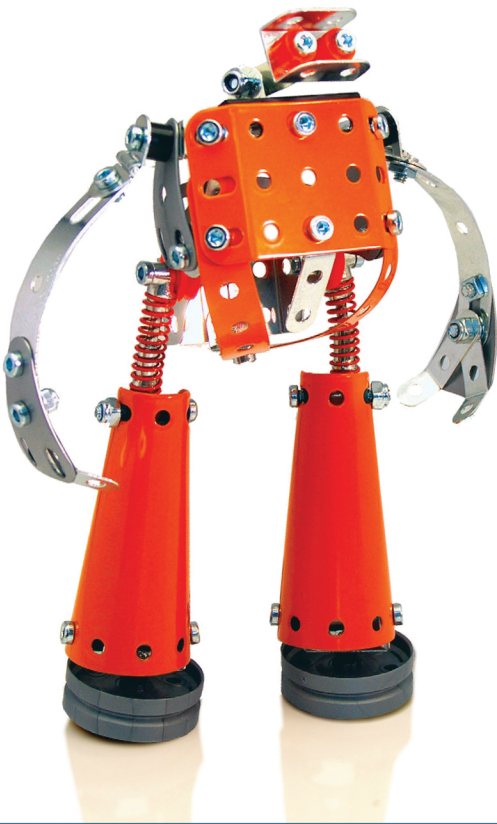


HTML5 & CSS3 FOR THE REAL WORLD

BY ALEXIS GOLDSTEIN
LOUIS LAZARIS
ESTELLE WEYL



POWERFUL HTML5 AND CSS3 TECHNIQUES YOU CAN USE TODAY!

Summary of Contents

Foreword	xxi
Preface	xxiii
1. Introducing HTML5 and CSS3	1
2. Markup, HTML5 Style	11
3. More HTML5 Semantics	35
4. HTML5 Forms	57
5. HTML5 Audio and Video	87
6. Introducing CSS3	119
7. CSS3 Gradients and Multiple Backgrounds	147
8. CSS3 Transforms and Transitions	175
9. Embedded Fonts and Multicolumn Layouts	197
10. Geolocation, Offline Web Apps, and Web Storage	225
11. Canvas, SVG, and Drag and Drop	265
A. Modernizr	313
B. WAI-ARIA	319
C. Microdata	323
Index	329



HTML5 & CSS3 FOR THE REAL WORLD

BY ALEXIS GOLDSTEIN
LOUIS LAZARIS
ESTELLE WEYL

HTML5 & CSS3 for the Real World

by Alexis Goldstein, Louis Lazaris, and Estelle Weyl

Copyright © 2011 SitePoint Pty. Ltd.

Program Director: Lisa Lang

Technical Editor: Louis Simoneau

Expert Reviewer: Russ Weakley

Printing History:

First Edition: May 2011

Indexer: Michele Combs

Editor: Kelly Steele

Cover Design: Alex Walker

Notice of Rights

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the prior written permission of the publisher, except in the case of brief quotations included in critical articles or reviews.

Notice of Liability

The author and publisher have made every effort to ensure the accuracy of the information herein. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors and SitePoint Pty. Ltd., nor its dealers or distributors, will be held liable for any damages caused either directly or indirectly by the instructions contained in this book, or by the software or hardware products described herein.

Trademark Notice

Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.



Published by SitePoint Pty. Ltd.

48 Cambridge Street, Collingwood

VIC 3066 Australia

Web: www.sitepoint.com

Email: business@sitepoint.com

ISBN 978-0-9808469-0-4

Printed and bound in the United States of America

About Alexis Goldstein

Alexis Goldstein first taught herself HTML while a high school student in the mid-1990s, and went on to get her degree in Computer Science from Columbia University. She runs her own software development and training company, aut faciam LLC. Before striking out on her own, Alexis spent seven years in technology on Wall Street, where she worked in both the cash equity and equity derivative spaces at three major firms, and learned to love daily code reviews. She is a teacher and co-organizer of Girl Develop It, a group that conducts low-cost programming classes for women, and a very proud member of the NYC Resistor hacker-space in Brooklyn, NY. You can find Alexis at her website, <http://alexisgo.com/>.

About Louis Lazaris

Louis Lazaris is a freelance web designer and front-end developer based in Toronto, Canada who has been involved in the web design industry since 2000. Louis has been working on websites ever since the days when table layouts and one-pixel GIFs dominated the industry. Over the past five years he has transitioned to embrace web standards while endeavoring to promote best practices that help both developers and their clients reach practical goals for their projects. Louis writes regularly for a number of top web design blogs including his own site, Impressive Webs (<http://www.impressivewebs.com/>).

