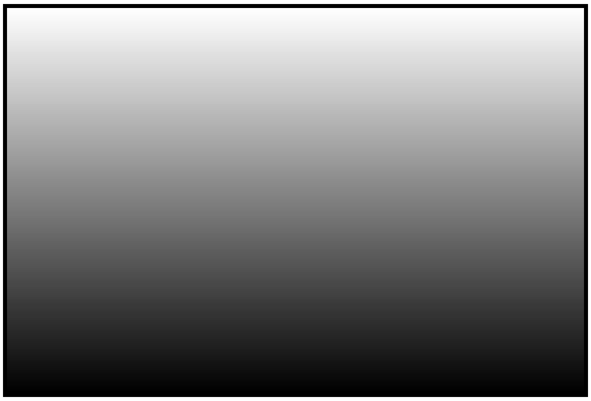


Figure 7.3. A white-to-black gradient from the top center to the bottom center of an element

The last declaration works because `top` is the default in the absence of a specified direction.

Because the first color stop is assumed to be at 0%, and the last color stop is assumed to be at 100%, you could also omit the percentages from that example and achieve the same result:

```
background-image: -moz-linear-gradient(#FFF, #000);
```

Now, let's put our gradient on an angle, and place an additional color stop. Let's say we want to go from black to white, and then back to black again:

```
background-image: -moz-linear-gradient(30deg, #000, #FFF 75%, #000);
```

We've placed the color stop 75% along the way, so the white band is closer to the gradient's end point than its starting point, as shown in Figure 7.4.

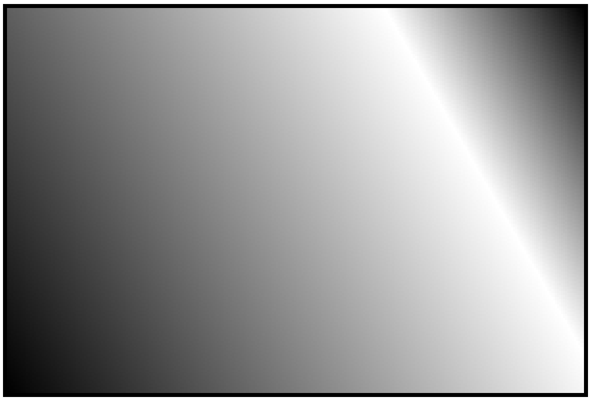


Figure 7.4. A gradient with three color stops

You can place your first color stop somewhere other than 0%, and your last color stop at a place other than 100%. All the space between 0% and the first stop will be the same color as the first stop, and all the space between the last stop and 100% will be the color of the last stop. Here's an example:

```
background-image: -moz-linear-gradient(30deg, #000 50%, #FFF 75%,
➥#000 90%);
```

The resulting gradient is shown in Figure 7.5.

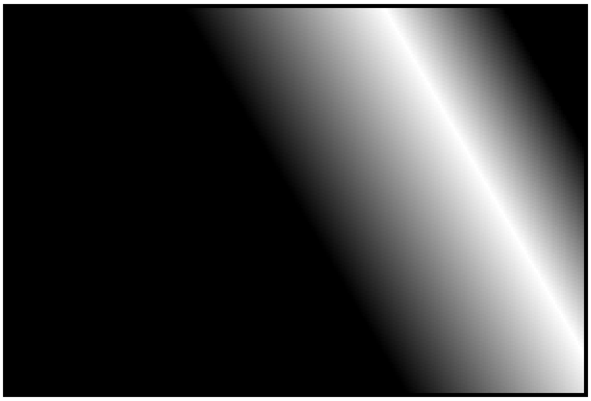


Figure 7.5. A gradient confined to a narrow band by offsetting the start and end color stops

You don't actually need to specify positions for any of the color stops. If you omit them, the stops will be evenly distributed. Here's an example:

```
background-image:
  -moz-linear-gradient(45deg,
    #FF0000 0%,
    #FF6633 20%,
    #FFFF00 40%,
    #00FF00 60%,
    #0000FF 80%,
    #AA00AA 100%);

background-image:
  -moz-linear-gradient(45deg,
    #FF0000,
    #FF6633,
    #FFFF00,
    #00FF00,
    #0000FF,
    #AA00AA);
```

Either of the previous declarations makes for a fairly unattractive angled rainbow. Note that we've added line breaks and indenting for ease of legibility—they are not essential.

Colors transition smoothly from one color stop to the next. However, if two color stops are placed at the same position along the gradient, the colors won't fade, but will stop and start on a hard line. This is a way to create a striped background effect, like the one shown in Figure 7.6.

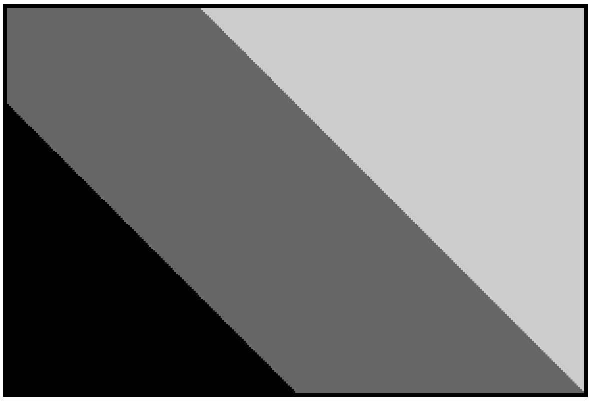


Figure 7.6. Careful placement of color stops can create striped backgrounds

Here are the styles used to construct that example:

```
background-image:
  -moz-linear-gradient(45deg,
    #000000 30%,
    #666666 30%,
    #666666 60%,
    #CCCCCC 60%,
    #CCCCCC 90%);
```

At some point in the reasonably near future, you can expect this updated version of the syntax to be the only one you'll need to write—but we're not quite there yet.

The Old WebKit Syntax

As we've mentioned, the latest development version of WebKit has adopted the W3C syntax; however, most current releases of WebKit-based browsers still use an older syntax, which is a little more complicated. Because those browsers still need to be supported, let's quickly take a look at the old syntax, using our first white-to-black gradient example again:

```
background-image:
  -webkit-gradient(linear, 0% 0%, 0% 100%, from(#FFFFFF),
➥to(#000000));
```

Rather than use a specific `linear-gradient` property, there's a general-purpose `-webkit-gradient` property, where you specify the type of gradient (linear in this case) as the first parameter. The linear gradient then needs both a start and end point to determine the direction of the gradient. The start and end points can be specified using percentages, numeric values, or the keywords top, bottom, left, right, or center.

The next step is to declare color stops of the gradients. You can include the originating color with the `from` keyword, and the end color with the `to` keyword. Then, you can include any number of intermediate colors using the `color-stop()` function to create a color stop. The first parameter of the `color-stop()` function is the position of the stop, expressed as a percentage, and the second parameter is the color at that location.

Here's an example:

```
background-image:
  -webkit-gradient(linear, left top, right bottom,
    from(red),
    to(purple),
    color-stop(20%, orange),
    color-stop(40%, yellow),
    color-stop(60%, green),
    color-stop(80%, blue));
```

With that, we've recreated our angled rainbow, reminiscent of GeoCities circa 1996.

It's actually unnecessary to specify the start and end colors using `from` and `to`. As `from(red)` is the equivalent to `color-stop(0, red)`, we could also have written:

```
background-image:
  -webkit-gradient(linear, left top, right bottom,
    color-stop(0, red),
    color-stop(20%, orange),
    color-stop(40%, yellow),
    color-stop(60%, green),
    color-stop(80%, blue),
    color-stop(100%, purple));
```

If you don't declare a `from` or a 0% color stop, the color of the first color stop is used for all the area up to that first stop. The element will have a solid area of the color declared from the edge of the container to the first specified color stop, at which point it becomes a gradient morphing into the color of the next color stop. At and after the last stop, the color of the last color stop is used. In other words, if the first color stop is at 40% and the last color stop is at 60%, the first color will be used from 0% to 40%, and the last color will be displayed from 60% to 100%, with the area from 40% to 60% a gradient morphing between the two colors.

As you can see, this is more complicated than the Mozilla syntax. Fortunately, tools exist to generate all the required code for a given gradient automatically. We'll be looking at some of them at the end of this section, but first, we'll see how to use both syntaxes to create a cross-browser gradient for *The HTML5 Herald*. The good news is that since the old WebKit syntax uses a different property name (`-webkit-gradient` instead of `-webkit-linear-gradient`), you can use both syntaxes side-by-side without conflict. In fact, the old syntax is still supported in the newer WebKit, so the browser will just use whichever one was declared last.

Putting It All Together

Now that we have a fairly good understanding of how to declare linear gradients, let's declare ours.

If your designer included a gradient in the design, it's likely to have been created in Photoshop or another image-editing program. You can use this to your advantage; if you have the original files, it's fairly easy to replicate exactly what your designer intended.

If we pop open Photoshop and inspect the gradient we want to use for the ad (shown in Figure 7.7), our gradient is linear, with five color stops that simply change the opacity of a single color (black).

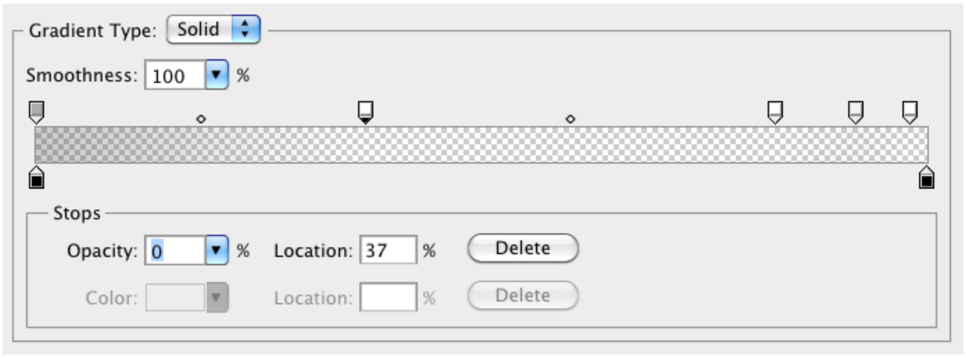

Gradient Type: Solid
Smoothness: 100 %
Stops
Opacity: 0 % Location: 37 % Delete
Color: Location: % Delete

Figure 7.7. An example linear gradient in Photoshop

You'll note via the Photoshop screengrab that the color starts from 40% opacity, and that the first color stop's location is at 37%, with an opacity of 0%. We can use this tool to grab the data for our CSS declaration, beginning with the W3C syntax declaration for Firefox, Opera 11.10, and newer WebKit browsers:

css/styles.css *(excerpt)*

```
#ad2 {
  ⋮
  background-image:
    -moz-linear-gradient(
      270deg,
      rgba(0,0,0,0.4) 0,
      rgba(0,0,0,0) 37%,
      rgba(0,0,0,0) 83%,
      rgba(0,0,0,0.06) 92%,
      rgba(0,0,0,0) 98%
    );
  background-image:
    -webkit-linear-gradient(
      270deg,
      rgba(0,0,0,0.4) 0,
      rgba(0,0,0,0) 37%,
      rgba(0,0,0,0) 83%,
      rgba(0,0,0,0.06) 92%,
      rgba(0,0,0,0) 98%
    );
  background-image:
    -o-linear-gradient(
      270deg,
      rgba(0,0,0,0.4) 0,
      rgba(0,0,0,0) 37%,
      rgba(0,0,0,0) 83%,
      rgba(0,0,0,0.06) 92%,
      rgba(0,0,0,0) 98%
    );
}
```

We want the gradient to run from the very top of the ad to the bottom, so we set the angle to `270deg` (towards the bottom). We've then added all the color stops from the Photoshop gradient. Note that we've omitted the end point of the gradient, because the last color stop is at 98%—everything after that stop will be the same color as the stop in question (in this case, black at 0% opacity, or full transparency).

Now let's add in the old WebKit syntax, with the unprefixed version last to future-proof the declaration:

css/styles.css *(excerpt)*

```
#ad2 {
  ⋮
  background-image:
    -webkit-gradient(linear,
      from(rgba(0,0,0,0.4)),
      color-stop(37%, rgba(0,0,0,0)),
      color-stop(83%, rgba(0,0,0,0)),
      color-stop(92%, rgba(0,0,0,0.16)),
      color-stop(98%, rgba(0,0,0,0)));
  background-image:
    -webkit-linear-gradient(
      270deg,
      rgba(0,0,0,0.4) 0,
      rgba(0,0,0,0) 37%,
      rgba(0,0,0,0) 83%,
      rgba(0,0,0,0.06) 92%,
      rgba(0,0,0,0) 98%
    );
  background-image:
    linear-gradient(
      270deg,
      rgba(0,0,0,0.4) 0,
      rgba(0,0,0,0) 37%,
      rgba(0,0,0,0) 83%,
      rgba(0,0,0,0.06) 92%,
      rgba(0,0,0,0) 98%
    );
}
```

We now have our gradient looking just right in Mozilla, Opera, and WebKit-based browsers.

Linear Gradients with SVG

We still have a few more browsers to add our linear gradient to. In Opera 11.01 and earlier—and more importantly, IE9—we can declare SVG files as background images. By creating a gradient in an SVG file, and declaring that SVG as the background image of an element, we can recreate the same effect we achieved with CSS3 gradients.