About Estelle Weyl

Estelle Weyl is a front-end engineer from San Francisco who has been developing standards-based accessible websites since 1999. Estelle began playing with CSS3 when the iPhone was released in 2007, and after four years of web application development for mobile WebKit, she knows (almost) every CSS3 quirk on WebKit, and has vast experience implementing components of HTML5. She writes two popular technical blogs with tutorials and detailed grids of CSS3 and HTML5 browser support (<http://www.standardista.com/>). Estelle's passion is teaching web development, where you'll find her speaking on CSS3, HTML5, JavaScript, and mobile web development at conferences around the USA (and, she hopes, the world).

About the Expert Reviewer

Russ Weakley has worked in the design field for over 18 years, primarily in web design and development, and web training. Russ co-chairs the Web Standards Group and is a founding committee member of the Web Industry Professionals Association of Australia (WIPA). Russ has produced a series of widely acclaimed CSS tutorials, and is internationally recognized for his presentations and workshops. He manages Max Design (<http://maxdesign.com.au/>).

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit <http://www.sitepoint.com/> to access our blogs, books, newsletters, articles, and community forums.

*To my parents, who always
encourage and believe in me.*

*And to my talented, prolific, and
loving Grandma Joan. You always
keep me painting, no matter what
else I may be doing.*

—Alexis

*To Melanie, the best cook in the
world.*

*And to my parents, for funding
the original course that got me
into this unique industry.*

—Louis

*To Amie, for putting up with me,
and to Spazzo and Puppies, for
snuggling with me as I worked
away.*

—Estelle

Table of Contents

Foreword	xxi
-----------------------	-----

Preface	xxiii
----------------------	-------

Who Should Read This Book	xxiii
What's in This Book	xxiv
Where to Find Help	xxvii
The SitePoint Forums	xxvii
The Book's Website	xxvii
The SitePoint Newsletters	xxviii
The SitePoint Podcast	xxviii
Your Feedback	xxviii
Acknowledgments	xxix
Alexis Goldstein	xxix
Louis Lazaris	xxix
Estelle Weyl	xxix
Conventions Used in This Book	xxx
Code Samples	xxx
Tips, Notes, and Warnings	xxxi

Chapter 1 Introducing HTML5 and CSS3	1
--	---

What is HTML5?	1
How did we get here?	3
Would the real HTML5 spec please stand up?	3
Why should I care about HTML5?	5
What is CSS3?	5
Why should I care about CSS3?	6

What do we mean by the “real world”?	7
The Varied Browser Market	8
The Growing Mobile Market	9
On to the Real Stuff	10

Chapter 2 Markup, HTML5 Style

Introducing <i>The HTML5 Herald</i>	11
A Basic HTML5 Template	13
The Doctype	14
The <code>html</code> Element	15
The <code>head</code> Element	15
Leveling the Playing Field	16
The Rest is History	18
HTML5 FAQ	19
Why do these changes still work in older browsers?	19
Shouldn't all tags be closed?	21
What about other XHTML-based syntax customs?	22
Defining the Page's Structure	24
The <code>header</code> Element	24
The <code>section</code> Element	25
The <code>article</code> Element	27
The <code>nav</code> Element	27
The <code>aside</code> Element	29
The <code>footer</code> Element	30
Structuring <i>The HTML5 Herald</i>	31
Wrapping Things Up	34

Chapter 3 More HTML5 Semantics

A New Perspective on Types of Content	35
The Document Outline	37

Breaking News	39
The hgroup Element	40
More New Elements	42
The figure and figcaption Elements	42
The mark Element	43
The progress and meter Elements	44
The time Element	45
Changes to Existing Features	47
The Word "Deprecated" is Deprecated	47
Block Elements Inside Links	47
Bold Text	48
Italicized Text	48
Big and Small Text	49
A cite for Sore Eyes	50
Description (not Definition) Lists	50
Other New Elements and Features	50
The details Element	51
Customized Ordered Lists	52
Scoped Styles	52
The async Attribute for Scripts	52
Validating HTML5 Documents	53
Summary	55

Chapter 4 **HTML5 Forms**

Dependable Tools in Our Toolbox	58
HTML5 Form Attributes	59
The required Attribute	60
The placeholder Attribute	64
The pattern Attribute	67
The disabled Attribute	69