SV what?

SVG stands for Scalable Vector Graphics. It's an XML-based language for defining vector graphics using a set of elements—like what you use in HTML to define the structure of a document. We'll be covering SVG in much more depth in Chapter 11, but for now we'll just skim over the basics, since all we're creating is a simple gradient.

An SVG file sounds scary, but for creating gradients it's quite straightforward. Here's our gradient again, in SVG form:

images/gradient.svg *(excerpt)*

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
  "http://www.w3.org/TR/2001/REC-SVG-20050904/DTD/svg10.dtd">
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:xlink="http://www.w3.org/1999/xlink" version="1.1">
<title>Module Gradient</title>
 <defs>
  <linearGradient id="grad"  x1="0" y1="0" x2="0" y2="100%">
    <stop offset="0" stop-opacity="0.3" color-stop="#000000" />
    <stop offset="0.37" stop-opacity="0" stop-color="#000000" />
    <stop offset="0.83" stop-opacity="0" stop-color="#000000" />
    <stop offset="0.92" stop-opacity="0.06" stop-color="#000000" />
    <stop offset="0.98" stop-opacity="0" stop-color="#000000" />
  </linearGradient>
</defs>
<rect x="0" y="0" width="100%" height="100%"
➥style="fill:url(#grad)" />
</svg>
```

Looking at the SVG file, you should notice that it's quite similar to the syntax for linear gradients in CSS3. We declare the gradient type and the orientation in the `linearGradient` element; then add color stops. The orientation is set with start and end coordinates, from `x1, y1` to `x2, y2`. The color stops are fairly self-explanatory, having an offset between 0 and 1 determining their position and a stop-color for their color. After declaring the gradient, we then have to create a rectangle (the `rect` element) and fill it with our gradient using the `style` attribute.

So, we've created a nifty little gradient, but how do we use it on our site? Save the SVG file with the **.svg** extension. Then, in your CSS, simply declare the SVG as your background image with the same syntax, as if it were a JPEG, GIF, or PNG:

css/styles.css *(excerpt)*

```
#ad2 {
  ⋮
  background-image: url(../images/gradient.svg);
  ⋮
}
```

The SVG background should be declared before the CSS3 gradients, so browsers that understand both will use the latter. Many browsers are even smart enough not to download the SVG if it's overwritten by another `background-image` property later on in your CSS.

The major difference between our CSS linear gradients and the SVG version is that the SVG background image won't default to 100% of the height and width of the container the way that CSS gradients do. To make the SVG fill the container, declare the `height` and `width` of your SVG rectangle as `100%`.

Linear Gradients with IE Filters

For Internet Explorer prior to version 9, we can use the proprietary IE filter syntax to create simple gradients. The IE gradient filter doesn't support color stops, gradient angle, or, as we'll see later, radial gradients. All you have is the ability to specify whether the gradient is horizontal or vertical, as well as the "to" and "from" colors. It's fairly basic, but if you need a gradient on these older browsers, it can provide the solution.

The filter syntax for IE is:

```
filter:progid:DXImageTransform.Microsoft.gradient(GradientType=0,
➥startColorstr='#COLOR', endColorstr='#COLOR); /* IE6 & IE7 */
-ms-filter:"progid:DXImageTransform.Microsoft.gradient(GradientType=
➥0,startColorstr='#COLOR', endColorstr='#COLOR')"; /* IE8 */
```

The *`GradientType`* parameter should be set to 1 for a horizontal gradient, or 0 for a vertical gradient.

Since the gradient we're using for our ad block requires color stops, we'll skip using the IE filters. The ad still looks fine without the gradient, so it's all good.

Filters Kinda Suck

As we've mentioned before, IE's filters can have a significant impact on performance, so use them sparingly, if at all. Calculating the display of filter effects takes processing time, with some effects being slower than others. SVGs can have a similar—albeit lesser—effect, so be sure to test your site in a number of browsers if you're using these fallbacks.

Tools of the Trade

Now that you understand how to create linear gradients and have mastered the intricacies of their convoluted syntax, you can forget almost everything you've learned. There are some very cool tools to help you create linear gradients without having to recreate your code four times for each different browser syntax.

John Allsop's http://www.westciv.com/tools/gradients/ is a tool that enables you to create gradients with color stops for both Firefox and WebKit. Note that there are separate tabs for Firefox and WebKit, and for radial and linear gradients. The tool only creates gradients with hexadecimal color notation, but it does provide you with copy-and-paste code, so you can copy it, and then switch the hexadecimal color values to RGBA or HSLA if you prefer.

Damian Galarza's http://gradients.glrzad.com/ provides for both color stops and RGB. It even lets you set colors with an HSL color picker, but converts it to RGB in the code. It does not provide for alpha transparency, but since the code generated is in RGB, it's easy to update. This gradient generator is more powerful than the Westciv one, but may be a bit overwhelming for a newbie.

Finally, Paul Irish's http://css3please.com/ allows you to create linear gradients, though it has no support for color stops. You may wonder why it's even worth mentioning—but it is, because it's the only one of the tools mentioned that provides the filter syntax for IE alongside the other gradient syntaxes. Plus, as well as gradients, it can give you cross-browser syntax for lots of other features as well, like shadows and rounded corners.

Radial Gradients

Radial gradients are circular or elliptical gradients. Rather than proceeding along a straight axis, colors blend out from a starting point in all directions. Radial gradients are supported in WebKit and Mozilla (beginning with Firefox 3.6). While Opera 11.10 has begun supporting linear gradients, it does not provide support for radial gradients; however, as with linear gradients, radial gradients can be created in SVG, so support can be provided to Opera and IE9. Radial gradients are entirely unsupported in IE8 and earlier—not even with filters.

The W3C Syntax

Let's start with a simple circular gradient to illustrate the standard syntax:

```
background-image: -moz-radial-gradient(#FFF, #000);
background-image: -moz-radial-gradient(center, #FFF, #000);
background-image: -moz-radial-gradient(center, ellipse cover,
➥#FFF, #000);
```

The above three declarations are functionally identical, and will all result in the gradient shown in Figure 7.8. At the minimum, you need to provide a start color and an end color. Alternatively, you can also provide a position for the center of the gradient as the first parameter, and a shape and size as the second parameter.

Figure 7.8. A simple, centered radial gradient

Let's start by playing with the position:

```
background-image: -moz-radial-gradient(30px 30px, #FFF, #000);
```

This will place the center of the gradient 30 pixels from the top and 30 pixels from the left of the element, as you can see in Figure 7.9. As with `background-position`, you can use values, percentages, or keywords to set the gradient's position.

Figure 7.9. A gradient positioned off center

Now let's look at the shape and size parameter. The shape can take one of two values, `circle` or `ellipse`, with the latter being the default.

For the size, you can use one of the following values:

`closest-side`
: The gradient's shape meets the side of the box closest to its center (for circles), or meets both the vertical and horizontal sides closest to the center (for ellipses).

`closest-corner`
: The gradient's shape is sized so it meets exactly the closest corner of the box from its center.

`farthest-side`
: Similar to `closest-side`, except that the shape is sized to meet the side of the box farthest from its center (or the farthest vertical and horizontal sides in the case of ellipses).

`farthest-corner`
: The gradient's shape is sized so that it meets exactly the farthest corner of the box from its center.

`contain`
: A synonym for `closest-side`.

`cover`

A synonym for `farthest-corner`.

According to the spec, you can also provide a second set of values to explicitly define the horizontal and vertical size of the radial gradient. This is currently only supported in WebKit, though support should be added to Firefox in the near future. For now, though, you should probably stick to the above constraints if you want to create the same gradient in all supporting browsers.

The color stop syntax is the same as for linear gradients: a color value followed by an optional stop position. Let's look at one last example:

```
background-image: -moz-radial-gradient(30px 30px, circle
➥farthest-side, #FFF, #000 30%, #FFF);
```

This will create a gradient like the one in Figure 7.10.

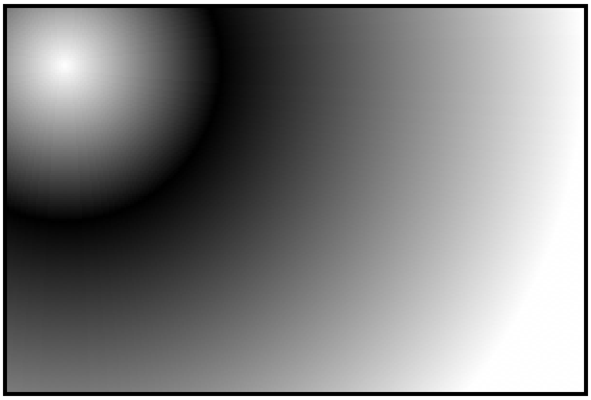


Figure 7.10. A radial gradient with a modified size and shape, and an extra color stop

The Old WebKit Syntax

To create the example in Figure 7.10 using the old-style WebKit syntax currently supported in Safari and Chrome, you'd need to write it as follows:

```
background-image: -webkit-gradient(radial, 30 30, 0, 30 30, 100%,
➥from(#FFFFFF), to(#FFFFFF), color-stop(.3,#000000))
```

The same `-webkit-gradient` property used earlier for linear gradients is used for radial gradients; the difference is that we pass `radial` as the first parameter. The next four parameters are the respective position and radius of two circles, with the gradient proceeding from the inner circle to the outer circle. Just to make it more confusing, these values are defined in pixels, but without the `px` unit. You can also specify the values as percentages, in which case you *do* need to include the % symbol. You should be starting to see why the W3C opted for a version of the Mozilla syntax rather than this one.

Furthermore, your inner circle doesn't need to be centered in your outer circle. If the first point is equal to the second point, the gradient will be symmetrical like the one in Figure 7.10. If they differ, however, your inner circle will be off-center, so the gradient will be asymmetrical. If your inner circle's center is outside the boundary of your outer circle, instead of having a circle off-center inside another circle, you'll have a very odd triangle gradient effect, as Figure 7.11 illustrates.

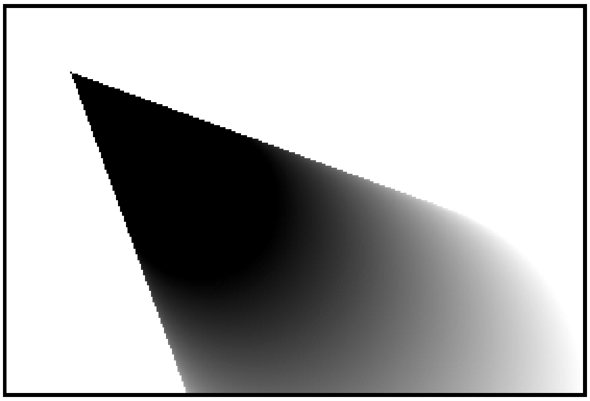


Figure 7.11. The older WebKit radial gradient syntax allowed for some interesting effects

Here's the code used to create that gradient:

```
background-image:-webkit-gradient(radial, 200 200, 100, 100 100, 40,
from(#FFFFFF), to(#000000));
```

As with the linear gradients, you can insert more colors with the `color-stop` function. The syntax for color stops is the same for both linear and radial gradients.

You'll generally want to create gradients that look the same in older versions of Chrome and Safari as they do in newer versions of those browsers and Firefox, so you should limit yourself to the kinds of gradients that can be replicated using the W3C syntax. However, if you do find yourself building specifically for WebKit browsers (for mobile platforms, for example), it can be useful to know that these additional options exist. As mentioned earlier, the old syntax will continue to be supported in WebKit browsers for the foreseeable future.

Putting it All Together

Let's take all we've learned and implement a radial gradient for *The HTML5 Herald*. You may yet to have noticed, but the form submit button has a radial gradient in the background. The center of the radial gradient is outside the button area, towards the left and a little below the bottom, as Figure 7.12 shows.

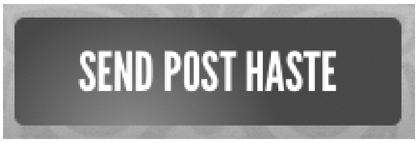


Figure 7.12. A radial gradient on a button in *The HTML5 Herald*'s sign-up form

We'll want to declare at least three background images: an SVG file for Opera and IE9, the older WebKit syntax for Chrome and Safari, and the `-moz-` vendor prefixed version for Firefox. You can also declare the newer WebKit vendor prefixed version (currently only in the WebKit nightly builds), as well as the non-prefixed version:

css/styles.css *(excerpt)*

```
input[type=submit] {
  ⋮
  background-color: #333;
  /* SVG for IE9 and Opera */
  background-image: url(../images/button-gradient.svg);
  /* Old WebKit */
  background-image: -webkit-gradient(radial,
    30% 120%, 0, 30% 120%, 100,
    color-stop(0,rgba(144,144,144,1)),
    color-stop(1,rgba(72,72,72,1)));
  /* W3C for Mozilla */
  background-image: -moz-radial-gradient(30% 120%, circle,
    rgba(144,144,144,1) 0%,
    rgba(72,72,72,1) 50%);
  /* W3C for new WebKit */
  background-image: -webkit-radial-gradient(30% 120%, circle,
    rgba(144,144,144,1) 0%,
    rgba(72,72,72,1) 50%);
  /* W3C unprefixed */
  background-image: radial-gradient(30% 120%, circle,
    rgba(144,144,144,1) 0%,
    rgba(72,72,72,1) 50%);
}
```

The center of the circle is 30% from the left, and 120% from the top, so it's actually *below* the bottom edge of the container. We've included two color stops for the color `#484848` (or `rgb(72,72,72)`) and `#909090` (or `rbg(144,144,144)`).

And here's the SVG file used as a fallback, though we'll stop short of explaining it here, as the syntax is fairly explanatory and we'll be covering SVG in Chapter 11:

button-gradient.svg *(excerpt)*

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
➥"http://www.w3.org/TR/2001/REC-SVG-20050904/DTD/svg10.dtd">
<svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="
➥http://www.w3.org/1999/xlink" version="1.1">
<title>Button Gradient</title>
 <defs>
  <radialGradient id="grad"  cx="30%" cy="120%" fx="30%" fy="120%"
➥r="50%" gradientUnits="userSpaceOnUse">
    <stop offset="0" stop-color="#909090" />
    <stop offset="1" stop-color="#484848" />
  </radialGradient>
</defs>
<rect x="0" y="0" width="100%" height="100%"
➥style="fill:url(#grad)" />
</svg>
```

Repeating Gradients

Sometimes you'll find yourself wanting to create a gradient "pattern" that repeats over the background of an element. While linear-repeating gradients can be created by repeating the background image (with `background-repeat`), there's no equivalent way to easily create repeating radial gradients. Fortunately, CSS3 comes to the rescue with both a `repeating-linear-gradient` and a `repeating-radial-gradient` syntax. The vendor-prefixed `repeating-linear-gradient` syntax is supported in Firefox 3.6+, Safari 5.0.3+, Chrome 10+, and Opera 11.10+.

Gradients with `repeating-linear-gradient` and `repeating-radial-gradient` have the same syntax as the nonrepeating versions.

Supported in Firefox 3.6, Chrome 10, and the WebKit nightly build (hence, Safari 6), here are examples of what can be created with just a few lines of CSS (again using only the `-webkit-` prefixed syntax for brevity):

```
.repeat_linear_1 {
  background-image:
    -webkit-repeating-linear-gradient(left,
      rgba(0,0,0,0.5) 10%,
      rgba(0,0,0,0.1) 30%);
}
.repeat_radial_2 {
  background-image:
    -webkit-repeating-radial-gradient(top left, circle,
      rgba(0,0,0,0.9),
      rgba(0,0,0,0.1) 10%,
      rgba(0,0,0,0.5) 20%);
}
.multiple_gradients_3 {
  background-image:
    -webkit-repeating-linear-gradient(left,
      rgba(0,0,0,0.5) 10%,
      rgba(0,0,0,0.1) 30%),
    -webkit-repeating-radial-gradient(top left, circle,
      rgba(0,0,0,0.9),
      rgba(0,0,0,0.1) 10%,
      rgba(0,0,0,0.5) 20%);
}
```

The resulting gradients are shown in Figure 7.13.

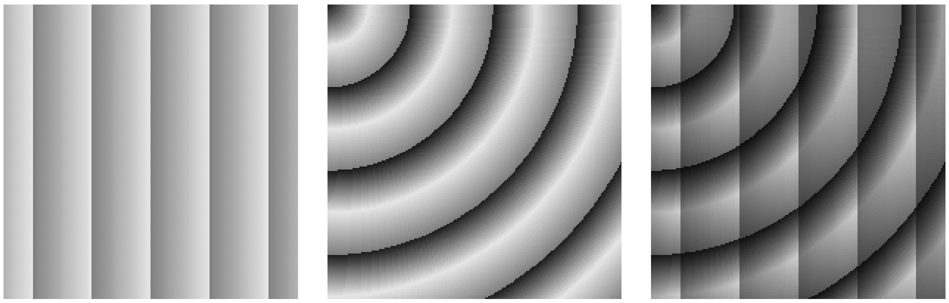


Figure 7.13. A few examples of repeating gradients

Multiple Background Images

You probably noticed that our advertisement with the linear gradient is incomplete: we're missing the bicycle. Prior to CSS3, adding the bicycle would have required placing an additional element in the markup to contain the new background image. In CSS3, there's no need to include an element for every background image; it provides us with the ability to add more than one background image to any element, even to pseudo-elements.

To understand multiple background images, you need to understand the syntax and values of the various background properties. The syntax for the values of all the background properties, including `background-image` and the shorthand `background` property, are the same whether you have one background image or many. To make a declaration for multiple background images, simply separate the values for each individual image with a comma. For example:

```
background-image:
  url(firstImage.jpg),
  url(secondImage.gif),
  url(thirdImage.png);
```

This works just as well if you're using the shorthand `background` property:

```
background:
  url(firstImage.jpg) no-repeat 0 0,
  url(secondImage.gif) no-repeat 100% 0,
  url(thirdImage.png) no-repeat 50% 0;
```

The background images are layered one on top of the other with the first declaration on top, as if it had a high `z-index`. The final image is drawn under all the images preceding it in the declaration, as if it had a low `z-index`. Basically, think of the images as being stacked in reverse order: first on top, last on the bottom.

If you want to declare a background color—which you should, especially if it's light-colored text on a dark-colored background image—declare it last. It's often simpler and more readable to declare it separately using the `background-color` property.

As a reminder, the shorthand `background` property is short for eight longhand background properties. If you use the shorthand, any longhand background property value that's omitted from the declaration will default to the longhand property's default (or initial) value. The default values for the various background properties are listed below:

- `background-color: transparent;`
- `background-image: none;`
- `background-position: 0 0;`
- `background-size: auto;`
- `background-repeat: repeat;`
- `background-clip: border-box;`
- `background-origin: padding-box;`
- `background-attachment: scroll;`

Just like for a declaration of a single background image, you can include a gradient as one of several background images. Here's how we do it for our advertisement. For brevity, only the unprefixed version is shown. The bicycle image would be included similarly in each `background-image` declaration:

css/styles.css *(excerpt)*

```
#ad2 {
  ⋮
  background-image:
    url(../images/bg-bike.png),
    linear-gradient(top,
    rgba(0,0,0,0.4) 0,
    rgba(0,0,0,0) 37%,
    rgba(0,0,0,0) 83%,
    rgba(0,0,0,0.06) 92%,
    rgba(0,0,0,0) 98%);
  background-position: 50% 88%, 0 0;
}
```

Note that we've put the bicycle picture first in the series of background images, since we want the bicycle to sit on top of the gradient, instead of the other way around. We've also declared the background position for each image by putting them in the same order as the images were declared in the `background-image` property. If only one set of values was declared—for example, `background-position: 50% 88%;`—all images would have the same background position as if you'd declared `background-position: 50% 88%, 50% 88%;`. In this case, the `50% 80%` positions the bicycle, which was declared first, and the `0 0` (or `left top`) positions the gradient.

Because a browser will only respect one `background-image` property declaration (whether it has one or many images declared), the bicycle image must be included in each `background-image` declaration, since they're all targeting different browsers. Remember, browsers ignore CSS that they fail to understand. So if Safari doesn't understand `-moz-linear-gradient` (which it doesn't), it will ignore the entire property/value pair.

The heading on our sign-up form also has two background images. While we could attach a single extra-wide image in this case, spanning across the entire form, there's no need! With multiple background images, CSS3 allows us to attach two separate small images, or a single image sprite twice with different background positions. This saves on bandwidth, of course, but it could also be beneficial if the heading needed to stretch; a single image would be unable to accommodate differently sized elements. This time, we'll use the background shorthand:

```
background:
  url(../images/bg-formtitle-left.png) left 13px no-repeat,
  url(../images/bg-formtitle-right.png) right 13px no-repeat;
```

The background Shorthand

When all the available background properties are fully supported, the following two statements will be equivalent:

```
div {
  background: url("tile.png") no-repeat scroll center
➥bottom / cover rgba(0, 0, 0, 0.2);
}

div {
  background-color: rgba(0,0,0,0.2);
  background-position: 50% 100%;
  background-size: cover;
  background-repeat: no-repeat;
  background-clip: border-box;
  background-origin: padding-box;
  background-attachment: scroll;
  background-image: url(form.png);
}
```

Currently, though, since only some browsers support all the values available, we recommend including `color`, `position`, `repeat`, `attachment`, and `image` in your shorthand declaration, with `clip`, `origin`, and `size` following, or avoiding the shorthand altogether. You must declare the shorthand *before* the longhand properties, as any value not explicitly declared in the shorthand will be treated as though you'd declared the default value.

Background Size

The `background-size` property allows you to specify the size you want your background images to have. In theory, you can include `background-size` within the shorthand `background` declaration by adding it after the background's position, separated with a slash (/). As it stands, no browser understands this shorthand; in fact, it will cause them to ignore the entire `background` declaration, since they see it as incorrectly formatted. As a result, you'll want to use the `background-size` property as a separate declaration instead.

Support for `background-size` is as follows:

- Opera 11.01+: `background-size` (unprefixed)

- Safari and Chrome: current versions support unprefixed, but older versions require `-webkit-background-size`
- Firefox: `-moz-background-size` for 3.6, `background-size` for 4+
- IE9: `background-size`

As you can see, adoption of the unprefixed version of this syntax was very quick; it's a simple property with a straightforward implementation that was unlikely to change. This is a great example of why you should always include the unprefixed version in your CSS.

If declaring the background image size in pixels, be careful to avoid the image distorting; define either the width or the height, not both, and set the other value to `auto`. This will preserve the aspect ratio of your image. If you only include one value, the second value is assumed to be `auto`. In other words, both these lines have the same meaning:

```
background-size: 100px auto, auto auto;
background-size: 100px, auto auto;
```

As with all background properties, use commas to separate values for each image declared. If we wanted our bicycle to be really big, we could declare:

```
-webkit-background-size: 100px, cover;
-moz-background-size: 100px, cover;
-o-background-size: 100px, cover;
background-size: 100px auto, cover;
```

By declaring just the width of the image, the second value will default to `auto`, and the browser will determine the correct height of the image based on the aspect ratio.

The default size of a background image is the actual size of the image. Sometimes the image is just a bit smaller or larger than its container. You can define the size of your background image in pixels (as shown above) or percentages, or you can use the `contain` or `cover` key terms.

The `contain` value scales the image while preserving its aspect ratio; this may leave uncovered space. The `cover` value scales the image so that it completely covers the element. This can result in clipping the image if the element and its background image have different aspect ratios.

Screen Pixel Density, or DPI

The `background-size` property comes in handy for devices that have different pixel densities, such as the newest generation of smartphones. For example, the iPhone 4 has a pixel density four times higher than previous iPhones; however, to prevent pages designed for older phones from looking tiny, the browser on the iPhone 4 *behaves* as though it only has a 320×480px display. In essence, every pixel in your CSS corresponds to four screen pixels. Images are scaled up to compensate, but this means they can sometimes look a little rough compared to the smoothness of the text displayed.

To deal with this, you can provide higher-resolution images to the iPhone 4. For example, if we were providing a high-resolution image of a bicycle for the iPhone, it would measure 74×90px instead of 37×45px. However, we don't actually want it to be twice as big! We only want it to take up 37×45px worth of space. We can use `background-size` to ensure that our high-resolution image still takes up the right amount of space:

```
-webkit-background-size: 37px 45px, cover;
-moz-background-size: 37px 45px, cover;
-o-background-size: 37px 45px, cover;
background-size: 37px 45px, cover;
```

In the Background

That's all for CSS3 backgrounds and gradients. In the next chapter, we'll be looking at transforms, animations, and transitions. These allow you to add dynamic effects and movement to your pages without relying on bandwidth- and processor-heavy JavaScript.

Chapter 11

Canvas, SVG, and Drag and Drop

The HTML5 Herald is really becoming quite dynamic for an “ol’ timey” newspaper! We’ve added a video with the new `video` element, made our site available offline, added support to remember the user’s name and email address, and used geolocation to detect the user’s location.

But there’s still more we can do to make it even more fun. First, the video is a little at odds with the rest of the paper, since it’s in color. Second, the geolocation feature, while fairly speedy, could use a progress indicator that lets the user know we haven’t left them stranded. And finally, it would be nice to add just one more dynamic piece to our page. We’ll take care of all three of these using the APIs we’ll discuss in this chapter: Canvas, SVG, and Drag and Drop.

Canvas

With HTML5’s Canvas API, we’re no longer limited to drawing rectangles on our sites. We can draw anything we can imagine, all through JavaScript. This can improve the performance of our websites by avoiding the need to download images off the network. With canvas, we can draw shapes and lines, arcs and text, gradients and patterns. In addition, canvas gives us the power to manipulate pixels in images and even video. We'll start by introducing some of the basic drawing features of canvas, but then move on to using its power to transform our video—taking our modern-looking color video and converting it into conventional black and white to match the overall look and feel of *The HTML5 Herald*.

The Canvas 2D Context spec is supported in:

- Safari 2.0+
- Chrome 3.0+
- Firefox 3.0+
- Internet Explorer 9.0+
- Opera 10.0+
- iOS (Mobile Safari) 1.0+
- Android 1.0+

A Bit of Canvas History

Canvas was first developed by Apple. Since they already had a framework—Quartz 2D—for drawing in two-dimensional space, they went ahead and based many of the concepts of HTML5's canvas on that framework. It was then adopted by Mozilla and Opera, and then standardized by the WHATWG (and subsequently picked up by the W3C, along with the rest of HTML5).