The readonly Attribute	69
The multiple Attribute	69
The form Attribute	70
The autocomplete Attribute	70
The datalist Element and the list Attribute	71
The autofocus Attribute	72
HTML5 New Form Input Types	72
Search	73
Email Addresses	74
URLs	75
Telephone Numbers	76
Numbers	76
Ranges	78
Colors	79
Dates and Times	79
Other New Form Controls in HTML5	83
The output Element	83
The keygen Element	83
Changes to Existing Form Controls and Attributes	84
The form Element	84
The optgroup Element	84
The textarea Element	84
In Conclusion	85

Chapter 5 **HTML5 Audio and Video**

A Bit of History	87
The Current State of Play	88
Video Container Formats	89
Video Codecs	89
Audio Codecs	89

What combinations work in current browsers?	89
The Markup	90
Enabling Native Controls	91
The <code>autoplay</code> Attribute	92
The <code>loop</code> Attribute	93
The <code>preload</code> Attribute	93
The <code>poster</code> Attribute	94
The <code>audio</code> Attribute	94
Adding Support for Multiple Video Formats	95
source Order	96
What about Internet Explorer 6–8?	97
MIME Types	99
Encoding Video Files for Use on the Web	100
Creating Custom Controls	101
Some Markup and Styling to Get Us Started	101
Introducing the Media Elements API	103
Playing and Pausing the Video	105
Muting and Unmuting the Video's Audio Track	108
Responding When the Video Ends Playback	110
Updating the Time as the Video Plays	110
Further Features of the Media Elements API	113
What about audio?	115
Accessible Media	116
It's Showtime	117

Chapter 6 Introducing CSS3	119
Getting Older Browsers on Board	119
CSS3 Selectors	120
Relational Selectors	121
Attribute Selectors	122

Pseudo-classes	124
Structural Pseudo-classes	126
Pseudo-elements and Generated Content	129
CSS3 Colors	131
RGBA	131
HSL and HSLA	132
Opacity	133
Putting It into Practice	134
Rounded Corners: <code>border-radius</code>	136
Drop Shadows	140
Inset and Multiple Shadows	143
Text Shadow	144
More Shadows	145
Up Next	146

Chapter 7 **CSS3 Gradients and Multiple Backgrounds**

Linear Gradients	148
The W3C Syntax	150
The Old WebKit Syntax	154
Putting It All Together	156
Linear Gradients with SVG	158
Linear Gradients with IE Filters	160
Tools of the Trade	161
Radial Gradients	161
The W3C Syntax	162
The Old WebKit Syntax	164
Putting it All Together	166
Repeating Gradients	168
Multiple Background Images	169

Background Size	172
In the Background	174

Chapter 8 **CSS3 Transforms and Transitions**

Transforms	175
Translation	176
Scaling	178
Rotation	180
Skew	181
Changing the Origin of the Transform	182
Support for Internet Explorer 8 and Earlier	182
Transitions	183
transition-property	184
transition-duration	186
transition-timing-function	187
transition-delay	187
The transition Shorthand Property	188
Multiple Transitions	189
Animations	190
Keyframes	191
Animation Properties	192
Moving On	196

Chapter 9 **Embedded Fonts and Multicolumn Layouts**

Web Fonts with @font-face	197
Implementing @font-face	199
Declaring Font Sources	200

Font Property Descriptors	203
Unicode Range	204
Applying the Font	205
Legal Considerations	205
Creating Various Font File Types: Font Squirrel	206
Other Considerations	210
CSS3 Multicolumn Layouts	211
The column-count Property	211
The column-gap Property	212
The column-width Property	213
The columns Shorthand Property	215
Columns and the height Property	215
Other Column Features	216
Other Considerations	218
Progressive Enhancement	219
Media Queries	220
What are Media Queries?	220
Syntax	221
Flexibility of Media Queries	222
Browser Support	222
Further Reading	223
Living in Style	224

Chapter 10 **Geolocation, Offline Web Apps, and Web Storage**

Geolocation	226
Privacy Concerns	227
Geolocation Methods	227
Checking for Support with Modernizr	228
Retrieving the Current Position	229

Geolocation's <code>Position</code> Object	229
Grabbing the Latitude and Longitude	231
Loading a Map	232
A Final Word on Older Mobile Devices	236
Offline Web Applications	237
How It Works: the HTML5 Application Cache	237
Setting Up Your Site to Work Offline	238
Getting Permission to Store the Site Offline	241
Going Offline to Test	241
Making <i>The HTML5 Herald</i> Available Offline	243
Limits to Offline Web Application Storage	245
The Fallback Section	245
Refreshing the Cache	247
Are we online?	248
Further Reading	249
Web Storage	250
Two Kinds of Storage	250
What Web Storage Data Looks Like	252
Getting and Setting Our Data	252
Converting Stored Data	253
The Shortcut Way	253
Removing Items and Clearing Data	254
Storage Limits	254
Security Considerations	255
Adding Web Storage to <i>The HTML5 Herald</i>	256
Viewing Our Web Storage Values with the Web Inspector	260
Additional HTML5 APIs	261
Web Workers	261
Web Sockets	262
Web SQL and IndexedDB	263

Back to the Drawing Board	264
---------------------------------	-----

Chapter 11 Canvas, SVG, and Drag and

Drop 265

Canvas	265
A Bit of Canvas History	266
Creating a canvas Element	266
Drawing on the Canvas	268
Getting the Context	268
Filling Our Brush with Color	269
Drawing a Rectangle to the Canvas	270
The Canvas Coordinate System	271
Variations on <code>fillStyle</code>	271
Drawing Other Shapes by Creating Paths	275
Saving Canvas Drawings	278
Drawing an Image to the Canvas	280
Manipulating Images	282
Converting an Image from Color to Black and White	284
Security Errors with <code>getImageData</code>	286
Manipulating Video with Canvas	287
Displaying Text on the Canvas	290
Accessibility Concerns	294
Further Reading	294
SVG	295
Drawing in SVG	296
Using Inkscape to Create SVG Images	299
SVG Filters	299
Using the Raphaël Library	301
Canvas versus SVG	303
Drag and Drop	304

Feeding the WAI-ARIA Cat	305
Making Elements Draggable	306
The DataTransfer Object	307
Accepting Dropped Elements	308
Further Reading	311
That's All, Folks!	312
Appendix A Modernizr	313
Using Modernizr with CSS	313
Using Modernizr with JavaScript	315
Support for Styling HTML5 Elements in IE8 and Earlier	317
Further Reading	317
Appendix B WAI-ARIA	319
How WAI-ARIA Complements Semantics	319
The Current State of WAI-ARIA	320
Further Reading	321
Appendix C Microdata	323
Aren't HTML5's semantics enough?	324
The Microdata Syntax	324
Understanding Name-Value Pairs	325
Microdata Namespaces	326
Further Reading	327
Index	329

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

Preface

Welcome to *HTML5 & CSS3 for the Real World*. We're glad you've decided to join us on this journey of discovering some of the latest and the greatest in front-end website building technology.

If you've picked up a copy of this book, it's likely that you've dabbled to some degree in HTML and CSS. You might even be a bit of a seasoned pro in certain areas of markup, styling, or scripting, and now want to extend those skills further by dipping into the new features and technologies associated with HTML5 and CSS3.

Learning a new task can be difficult. You may have limited time to invest in poring over the official documentation and specifications for these web-based languages. You also might be turned off by some of the overly technical books that work well as references but provide little in the way of real-world, practical examples.

To that end, our goal with this book is to help you learn through hands-on, practical instruction that will assist you to tackle the real-world problems you face in building websites today—with a specific focus on HTML5 and CSS3.

But this is more than just a step-by-step tutorial. Along the way, we'll provide plenty of theory and technical information to help fill in any gaps in your understanding—the whys and hows of these new technologies—while doing our best not to overwhelm you with the sheer volume of cool new stuff. So let's get started!

Who Should Read This Book

This book is aimed at web designers and front-end developers who want to learn about the latest generation of browser-based technologies. You should already have at least intermediate knowledge of HTML and CSS, as we won't be spending any time covering the basics of markup and styles. Instead, we'll focus on teaching you what new powers are available to you in the form of HTML5 and CSS3.

The final two chapters of this book cover some of the new JavaScript APIs that have come to be associated with HTML5. These chapters, of course, require some basic familiarity with JavaScript—but they're not critical to the rest of the book. If you're

unfamiliar with JavaScript, there's no harm in skipping over them for now, returning later when you're better acquainted with it.

What's in This Book

This book comprises eleven chapters and three appendices. Most chapters follow on from each other, so you'll probably get the most benefit reading them in sequence, but you can certainly skip around if you only need a refresher on a particular topic.

Chapter 1: *Introducing HTML5 and CSS3*

Before we tackle the hands-on stuff, we'll present you with a little bit of history, along with some compelling reasons to start using HTML5 and CSS3 today.

We'll also look at the current state of affairs in terms of browser support, and argue that a great deal of these new technologies are ready to be used today—so long as they're used wisely.

Chapter 2: *Markup, HTML5 Style*

In this chapter, we'll show you some of the new structural and semantic elements that are new in HTML5. We'll also be introducing *The HTML5 Herald*, a demo site we'll be working on throughout the rest of the book. Think `div`s are boring? So do we. Good thing HTML5 now provides an assortment of options: `article`, `section`, `nav`, `footer`, `aside`, and `header`!