There's some good news here. If you aspire to do development for the iPhone or iPad (referred to jointly as iOS), or for the Mac, what you learn in canvas should help you understand some of the basics concepts of Quartz 2D. If you already develop for the Mac or iOS and have worked with Quartz 2D, many canvas concepts will look very familiar to you.

Creating a `canvas` Element

The first step to using canvas is to add a `canvas` element to the page:

canvas/demo1.html ***(excerpt)***

```
<canvas>
Sorry! Your browser doesn't support Canvas.
</canvas>
```

The text in between the `canvas` tags will only be shown if the `canvas` element is not supported by the visitor's browser.

Since drawing on the canvas is done using JavaScript, we'll need a way to grab the element from the DOM. We'll do so by giving our `canvas` an `id`:

canvas/demo1.html *(excerpt)*

```
<canvas id="myCanvas">
Sorry! Your browser doesn't support Canvas.
</canvas>
```

All drawing on the canvas happens via JavaScript, so let's make sure we're calling a JavaScript function when our page is ready. We'll add our jQuery document ready check to a `script` element at the bottom of the page:

canvas/demo1.html *(excerpt)*

```
<script>
$('document').ready(function(){
  draw();
});
</script>
```

The `canvas` element takes both a `width` and `height` attribute, which should also be set:

canvas/demo1.html *(excerpt)*

```
<canvas id="myCanvas" width="200" height="200">
Sorry! Your browser doesn't support Canvas.
</canvas>
```

Finally, let's add a border to our canvas to visually distinguish it on the page, using some CSS. Canvas has no default styling, so it's difficult to see where it is on the page unless you give it some kind of border:

css/canvas.css *(excerpt)*

```
#myCanvas {
  border: dotted 2px black;
}
```

Now that we've styled it, we can actually view the `canvas` container on our page —Figure 11.1 shows what it looks like.

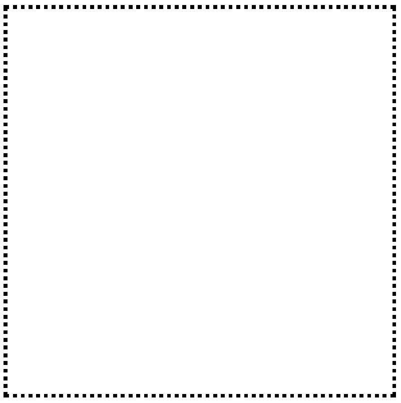


Figure 11.1. An empty canvas with a dotted border

Drawing on the Canvas

All drawing on the canvas happens via the Canvas JavaScript API. We've called a function called `draw()` when our page is ready, so let's go ahead and create that function. We'll add the function to our `script` element. The first step is to grab hold of the `canvas` element on our page:

canvas/demo1.html *(excerpt)*

```
<script>
⋮
function draw() {
  var canvas = document.getElementById("myCanvas");
}
</script>
```

Getting the Context

Once we've stored our `canvas` element in a variable, we need to set up the canvas's context. The **context** is the place where your drawing is rendered. Currently, there's only wide support for drawing to a two-dimensional context. The W3C Canvas spec defines the context in the `CanvasRenderingContext2D` object. Most methods we'll be using to draw onto the canvas are methods of this object.

We obtain our drawing context by calling the `getContext` method and passing it the string `"2d"`, since we'll be drawing in two dimensions:

canvas/demo1.html *(excerpt)*

```
function draw() {
  var canvas = document.getElementById("myCanvas");
  var context = canvas.getContext("2d");
}
```

The object that's returned by `getContext` is a `CanvasRenderingContext2D` object. In this chapter, we'll refer to it as simply "the context object" for brevity.

WebGL

WebGL is a new API for 3D graphics being managed by the Khronos Group, with a WebGL working group that includes Apple, Mozilla, Google, and Opera.

By combining WebGL with HTML5 Canvas, you can draw in three dimensions. WebGL is currently supported in Firefox 4+, Chrome 8+, and Safari 6. To learn more, see http://www.khronos.org/webgl/.

Filling Our Brush with Color

On a regular painting canvas, before you can begin, you must first saturate your brush with paint. In the HTML5 Canvas, you must do the same, and we do so with the `strokeStyle` or `fillStyle` properties. Both `strokeStyle` and `fillStyle` are set on a context object. And both take one of three values: a string representing a color, a `CanvasGradient`, or a `CanvasPattern`.

Let's start by using a color string to style the stroke. You can think of the **stroke** as the border of the shape you're going to draw. To draw a rectangle with a red border, we first define the stroke color:

canvas/demo1.html *(excerpt)*

```
function draw() {
  var canvas = document.getElementById("myCanvas");
  var context = canvas.getContext("2d");
  context.strokeStyle = "red";
}
```

To draw a rectangle with a red border and blue fill, we must also define the fill color:

canvas/demo1.html *(excerpt)*

```
function draw() {
  ⋮
  context.fillStyle = "blue";
}
```

We can use any CSS color value to set the stroke or fill color, as long as we specify it as a string: a hexadecimal value like `#00FFFF`, a color name like `red` or `blue`, or an RGB value like `rgb(0, 0, 255)`. We can even use RGBA to set a semitransparent stroke or fill color. Let's change our blue fill to blue with a 50% opacity:

canvas/demo1.html *(excerpt)*

```
function draw() {
  ⋮
  context.fillStyle = "rgba(0, 0, 255, 0.5)";
}
```

Drawing a Rectangle to the Canvas

Once we've defined the color of the stroke and the fill, we're ready to actually start drawing! Let's begin by drawing a rectangle. We can do this by calling the `fillRect` and `strokeRect` methods. Both of these methods take the X and Y coordinates where you want to begin drawing the fill or the stroke, and the width and the height of the rectangle. We'll add the stroke and fill 10 pixels from the top and 10 pixels from the left of the canvas's top left corner:

canvas/demo1.html *(excerpt)*

```
function draw() {
  ⋮
  context.fillRect(10,10,100,100);
  context.strokeRect(10,10,100,100);
}
```

This will create a semitransparent blue rectangle with a red border, like the one in Figure 11.2.

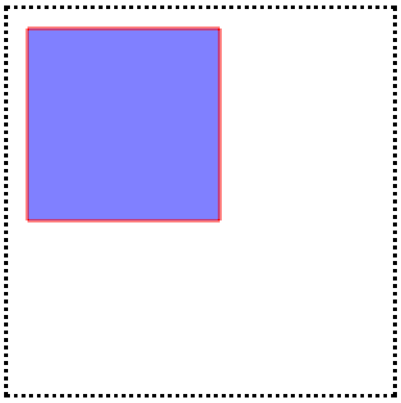


Figure 11.2. A simple rectangle—not bad for our first canvas drawing!

The Canvas Coordinate System

As you may have gathered, the coordinate system in the `canvas` element is different from the Cartesian coordinate system you learned in math class. In the canvas coordinate system, the top-left corner is (0,0). If the canvas is 200 pixels by 200 pixels, then the bottom-right corner is (200, 200), as Figure 11.3 illustrates.

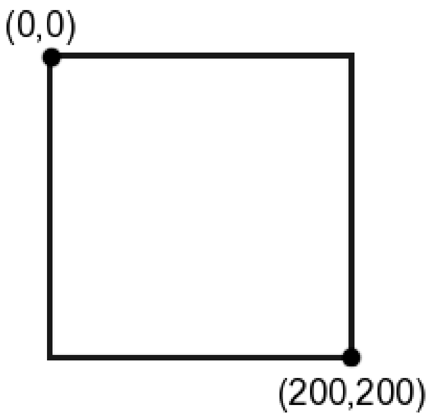


Figure 11.3. The canvas coordinate system goes top-to-bottom and left-to-right

Variations on `fillStyle`

Instead of a color as our `fillStyle`, we could have also used a `CanvasGradient` or a `CanvasPattern`.

We create a `CanvasPattern` by calling the `createPattern` method. `createPattern` takes two parameters: the image to create the pattern with, and how that image should be repeated. The repeat value is a string, and the valid values are the same as those in CSS: `repeat`, `repeat-x`, `repeat-y`, and `no-repeat`.

Instead of using a semitransparent blue `fillStyle`, let's create a pattern using our bicycle image. First, we must create an `Image` object, and set its `src` property to our image:

canvas/demo2.html *(excerpt)*

```
function draw() {
  ⋮
  var img = new Image();
  img.src = "../images/bg-bike.png";
}
```

Setting the `src` attribute will tell the browser to start downloading the image—but if we try to use it right away to create our gradient, we'll run into some problems, because the image will still be loading. So we'll use the image's `onload` property to create our pattern once the image has been fully loaded by the browser:

canvas/demo2.html *(excerpt)*

```
function draw() {
  ⋮
  var img = new Image();
  img.src = "../images/bg-bike.png";

  img.onload = function() {

  };
}
```

In our `onload` event handler, we call `createPattern`, passing it the `Image` object and the string `repeat`, so that our image repeats along both the X and Y axes. We store the results of `createPattern` in the variable `pattern`, and set the `fillStyle` to that variable:

canvas/demo2.html *(excerpt)*

```
function draw() {
  ⋮
  var img = new Image();
  img.src = "../images/bg-bike.png";
  img.onload = function() {
    pattern = context.createPattern(img, "repeat");
    context.fillStyle = pattern;
    context.fillRect(10,10,100,100);
    context.strokeRect(10,10,100,100);
  };
}
```

Anonymous Functions

You may be asking yourself, "what is that `function` statement that comes right before the call to `img.onload`?" It's an **anonymous function**. Anonymous functions are much like regular functions except, as you might guess, they don't have names.

When you see an anonymous function inside of an event listener, it means that the anonymous function is being bound to that event. In other words, the code inside that anonymous function will be run when the `load` event is fired.

Now, our rectangle's fill is a pattern made up of our bicycle image, as Figure 11.4 shows.

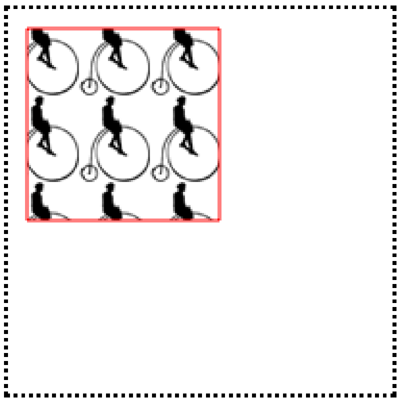


Figure 11.4. A pattern fill on a canvas

Something seriously cool has happened in web development!

We can also create a `CanvasGradient` to use as our `fillStyle`. To create a `CanvasGradient`, we call one of two methods: `createLinearGradient(x0, y0, x1, y1)` or `createRadialGradient(x0, y0, r0, x1, y1, r1)`; then we add one or more color stops to the gradient.

`createLinearGradient`'s *`x0`* and *`y0`* represent the starting location of the gradient. *`x1`* and *`y1`* represent the ending location.

To create a gradient that begins at the top of the canvas and blends the color down to the bottom, we'd define our starting point at the origin (0,0), and our ending point 200 pixels down from there (0,200):

canvas/demo3.html *(excerpt)*

```
function draw() {
  ⋮
  var gradient = context.createLinearGradient(0, 0, 0, 200);
}
```

Next, we specify our color stops. The color stop method is simply `addColorStop(offset, color)`.

The `offset` is a value between `0` and `1`. An `offset` of `0` is at the starting end of the gradient, and an `offset` of `1` is at the other end. The `color` is a string value that, as with the `fillStyle`, can be a color name, a hexadecimal color value, an `rgb()` value, or an `rgba()` value.

To make a gradient that starts as blue and begins to blend into white halfway down the gradient, we can specify a blue color stop with an `offset` of `0` and a purple color stop with an `offset` of `1`:

canvas/demo3.html *(excerpt)*

```
function draw() {
  ⋮
  var gradient = context.createLinearGradient(0, 0, 0, 200);
  gradient.addColorStop(0,"blue");
  gradient.addColorStop(1,"white");
  context.fillStyle = gradient;
  context.fillRect(10,10,100,100);
  context.strokeRect(10,10,100,100);
}
```

Figure 11.5 is the result of setting our `CanvasGradient` to be the `fillStyle` of our rectangle.

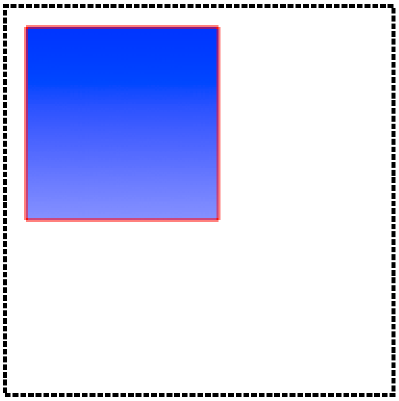


Figure 11.5. Creating a linear gradient with Canvas

Drawing Other Shapes by Creating Paths

We're not limited to drawing rectangles—we can draw any shape we can imagine! Unlike rectangles and squares, however, there's no built-in method for drawing circles, or other shapes. To draw more interesting shapes, we must first lay out the **path** of the shape.

Paths create a blueprint for your lines, arcs, and shapes, but paths are invisible until you give them a stroke! When we drew rectangles, we first set the `strokeStyle` and then called `fillRect`. With more complex shapes, we need to take three steps: lay out the path, stroke the path, and fill the path. As with drawing rectangles, we can just stroke the path, or fill the path—or we can do both.

Let's start with a simple circle:

canvas/demo4.html *(excerpt)*

```
function draw() {
  var canvas = document.getElementById("myCanvas");
  var context = canvas.getContext("2d");

  context.beginPath();
}
```

Now we need to create an **arc**. An arc is a segment of a circle; there's no method for creating a circle, but we can simply draw a 360° arc. We create it using the `arc` method:

canvas/demo4.html ***(excerpt)***

```
function draw() {
  var canvas = document.getElementById("myCanvas");
  var context = canvas.getContext("2d");

  context.beginPath();
  context.arc(50, 50, 30, 0, Math.PI*2, true);
}
```

The arguments for the `arc` method are as follows: `arc(x, y, radius, startAngle, endAngle, anticlockwise)`.

`x` and *`y`* represent where on the canvas you want the arc's path to begin. Imagine this as the center of the circle that you'll be drawing. *`radius`* is the distance from the center to the edge of the circle.

`startAngle` and *`endAngle`* represent the start and end angles along the circle's circumference that you want to draw. The units for the angles are in radians, and a circle is 2π radians. We want to draw a complete circle, so we will use 2π for the *`endAngle`*. In JavaScript, we can get this value by multiplying `Math.PI` by 2.

`anticlockwise` is an optional argument. If you wanted the arc to be drawn counterclockwise instead of clockwise, you would set this value to `true`. Since we are drawing a full circle, it doesn't matter which direction we draw it in, so we omit this argument.

Our next step is to close the path, since we've now finished drawing our circle. We do that with the `closePath` method:

canvas/demo4.html ***(excerpt)***

```
function draw() {
  var canvas = document.getElementById("myCanvas");
  var context = canvas.getContext("2d");

  context.beginPath();
  context.arc(100, 100, 50, 0, Math.PI*2, true);
  context.closePath();
}
```

Now we have a path! But unless we stroke it or fill it, we'll be unable to see it. Thus, we must set a `strokeStyle` if we would like to give it a border, and we must set a `fillStyle` if we'd like our circle to have a fill color. By default, the width of the stroke is 1 pixel—this is stored in the `lineWidth` property of the `context` object. Let's make our border a bit bigger by setting the `lineWidth` to 3:

canvas/demo4.html *(excerpt)*

```
function draw() {
  var canvas = document.getElementById("myCanvas");
  var context = canvas.getContext("2d");

  context.beginPath();
  context.arc(50, 50, 30, 0, Math.PI*2, true);
  context.closePath();
  context.strokeStyle = "red";
  context.fillStyle = "blue";
  context.lineWidth = 3;
}
```

Lastly, we fill and stroke the path. Note that this time, the method names are different than those we used with our rectangle. To fill a path, you simply call `fill`, and to stroke it you call `stroke`:

canvas/demo4.html *(excerpt)*

```
function draw() {
  var canvas = document.getElementById("myCanvas");
  var context = canvas.getContext("2d");

  context.beginPath();
  context.arc(100, 100, 50, 0, Math.PI*2, true);
  context.closePath();
  context.strokeStyle = "red";
  context.fillStyle = "blue";
  context.lineWidth = 3;
  context.fill();
  context.stroke();
}
```

Figure 11.6 shows the finished circle.

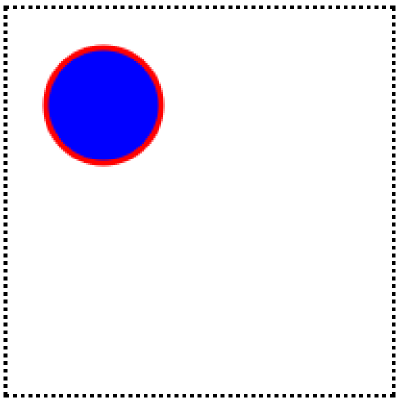


Figure 11.6. Our shiny new circle

To learn more about drawing shapes, the Mozilla Developer Network has an excellent tutorial.[1]

Saving Canvas Drawings

If we create an image programmatically using the Canvas API, but decide we'd like to have a local copy of our drawing, we can use the API's `toDataURL` method to save our drawing as a PNG or JPEG file.

To preserve the circle we just drew, we could add a new button to our HTML, and open the canvas drawing as an image in a new window once the button is clicked. To do that, let's define a new JavaScript function:

canvas/demo5.html *(excerpt)*

```
function saveDrawing() {
  var canvas = document.getElementById("myCanvas");
  window.open(canvas.toDataURL("image/png"));
}
```

Next, we'll add a button to our HTML and call our function when it's clicked:

[1] https://developer.mozilla.org/en/Canvas_tutorial/Drawing_shapes

canvas/demo5.html *(excerpt)*

```
<canvas id="myCanvas" width="200" height="200">
Sorry! Your browser doesn't support Canvas.
</canvas>
<form>
  <input type="button" name="saveButton" id="saveButton"
➥value="Save Drawing">
</form>
⋮
<script>

$('document').ready(function(){
  draw();
  $('#saveButton').click(saveDrawing);
});
⋮
```

When the button is clicked, a new window or tab opens up with a PNG file loaded into it, as shown in Figure 11.7.

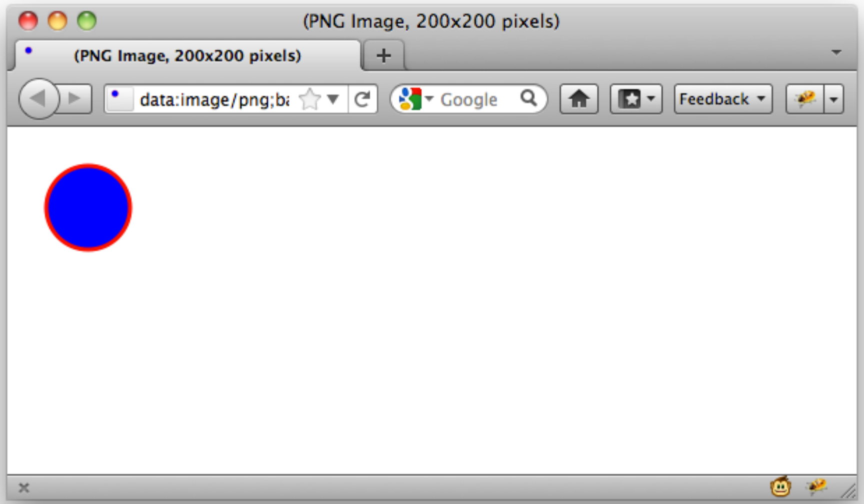


Figure 11.7. Our image loads in a new window

To learn more about saving our canvas drawings as files, see the W3C Canvas spec[2] and the "Saving a canvas image to file" section of Mozilla's Canvas code snippets.[3]

Drawing an Image to the Canvas

We can also draw images into the `canvas` element. In this example, we'll be redrawing into the canvas an image that already exists on the page.

For the sake of illustration, we'll be using the HTML5 logo[4] as our image for the next few examples. Let's start by adding it to our page in an `img` element:

canvas/demo6.html *(excerpt)*

```
<canvas id="myCanvas" width="200" height="200">
Your browser does not support canvas.
</canvas>
<img src="../images/html5-logo.png" id="myImageElem">
```

Next, after grabbing the `canvas` element and setting up the canvas's context, we can grab an image from our page via `document.getElementById`:

canvas/demo6.html *(excerpt)*

```
function draw() {
  var canvas = document.getElementById("myCanvas");
  var context = canvas.getContext("2d");
  var image = document.getElementById("myImageElem");
}
```

We'll use the same CSS we used before to make the `canvas` element visible:

css/canvas.css *(excerpt)*

```
#myCanvas {
  border: dotted 2px black;
}
```

Let's modify it slightly to space out our canvas and our image:

[2] http://www.w3.org/TR/html5/the-canvas-element.html#dom-canvas-todataurl
[3] https://developer.mozilla.org/en/Code_snippets/Canvas
[4] http://www.w3.org/html/logo/

css/canvas.css *(excerpt)*

```
#myCanvas {
  border: dotted 2px black;
  margin: 0px 20px;
}
```

Figure 11.8 shows our empty canvas next to our image.

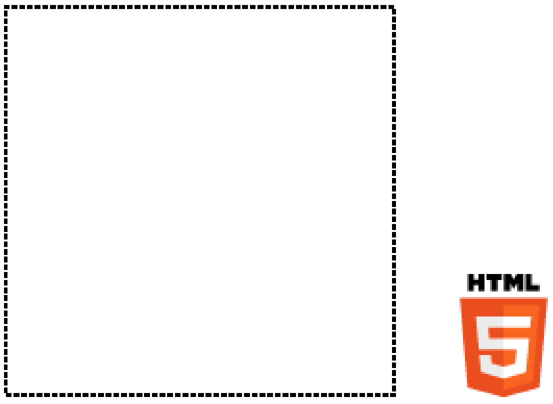


Figure 11.8. An image and a canvas sitting on a page, not doing much

We can use canvas's `drawImage` method to redraw the image from our page into the canvas:

canvas/demo6.html *(excerpt)*

```
function draw() {
  var canvas = document.getElementById("myCanvas");
  var context = canvas.getContext("2d");
  var image = document.getElementById("myImageElem");
  context.drawImage(image, 0, 0);
}
```

Because we've drawn the image to the (0,0) coordinate, the image appears in the top-left of the canvas, as you can see in Figure 11.9.

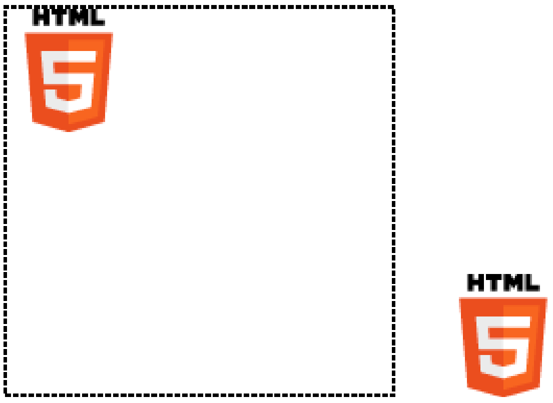


Figure 11.9. Redrawing an image inside a canvas

We could instead draw the image at the center of the canvas, by changing the X and Y coordinates that we pass to `drawImage`. Since the image is 64 by 64 pixels, and the canvas is 200 by 200 pixels, if we draw the image to `(68, 68)`,[5] the image will be in the center of the canvas, as in Figure 11.10.

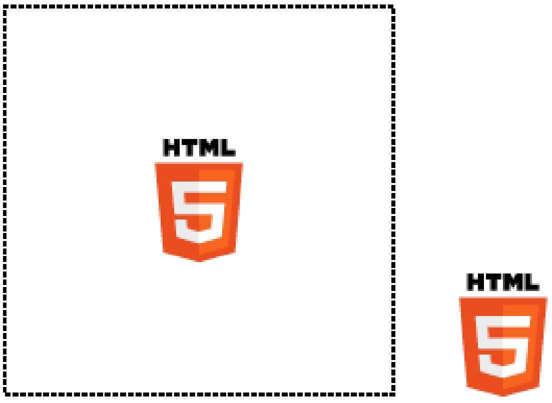


Figure 11.10. Displaying the image in the center of the canvas

Manipulating Images

Redrawing an image element from the page onto a canvas is fairly unexciting. It's really no different from using an `img` element! Where it does become interesting is how we can manipulate an image after we've drawn it into the canvas.

[5] Half of the canvas's dimensions minus half of the image's dimensions: (200/2) - (64/2) = 68.

Once we've drawn an image on the canvas, we can use the `getImageData` method from the Canvas API to manipulate the pixels of that image. For example, if we wanted to convert our logo from color to black and white, we can do so using methods in the Canvas API.

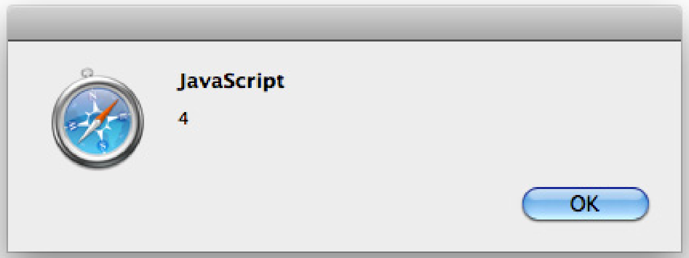
`getImageData` will return an `ImageData` object, which contains three properties: `width`, `height`, and `data`. The first two are self-explanatory, but it's the last one, `data`, that interests us.

`data` contains information about the pixels in the `ImageData` object, in the form of an array. Each pixel on the canvas will have four values in the `data` array—these correspond to that pixel's R, G, B, and A values.

The `getImageData` method allows us to examine a small section of a canvas, so let's use this feature to become more familiar with the data array. `getImageData` takes four parameters, corresponding to the four corners of a rectangular piece of the canvas we'd like to inspect. If we call `getImageData` on a very small section of the canvas, say `context.getImageData(0, 0, 1, 1)`, we'd be examining just one pixel (the rectangle from 0,0 to 1,1). The array that's returned is four items long, as it contains a red, green, blue, and alpha value for this lone pixel:

```
function draw() {
  var canvas = document.getElementById("myCanvas");
  var context = canvas.getContext("2d");
  var image = document.getElementById("myImageElem");
  // draw the image at x=0 and y=0 on the canvas
  context.drawImage(image, 68, 68);
  var imageData = context.getImageData(0, 0, 1, 1);
  var pixelData = imageData.data;
  alert(pixelData.length);
}
```

The alert prompt confirms that the data array for a one-pixel section of the canvas will have four values, as Figure 11.11 demonstrates.


JavaScript

4

Figure 11.11. The data array for a single pixel contains four values

Converting an Image from Color to Black and White

Let's look at how we'd go about using `getImageData` to convert a full color image into black and white on a canvas. Assuming we've already placed an image onto the canvas, as we did above, we can use a `for` loop to iterate through each pixel in the image, and change it to grayscale.

First, we'll call `getImageData(0,0,200,200)` to retrieve the entire canvas. Then, we need to grab the red, green, and blue values of each pixel, which appear in the array in that order:

canvas/demo7.html *(excerpt)*

```
function draw() {
  var canvas = document.getElementById("myCanvas");
  var context = canvas.getContext("2d");
  var image = document.getElementById("myImageElem");
  context.drawImage(image, 68, 68);

  var imageData = context.getImageData(0, 0, 200, 200);
  var pixelData = imageData.data;

  for (var i = 0; i < pixelData.length; i += 4) {
    var red = pixelData[i];
    var green = pixelData[i + 1];
    var blue = pixelData[i + 2];
  }
}
```

Notice that our `for` loop is incrementing `i` by `4` instead of the usual `1`. This is because each pixel takes up four values in the `imageData` array—one number each for the R, G, B, and A values.

Next, we must determine the grayscale value for the current pixel. It turns out that there's a mathematical formula for converting RGB to grayscale: you simply need to multiply each of the red, green, and blue values by some specific numbers, seen in the code block below:

canvas/demo7.html *(excerpt)*

```
⋮
for (var i = 0; i < pixelData.length; i += 4) {
  var red = pixelData[i];
  var green = pixelData[i + 1];
  var blue = pixelData[i + 2];

  var grayscale = red * 0.3 + green * 0.59 + blue * 0.11;
}
⋮
```

Now that we have the proper grayscale value, we're going to store it back into the red, green, and blue values in the `data` array:

canvas/demo7.html *(excerpt)*

```
⋮
for (var i = 0; i < pixelData.length; i += 4) {
  var red = pixelData[i];
  var green = pixelData[i + 1];
  var blue = pixelData[i + 2];

  var grayscale = red * 0.3 + green * 0.59 + blue * 0.11;

  pixelData[i] = grayscale;
  pixelData[i + 1] = grayscale;
  pixelData[i + 2] = grayscale;
}
⋮
```

So now we've modified our pixel data by individually converting each pixel to grayscale. The final step? Putting the image data we've modified back into the canvas via a method called `putImageData`. This method does exactly what you'd expect: it takes image data and writes it onto the canvas. Here's the method in action:

canvas/demo7.html *(excerpt)*

```
function draw() {
  var canvas = document.getElementById("myCanvas");
  var context = canvas.getContext("2d");
  var image = document.getElementById("myImageElem");
  context.drawImage(image, 60, 60);

  var imageData = context.getImageData(0, 0, 200, 200);
  var pixelData = imageData.data;

  for (var i = 0; i < pixelData.length; i += 4) {
    var red = pixelData[i];
    var green = pixelData[i + 1];
    var blue = pixelData[i + 2];

    var grayscale = red * 0.3 + green * 0.59 + blue * 0.11;

    pixelData[i] = grayscale;
    pixelData[i + 1] = grayscale;
    pixelData[i + 2] = grayscale;
  }
  context.putImageData(imageData, 0, 0);
}
```

With that, we've drawn a black-and-white version of the validation image into the canvas.

Security Errors with `getImageData`

If you tried out this code in Chrome or Firefox, you may have noticed that it failed to work—the image on the canvas is in color. That's because in these two browsers, if you try to convert an image on your desktop in an HTML file that's also on your desktop, an error will occur in `getImageData`. The error is a security error, though in our case it's an unnecessary one.

The true security issue that Chrome and Firefox are attempting to stop is a user on one domain from manipulating images on another domain. For example, stopping me from loading an official logo from http://google.com/ and then manipulating the pixel data.

The W3C Canvas spec[6] describes it this way:

> Information leakage can occur if scripts from one origin can access information (e.g. read pixels) from images from another origin (one that isn't the same). To mitigate this, `canvas` elements are defined to have a flag indicating whether they are origin-clean.

This origin-clean flag will be set to `false` if the image you want to manipulate is on a different domain from the JavaScript doing the manipulating. Unfortunately, in Chrome and Firefox, this origin-clean flag is also set to `false` while you're testing from files on your hard drive—they're seen as files living on different domains.

If you want to test pixel manipulation using canvas in Firefox or Chrome, you'll need to either test it on a web server running on your computer (http://localhost/), or test it online on a real web server.

Manipulating Video with Canvas

We can take the code we've already written to convert a color image to black and white, and enhance it to make our color *video* black and white, to match the old-timey feel of *The HTML5 Herald* page. We'll do this in a new, separate JavaScript file called **videoToBW.js**, so that we can include it on the site's home page.

The file begins, as always, by setting up the canvas and the context:

js/videoToBW.js ***(excerpt)***

```
function makeVideoOldTimey ()
{
  var video = document.getElementById("video");
  var canvas = document.getElementById("canvasOverlay");
  var context = canvas.getContext("2d");
}
```

Next, we'll add a new event listener to react to the `play` event firing on the `video` element.

We want to call a `draw` function when the video begins playing. To do so, we'll add an event listener to our video `element` that responds to the `play` event:

[6] http://dev.w3.org/html5/2dcontext/

js/videoToBW.js *(excerpt)*

```
function makeVideoOldTimey ()
{
  var video = document.getElementById("video");
  var canvas = document.getElementById("canvasOverlay");
  var context = canvas.getContext("2d");

  video.addEventListener("play", function(){
    draw(video,context,canvas);
  },false);
}
```

The `draw` function will be called when the `play` event fires, and it will be passed the `video`, `context`, and `canvas` objects. We're using an anonymous function here instead of a normal named function because we can't actually pass parameters to named functions when declaring them as event handlers.

Since we want to pass several parameters to the `draw` function—*`video`*, *`context`*, and *`canvas`*—we must call it from inside an anonymous function.

Let's look at the `draw` function:

js/videoToBW.js *(excerpt)*

```
function draw(video, context, canvas)
{
  if (video.paused || video.ended)
  {
    return false;
  }

  drawOneFrame(video, context, canvas);
}
```

Before doing anything else, we check to see if the video is paused or has ended, in which case we'll just cut the function short by returning `false`. Otherwise, we continue onto the `drawOneFrame` function. The `drawOneFrame` function is nearly identical to the code we had above for converting an image from color to black and white, except that we're drawing the `video` element onto the canvas instead of a static image:

js/videoToBW.js *(excerpt)*

```
function drawOneFrame(video, context, canvas){
  // draw the video onto the canvas
  context.drawImage(video, 0, 0, canvas.width, canvas.height);

  var imageData = context.getImageData(0, 0, canvas.width,
➥canvas.height);
  var pixelData = imageData.data;
  // Loop through the red, green and blue pixels,
  // turning them grayscale
  for (var i = 0; i < pixelData.length; i += 4) {
    var red = pixelData[i];
    var green = pixelData[i + 1];
    var blue = pixelData[i + 2];
    //we'll ignore the alpha value, which is in position i+3

    var grayscale = red * 0.3 + green * 0.59 + blue * 0.11;

    pixelData[i] = grayscale;
    pixelData[i + 1] = grayscale;
    pixelData[i + 2] = grayscale;
  }

  imageData.data = pixelData;

  context.putImageData(imageData, 0, 0);
}
```

After we've drawn one frame, what's the next step? We need to draw another frame! The `setTimeout` method allows us to keep calling the `draw` function over and over again, without pause: the final parameter is the value for delay—or how long, in milliseconds, the browser should wait before calling the function. Because it's set to `0`, we are essentially running `draw` continuously. This goes on until the video has either ended, or been paused:

js/videoToBW.js *(excerpt)*

```
function draw(video, context, canvas) {
  if (video.paused || video.ended)
  {
    return false;
  }
  var status = drawOneFrame(video, context, canvas);

  // Start over!
  setTimeout(function(){ draw(video, context, canvas); }, 0);
}
```

The net result? Our color video of a plane taking off now plays in black and white!

Displaying Text on the Canvas

If we were to view *The HTML5 Herald* from a file on our computer, we'd encounter security errors in Firefox and Chrome when trying to manipulate an entire video, as we would a simple image.

We can add a bit of error-checking in order to make our video work anyway, even if we view it from our local machine in Chrome or Firefox.

The first step is to add a `try/catch` block to catch the error:

js/videoToBW.js *(excerpt)*

```
function drawOneFrame(video, context, canvas){
  context.drawImage(video, 0, 0, canvas.width, canvas.height);

  try {
    var imageData = context.getImageData(0, 0, canvas.width,
➥canvas.height);
    var pixelData = imageData.data;
    for (var i = 0; i < pixelData.length; i += 4) {
      var red = pixelData[i];
      var green = pixelData[i + 1];
      var blue = pixelData[i + 2];
      var grayscale = red * 0.3 + green * 0.59 + blue * 0.11;
      pixelData[i] = grayscale;
      pixelData[i + 1] = grayscale;
      pixelData[i + 2] = grayscale;
    }

    imageData.data = pixelData;
    context.putImageData(imageData, 0, 0);
  }
  catch (err) {
    // error handling code will go here
  }
}
```

When an error occurs in trying to call `getImageData`, it would be nice to give some sort of message to the user in order to give them a hint about what may be going wrong. We'll do just that, using the `fillText` method of the Canvas API.

Before we write any text to the canvas, we should clear what's already drawn to it. We've already drawn the first frame of the video into the canvas using the call to `drawImage`. How can we clear that out?

It turns out that if we reset the width or height on the canvas, the canvas will be cleared. So, let's reset the width:

js/videoToBW.js ***(excerpt)***

```
function drawOneFrame(video, context, canvas){
  context.drawImage(video, 0, 0, canvas.width, canvas.height);

  try {
    ⋮
  }
  catch (err) {
    canvas.width = canvas.width;
  }
}
```

Next, let's change the background color from black to transparent, since the `canvas` element is positioned on top of the video:

js/videoToBW.js ***(excerpt)***

```
function drawOneFrame(video, context, canvas){
  context.drawImage(video, 0, 0, canvas.width, canvas.height);

  try {
    ⋮
  }
  catch (err) {
    canvas.width = canvas.width;
    canvas.style.backgroundColor = "transparent";
  }
}
```

Before we can draw any text to the now transparent canvas, we first must set up the style of our text—similar to what we did with paths earlier. We do that with the `fillStyle` and `textAlign` methods:

js/videoToBW.js *(excerpt)*

```
videoToBW.js (excerpt)
function drawOneFrame(video, context, canvas){
  context.drawImage(video, 0, 0, canvas.width, canvas.height);

  try {
    ⋮
  }
  catch (err) {
    canvas.width = canvas.width;
    canvas.style.backgroundColor = "transparent";
    context.fillStyle = "white";
    context.textAlign = "left";
  }
}
```

We must also set the font we'd like to use. The `font` property of the context object works the same way the CSS `font` property does. We'll specify a font size of 18px and a comma-separated list of font families:

js/videoToBW.js *(excerpt)*

```
function drawOneFrame(video, context, canvas){
  context.drawImage(video, 0, 0, canvas.width, canvas.height);

  try {
    ⋮
  }
  catch (err) {
    canvas.width = canvas.width;
    canvas.style.backgroundColor = "transparent";
    context.fillStyle = "white";
    context.textAlign = "left";
    context.font = "18px LeagueGothic, Tahoma, Geneva, sans-serif";
  }
}
```

Notice that we're using League Gothic; any fonts you've included with `@font-face` are also available for you to use on your canvas. Finally, we draw the text. We use a method of the context object called `fillText`, which takes the text to be drawn and the (x,y) coordinates where it should be placed. Since we want to write out a fairly long message, we'll split it up into several sections, placing each one on the canvas separately:

js/videoToBW.js *(excerpt)*

```
function drawOneFrame(video, context, canvas){
  context.drawImage(video, 0, 0, canvas.width, canvas.height);

  try {
    ⋮
  }
  catch (err) {
    canvas.width = canvas.width;
    canvas.style.backgroundColor = "transparent";
    context.fillStyle = "white";
    context.textAlign = "left";
    context.font = "18px LeagueGothic, Tahoma, Geneva, sans-serif";
    context.fillText("There was an error rendering ", 10, 20);
    context.fillText("the video to the canvas.", 10, 40);
    context.fillText("Perhaps you are viewing this page from", 10,
➥70);
    context.fillText("a file on your computer?", 10, 90);
    context.fillText("Try viewing this page online instead.", 10,
➥130);

    return false;
  }
}
```

As a last step, we return `false`. This lets us check in the `draw` function whether an exception was thrown. If it was, we want to stop calling `drawOneFrame` for each video frame, so we exit the `draw` function:

js/videoToBW.js *(excerpt)*

```
function draw(video, context, canvas) {
    if (video.paused || video.ended)
  {
    return false;
  }

  var status = drawOneFrame(video, context, canvas);

  if (status == false)
  {
    return false;
  }
  // Start over!
  setTimeout(function(){ draw(video, context, canvas); }, 0);
}
```

Accessibility Concerns

A major downside of canvas in its current form is its lack of accessibility. The canvas doesn't create a DOM node, is not a text-based format, and is thus essentially invisible to tools like screen readers. For example, even though we wrote text to the canvas in our last example, that text is essentially no more than a bunch of pixels, and is therefore inaccessible.

The HTML5 community is aware of these failings, and while no solution has been finalized, debates on how canvas could be changed to make it accessible are underway. You can read a compilation of the arguments and currently proposed solutions on the W3C's wiki page.[7]

Further Reading

To read more about canvas and the Canvas API, here are a couple of good resources:

- "HTML5 canvas—the basics" at Dev.Opera[8]
- Safari's HTML5 Canvas Guide[9]

[7] http://www.w3.org/html/wg/wiki/AddedElementCanvas
[8] http://dev.opera.com/articles/view/html-5-canvas-the-basics/
[9] http://developer.apple.com/library/safari/#documentation/AudioVideo/Conceptual/HTML-canvas-guide/Introduction/Introduction.html

SVG

We already learned a bit about SVG back in Chapter 7, when we used SVG files as a fallback for gradients in IE9 and older versions of Opera. In this chapter, we'll dive into SVG in more detail and learn how to use it in other ways.

First, a quick refresher: SVG stands for Scalable Vector Graphics. SVG is a specific file format that allows you to describe vector graphics using XML. A major selling point of vector graphics in general is that, unlike bitmap images (such as GIF, JPEG, PNG, and TIFF), vector images preserve their shape even as you blow them up or shrink them down. We can use SVG to do many of the same tasks we can do with canvas, including drawing paths, shapes, text, gradients, and patterns. There are also some very useful open source tools relevant to SVG, some of which we will leverage in order to add a spinning progress indicator to *The HTML5 Herald*'s geolocation widget.

Basic SVG, including using SVG in an HTML `img` element, is supported in:

- Safari 3.2+
- Chrome 6.0+
- Firefox 4.0+
- Internet Explorer 9.0+
- Opera 10.5+

There is currently no support for SVG in Android's browser.

XML

XML stands for eXtensible Markup Language. Like HTML, it's a markup language, which means it's a system meant to annotate text. Just as we can use HTML tags to wrap our content and give it meaning, so can XML tags be used to describe the content of files.

Unlike canvas, images created with SVG are available via the DOM. This allows technologies like screen readers to see what's present in an SVG object through its DOM node—and it also allows you to inspect SVG using your browser's developer tools. Since SVG is an XML file format, it's also more accessible to search engines than canvas.

Drawing in SVG

Drawing a circle in SVG is arguably easier than drawing a circle with canvas. Here's how we do it:

images/circle.svg

```
<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 400 400">
  <circle cx="50" cy="50" r="25" fill="red"/>
</svg>
```

The `viewBox` attribute defines the starting location, width, and height of the SVG image.

The `circle` element defines a circle, with `cx` and `cy` the X and Y coordinates of the center of the circle. The radius is represented by `r`, while `fill` is for the fill style.

To view an SVG file, you simply open it via the **File** menu in any browser that supports SVG. Figure 11.12 shows what our circle looks like.

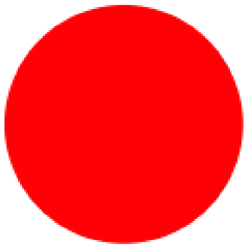


Figure 11.12. A circle drawn using SVG

We can also draw rectangles in SVG, and add a stroke to them, as we did with canvas.

This time, let's take advantage of SVG being an XML—and thus text-based—file format, and utilize the `desc` tag, which allows us to provide a description for the image we're going to draw:

images/rectangle.svg *(excerpt)*

```
<svg xmlns="http://www.w3.org/2000/svg" viewbox="0 0 400 400">
  <desc>Drawing a rectangle</desc>
</svg>
```

Next, we populate the `rect` tag with a number of attributes that describe the rectangle. This includes the X and Y coordinate where the rectangle should be drawn, the width and height of the rectangle, the fill, the stroke, and the width of the stroke:

images/rectangle.svg

```
<svg xmlns="http://www.w3.org/2000/svg" viewbox="0 0 400 400">
  <desc>Drawing a rectangle</desc>
  <rect x="10" y="10" width="100" height="100"
        fill="blue" stroke="red" stroke-width="3"  />
</svg>
```

Figure 11.13 shows our what our rectangle looks like.

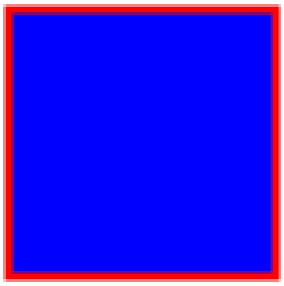


Figure 11.13. A rectangle drawn with SVG

Unfortunately, it's not always this easy. If you want to create complex shapes, the code begins to look a little scary. Figure 11.14 shows a fairly simple-looking star image from http://openclipart.org/:

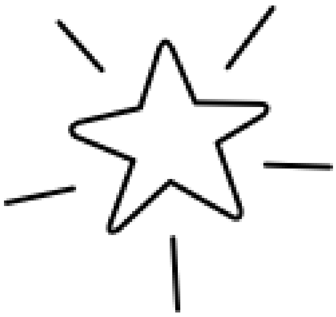


Figure 11.14. A line drawing of a star

And here are just the first few lines of SVG for this image:

```
<svg xmlns="http://www.w3.org/2000/svg"
  width="122.88545" height="114.88568">
<g
  inkscape:label="Calque 1"
  inkscape:groupmode="layer"
  id="layer1"
  transform="translate(-242.42282,-449.03699)">
  <g
    transform="matrix(0.72428496,0,0,0.72428496,119.87078,183.8127)"
    id="g7153">
    <path
      style="fill:#ffffff;fill-opacity:1;stroke:#000000;stroke-width
➥:2.761343;stroke-linecap:round;stroke-linejoin:round;stroke-miterl
➥imit:4;stroke-opacity:1;stroke-dasharray:none;stroke-dashoffset:0"
      d="m 249.28667,389.00422 -9.7738,30.15957 -31.91999,7.5995 c -
➥2.74681,1.46591 -5.51239,2.92436 -1.69852,6.99979 l 30.15935,12.57
➥796 -11.80876,32.07362 c -1.56949,4.62283 -0.21957,6.36158 4.24212
➥,3.35419 l 26.59198,-24.55691 30.9576,17.75909 c 3.83318,2.65893 6
➥.12086,0.80055 5.36349,-3.57143 l -12.10702,-34.11764 22.72561,-13
➥.7066 c 2.32805,-1.03398 5.8555,-6.16054 -0.46651,-6.46042 l -33.5
➥0135,-0.66887 -11.69597,-27.26175 c -2.04282,-3.50583 -4.06602,-7.
➥22748 -7.06823,-0.1801 z"
      id="path7155"
      inkscape:connector-curvature="0"
      sodipodi:nodetypes="cccccccccccccccc" />
⋮
```

Eek!

Using Inkscape to Create SVG Images

To save ourselves some work (and sanity), instead of creating SVG images by hand, we can use an image editor to help. One open source tool that you can use to make SVG images is Inkscape. Inkscape is an open source vector graphics editor that outputs SVG. Inkscape is available for download at http://inkscape.org/.

For our progress-indicating spinner, instead of starting from scratch, we've searched the public domain to find a good image from which to begin. A good resource to know about for public domain images is http://openclipart.org/, where you can find images that are copyright-free and free to use. The images have been donated by the creators for use in the public domain, even for commercial purposes, without the need to ask for permission.

We will be using an image of three arrows as the basis of our progress spinner, shown in Figure 11.15. The original can be found at openclipart.org.[10]

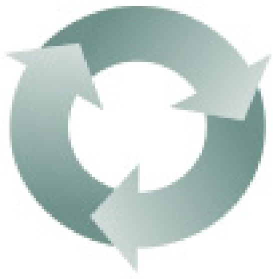


Figure 11.15. The image we'll be using for our progress indicator

SVG Filters

To make our progress spinner match our page a bit better, we can use a filter in Inkscape to make it black and white. Start by opening the file in Inkscape, then choose **Filters** > **Color** > **Moonarize**.

You may notice if you test out *The HTML5 Herald* in Safari that our black-and-white spinner is still … in color. That's because SVG filters are a specific feature of SVG yet to be implemented in Safari 5, though it will be part of Safari 6. SVG filters are

[10] http://www.openclipart.org/people/JoBrad/arrows_3_circular_interlocking.svg

supported in Firefox, Chrome, and Opera. They're currently unsupported in all versions of Safari, Internet Explorer, the Android browser, and iOS.

A safer approach would be to avoid using filters, and instead simply modify the color of the original image.

We can do this in Inkscape by selecting the three arrows in the **spinner.svg** image, and then selecting **Object** > **Fill and Stroke**. The **Fill and Stroke** menu will appear on the right-hand side of the screen, as seen in Figure 11.16.

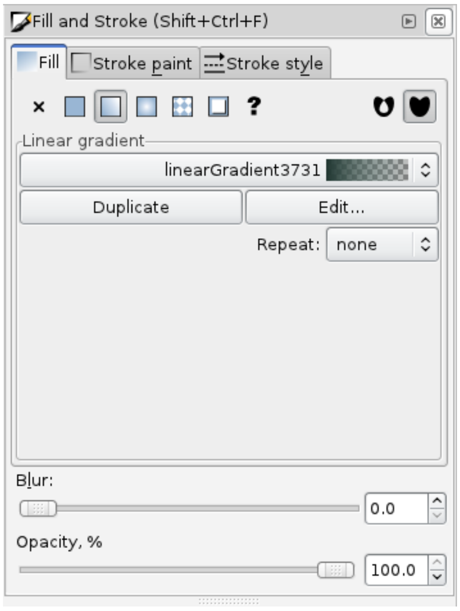


Figure 11.16. Modifying color using **Fill and Stroke**

From this menu, we can choose to edit the existing linear gradient by clicking the **Edit** button. We can then change the **Red**, **Green**, and **Blue** values all to 0 to make our image black and white.

We've saved the resulting SVG as **spinnerBW.svg**.

Using the Raphaël Library

Raphaël[11] is an open source JavaScript library that wraps around SVG. It makes drawing and animating with SVG much easier than with SVG alone.

Drawing an Image to Raphaël's Container

Much as with canvas, you can also draw images into a container you create using Raphaël.

Let's add a `div` to our main index file, which we'll use as the container for the SVG elements we'll create using Raphaël. We've named this `div` spinner:

css/styles.css (excerpt)

```
<article id="ad4">
  <div id="mapDiv">
    <h1 id="geoHeading">Where in the world are you?</h1>
    <form id="geoForm">
      <input type="button" id="geobutton" value="Tell us!">
    </form>
    <div id="spinner"></div>
  </div>
</article>
```

We have styled this `div` to be placed in the center of the parent `mapDiv` using the following CSS:

css/styles.css (excerpt)

```
#spinner {
  position:absolute;
  top:8px;
  left:55px;
}
```

Now, in our geolocation JavaScript, let's put the spinner in place while we're fetching the map. The first step is to turn our `div` into a Raphaël container. This is as simple as calling the `Raphael` method, and passing in the element we'd like to use, along with a width and height:

[11] http://raphaeljs.com/

js/geolocation.js *(excerpt)*

```
function determineLocation(){
  if (navigator.onLine) {
    if (Modernizr.geolocation) {
      navigator.geolocation.getCurrentPosition(displayOnMap);

      var container = Raphael(document.getElementById("spinner"),
➥125, 125);
```

Next, we draw the spinner SVG image into the newly created container with the Raphaël method `image`, which is called on a Raphaël container object. This method takes the path to the image, the starting coordinates where the image should be drawn, and the width and height of the image:

js/geolocation.js *(excerpt)*

```
var container = Raphael(document.getElementById("spinner"),125,125);
var spinner = container.image("images/spinnerBW.svg",0,0,125,125);
```

With this, our spinner image will appear when we click on the button in the geolocation widget.

Rotating a Spinner with Raphaël

Now that we have our container and the spinner SVG image drawn into it, we want to animate the image to make it spin. Raphaël has animation features built in with the `animate` method. Before we can use this method, though, we first need to tell it which attribute to animate. Since we want to rotate our image, we'll create an object that specifies how many degrees of rotation we want.

We create a new object `attrsToAnimate`, specifying that we want to animate the rotation, and we want to rotate by 720 degrees (two full turns):

js/geolocation.js *(excerpt)*

```
var container = Raphael(document.getElementById("spinner"),125,125);
var spinner = container.image("images/spinnerBW.png",0,0,125,125);
var attrsToAnimate = { rotation: "720" };
```

The final step is to call the `animate` method, and specify how long the animation should last. In our case, we will let it run for a maximum of sixty seconds. Since `animate` takes its values in milliseconds, we'll pass it `60000`:

js/geolocation.js *(excerpt)*

```
var container = Raphael(document.getElementById("spinner"),125,125);
var spinner = container.image("images/spinnerBW.png",0,0,125,125);
var attrsToAnimate = { rotation: "720" };
spinner.animate(attrsToAnimate, 60000);
```

That's great! We now have a spinning progress indicator to keep our visitors in the know while our map is loading. There's still one problem though: it remains after the map has loaded. We can fix this by adding one line to the beginning of the existing `displayOnMap` function:

js/geolocation.js *(excerpt)*

```
function displayOnMap(position){
  document.getElementById("spinner").style.visibility = "hidden";
```

This line sets the `visibility` property of the spinner element to `hidden`, effectively hiding the spinner `div` and the SVG image we've loaded into it.

Canvas versus SVG

Now that we've learned about canvas and SVG, you may be asking yourself, which is the right one to use? The answer is: it depends on what you're doing.

Both canvas and SVG allow you to draw custom shapes, paths, and fonts. But what's unique about each?

Canvas allows for pixel manipulation, as we saw when we turned our video from color to black and white. One downside of canvas is that it operates in what's known as **immediate mode**. This means that if you ever want to add more to the canvas, you can't simply add to what's already there. Everything must be redrawn from scratch each time you want to change what's on the canvas. There's also no access to what's drawn on the canvas via the DOM. However, canvas does allow you to save the images you create to a PNG or JPEG file.

By contrast, what you draw to SVG is accessible via the DOM, because its mode is **retained mode**—the structure of the image is preserved in the XML document that describes it. SVG also has, at this time, a more complete set of tools to help you work with it, like the Raphaël library and Inkscape. However, since SVG is a file format—rather than a set of methods that allows you to dynamically draw on a surface—you can't manipulate SVG images the way you can manipulate pixels on canvas. It would have been impossible, for example, to use SVG to convert our color video to black and white as we did with canvas.

In summary, if you need to paint pixels to the screen, and have no concerns about the ability to retrieve and modify your shapes, canvas is probably the better choice. If, on the other hand, you need to be able to access and change specific aspects of your graphics, SVG might be more appropriate.

It's also worth noting that neither technology is appropriate for static images—at least not while browser support remains a stumbling block. In this chapter, we've made use of canvas and SVG for a number of such static examples, which is fine for the purpose of demonstrating what they can do. But in the real world, they're only really appropriate for cases where user interaction defines what's going to be drawn.

Drag and Drop

In order to add one final dynamic effect to our site, we're going to examine the new Drag and Drop API. This API allows us to specify that certain elements are **draggable**, and then specify what should happen when these draggable elements are dragged over or dropped onto other elements on the page.

Drag and Drop is supported in:

- Safari 3.2+
- Chrome 6.0+
- Firefox 3.5+ (there is an older API that was supported in Firefox 3.0)
- Internet Explorer 7.0+
- Android 2.1+

There is currently no support for Drag and Drop in Opera. The API is unsupported by design in iOS, as Apple directs you to use the DOM Touch API[12] instead.

There are two major kinds of functionality you can implement with Drag and Drop: dragging files from your computer into a web page—in combination with the File API—or dragging elements into other elements on the same page. In this chapter, we'll focus on the latter.

Drag and Drop and the File API

If you'd like to learn more about how to combine Drag and Drop with the File API, in order to let users drag files from their desktop onto your websites, an excellent guide can be found at the Mozilla Developer Network.[13]

The File API is currently only supported in Firefox 3.6+ and Chrome.

There are several steps to adding Drag and Drop to your page:

1. Set the `draggable` attribute on any HTML elements you'd like to be draggable.
2. Add an event listener for the `dragstart` event on any draggable HTML elements.
3. Add an event listener for the `dragover` and `drop` events on any elements you want to have accept dropped items.

Feeding the WAI-ARIA Cat

In order to add a bit of fun and frivolity to our page, let's add a few images of mice, so that we can then drag them onto our cat image and watch the cat react and devour them. Before you start worrying (or call the SPCA), rest assured that we mean, of course, computer mice. We'll use another image from OpenClipArt for our mice.[14]

The first step is to add these new images to our **index.html** file. We'll give each mouse image an `id` as well:

[12] http://developer.apple.com/library/safari/#documentation/AppleApplications/Reference/SafariWebContent/HandlingEvents/HandlingEvents.html
[13] https://developer.mozilla.org/en/Using_files_from_web_applications
[14] http://www.openclipart.org/detail/111289

js/dragDrop.js *(excerpt)*

```
<article id="ac3">
  <hgroup>
    <h1>Wai-Aria? HAHA!</h1>
    <h2>Form Accessibility</h2>
  </hgroup>

  <img src="images/cat.png" alt="WAI-ARIA Cat">

  <div class="content">
    <p id=”mouseContainer”>
      <img src="images/computer-mouse-pic.svg" width="30"
➥alt="mouse treat" id="mouse1">
      <img src="images/computer-mouse-pic.svg" width="30"
➥alt="mouse treat" id="mouse2">
      <img src="images/computer-mouse-pic.svg" width="30"
➥alt="mouse treat" id="mouse3">
    </p>
⋮
```

Figure 11.17 shows our images in their initial state.

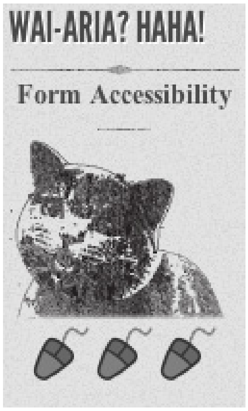


Figure 11.17. Three little mice, ready to be fed to the WAI-ARIA cat

Making Elements Draggable

The next step is to make our images draggable. In order to do that, we add the `draggable` attribute to them, and set the value to `true`:

js/dragDrop.js *(excerpt)*

```
<img src="images/computer-mouse-pic.svg" width="30"
➥alt="mouse treat" id="mouse1" draggable="true">
<img src="images/computer-mouse-pic.svg" width="30"
➥alt="mouse treat" id="mouse2" draggable="true">
<img src="images/computer-mouse-pic.svg" width="30"
➥alt="mouse treat" id="mouse3" draggable="true">
```

`draggable` Must Be Set

Note that `draggable` is *not* a Boolean attribute; you have to explicitly set it to `true`.

Now that we have set `draggable` to `true`, we have to set an event listener for the `dragstart` event on each image. We'll use jQuery's `bind` method to attach the event listener:

js/dragDrop.js *(excerpt)*

```
$('document').ready(function() {
  $('#mouseContainer img').bind('dragstart', function(event) {
    // handle the dragstart event
  });
});
```

The `DataTransfer` Object

`DataTransfer` objects are one of the new objects outlined in the Drag and Drop API. These objects allow us to set and get data about the objects that are being dragged. Specifically, `DataTransfer` lets us define two pieces of information:

1. the type of data we're saving about the draggable element
2. the value of the data itself

In the case of our draggable mouse images, we want to be able to store the `id` of these images, so we know which image is being dragged around.

To do this, we first need to tell `DataTransfer` that we want to save some plain text by passing in the string `text/plain`. Then we give it the `id` of our mouse image:

js/dragDrop.js *(excerpt)*

```
$('#mouseContainer img').bind('dragstart', function(event) {
  event.originalEvent.dataTransfer.setData("text/plain",
➥event.target.getAttribute('id'));
});
```

When an element is dragged, we save the `id` of the element in the `DataTransfer` object, to be used again once the element is dropped. The target property of a `dragstart` event will be the element that's being dragged.

`dataTransfer` and jQuery

The jQuery library's `Event` object only gives you access to properties it knows about. This causes problems when you're using new native events like `DataTransfer`; trying to access the `dataTransfer` property of a jQuery event will result in an error. However, you can always retrieve the original DOM event by calling the `originalEvent` method on your jQuery event, as we did above. This will give you access to any properties your browser supports—in this case that includes the new `DataTransfer` object.

Of course, this isn't an issue if you're rolling your own JavaScript from scratch!

Accepting Dropped Elements

Now our mouse images are set up to be dragged. Yet, when we try to drag them around, we're unable to drop them anywhere, which is no fun.

The reason is that by default, elements on the page aren't set up to receive dragged items. In order to override the default behavior on a specific element, we must stop it from happening. We can do that by creating two more event listeners.

The two events we need to monitor for are `dragover` and `drop`. As you'd expect, `dragover` fires when you drag something over an element, and `drop` fires when you drop something on it.

We'll need to prevent the default behavior for both these events—since the default prohibits you from dropping an element.

Let's start by adding an `id` to our cat image so that we can bind event handlers to it:

js/dragDrop.js *(excerpt)*

```
<article id="ac3">
  <hgroup>
    <h1>Wai-Aria? HAHA!</h1>
    <h2 id="catHeading">Form Accessibility</h2>
  </hgroup>

  <img src="images/cat.png" id="cat" alt="WAI-ARIA Cat">
```

You may have noticed that we also gave an `id` to the `h2` element. This is so we can change this text once we've dropped a mouse onto the cat.

Now, let's handle the `dragover` event:

js/dragDrop.js *(excerpt)*

```
$('#cat').bind('dragover', function(event) {
  event.preventDefault();
});
```

That was easy! In this case, we merely ensured that the mouse picture can actually be dragged over the cat picture. We simply need to prevent the default behavior —and jQuery's `preventDefault` method serves this purpose exactly.

The code for the `drop` handler is a bit more complex, so let us review it piece by piece. Our first task is to figure out what the cat should say when a mouse is dropped on it. In order to demonstrate that we can retrieve the `id` of the dropped mouse from the `DataTransfer` object, we'll use a different phrase for each mouse, regardless of the order they're dropped in. We've given three cat-appropriate options: "MEOW!", "Purr ...", and "NOMNOMNOM."

We'll store these options inside an object called `mouseHash`. The first step is to declare our object:

js/dragDrop.js *(excerpt)*

```
$('#cat').bind('drop', function(event) {
  var mouseHash = {};
```

Next, we're going to take advantage of JavaScript's objects allowing us to store key/value pairs inside them, as well as storing each response in the `mouseHash` object, associating each response with the `id` of one of the mouse images:

js/dragDrop.js *(excerpt)*

```
$('#cat').bind('drop', function(event) {
  var mouseHash = {};
  mouseHash['mouse1'] = "NOMNOMNOM";
  mouseHash['mouse2'] = "MEOW!";
  mouseHash['mouse3'] = "Purr...";
```

Our next step is to grab the `h2` element that we'll change to reflect the cat's response:

js/dragDrop.js *(excerpt)*

```
var catHeading = document.getElementById('catHeading');
```

Remember when we saved the `id` of the dragged element to the `DataTransfer` object using `setData`? Well, now we want to retrieve that `id`. If you guessed that we will need a method called `getData` for this, you guessed right:

js/dragDrop.js *(excerpt)*

```
var item = event.originalEvent.dataTransfer.getData("text/plain");
```

Note that we've stored the mouse's `id` in a variable called `item`. Now that we know which mouse was dropped, and we have our heading, we just need to change the text to the appropriate response:

js/dragDrop.js *(excerpt)*

```
catHeading.innerHTML = mouseHash[item];
```

We use the information stored in the `item` variable (the dragged mouse's `id`) to retrieve the correct message for the `h2` element. For example, if the dragged mouse is `mouse1`, calling `mouseHash[item]` will retrieve "NOMNOMNOM" and set that as the `h2` element's text.

Given that the mouse has now been "eaten," it makes sense to remove it from the page:

js/dragDrop.js *(excerpt)*

```
var mousey = document.getElementById(item);
mousey.parentNode.removeChild(mousey);
```

Last but not least, we must also prevent the default behavior of not allowing elements to be dropped on our cat image, as before:

js/dragDrop.js *(excerpt)*

```
event.preventDefault();
```

Figure 11.18 shows our happy cat, with one mouse to go.

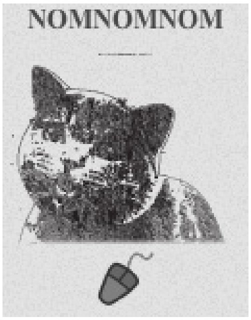


Figure 11.18. This cat's already eaten two mice

Further Reading

We've only touched on the basics of the Drag and Drop API, to give you a taste of what's available. We've shown you how you can use `DataTransfer` to pass data from your dragged items to their drop targets. What you do with this power is up to you!

To learn more about the Drag and Drop API, here are a couple of good resources:

- The Mozilla Developer Center's Drag and Drop documentation[15]
- The W3C's Drag and Drop specification[16]

[15] https://developer.mozilla.org/En/DragDrop/Drag_and_Drop
[16] http://dev.w3.org/html5/spec/dnd.html

That's All, Folks!

With these final bits of interactivity, our work on *The HTML5 Herald* has come to an end, and your journey into the world of HTML5 and CSS3 is well on its way! We've tried to provide a solid foundation of knowledge about as many of the cool new features available in today's browsers as possible—but how you build on that is up to you.

We hope we've given you a clear picture of how most of these features can be used today on real projects. Many are already well-supported, and browser development is once again progressing at a rapid clip. And when it comes to those elements for which support is still lacking, you have the aid of an online army of ingenious developers. These community-minded individuals are constantly working at coming up with fallbacks and polyfills to help us push forward and build the next generation of websites and applications.

Get to it!