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 7: *CSS3 Gradients and Multiple Backgrounds*

When was the last time you worked on a site that didn't have a gradient or a background image on it? CSS3 provides some overdue support to developers spending far too much time wrangling with Photoshop, trying to create the perfect background gradients and images without breaking the bandwidth bank. Now you can specify linear or radial gradients right in your CSS without images, and you can give an element any number of background images. Time to ditch all those spare divs you've been lugging around.

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 animations still lack widespread support, we think you'll agree they're pretty sweet.

Chapter 9: *Embedded Fonts and Multicolumn Layouts*

Do you prefer Arial or Verdana? Georgia or Times? How about none of those? In this chapter, we'll look at how we can move past the “web-safe” fonts of yesteryear and embed any fonts right into our pages for visitors to download

along with our stylesheets and images. We'll also look at a promising new CSS feature that allows us to lay out content across multiple columns without using extra markup or the dreaded `float`.

Chapter 10: *Geolocation, Offline Web Apps, and Web Storage*

The latest generation of browsers come equipped with a wide selection of new standard JavaScript APIs. Many of these are specifically geared towards mobile browsers, but still carry benefits for desktop users. In this chapter, we'll look at three of the most exciting: Geolocation, Offline Web Apps, and Web Storage. We'll also touch briefly on some of the APIs that we won't be covering in detail—either because they're poorly supported, or have limited use cases—and give you some pointers should you want to investigate further.

Chapter 11: *Canvas, SVG, and Drag and Drop*

We devote the book's final chapter to, first of all, covering two somewhat competing technologies for drawing and displaying graphics. Canvas is new to HTML5, and provides a pixel surface and a JavaScript API for drawing shapes to it. SVG, on the other hand, has been around for years, but is now achieving very good levels of browser support, so it's an increasingly viable alternative. Finally, we'll cover one more new JavaScript API—Drag and Drop—which provides native handling of drag-and-drop interfaces.

Appendix A: *Modernizr*

A key tool in any HTML5 superhero's utility belt, Modernizr is a nifty little JavaScript library that detects support for just about every HTML5 and CSS3 feature, allowing you to selectively style your site or apply fallback strategies. We've included a quick primer on how to use Modernizr in this appendix, even though Modernizr is used throughout the book. This way, you have a ready reference available in one place, while the other chapters focus on the meat of HTML5 and CSS3.

Appendix B: *WAI-ARIA*

A separate specification that's often mentioned in the same breath as HTML5, WAI-ARIA is the latest set of tools to help make sophisticated web applications accessible to users of assistive technology. While a whole book could be devoted to WAI-ARIA, we thought it beneficial to include a quick summary of what it is, as well as some pointers to places where you can learn more.

Appendix C: *Microdata*

Microdata is part of the HTML5 specification that deals with annotating markup with machine-readable labels. It's still somewhat in flux, but we thought it was worthwhile to get you up to speed with a few examples.

Where to Find Help

SitePoint has a thriving community of web designers and developers ready and waiting to help you out if you run into trouble. We also maintain a list of known errata for the book, which you can consult for the latest updates.

The SitePoint Forums

The SitePoint Forums¹ are discussion forums where you can ask questions about anything related to web development. You may, of course, answer questions too. That's how a forum site works—some people ask, some people answer, and most people do a bit of both. Sharing your knowledge benefits others and strengthens the community. A lot of interesting and experienced web designers and developers hang out there. It's a good way to learn new stuff, have questions answered in a hurry, and generally have a blast.

The Book's Website

Located at <http://sitepoint.com/books/rw1/>, the website that supports this book will give you access to the following facilities:

The Code Archive

As you progress through this book, you'll note a number of references to the code archive. This is a downloadable ZIP archive that contains every line of example source code printed in this book. If you want to cheat (or save yourself from carpal tunnel syndrome), go ahead and download the archive.²

Updates and Errata

No book is perfect, and we expect that watchful readers will be able to spot at least one or two mistakes before the end of this one. The Errata page³ on the book's

¹ <http://www.sitepoint.com/forums/>

² <http://www.sitepoint.com/books/rw1/code.php>

³ <http://www.sitepoint.com/books/rw1/errata.php>

website will always have the latest information about known typographical and code errors.

The SitePoint Newsletters

In addition to books like this one, SitePoint publishes free email newsletters, such as the *SitePoint Tech Times*, *SitePoint Tribune*, and *SitePoint Design View*, to name a few. In them, you'll read about the latest news, product releases, trends, tips, and techniques for all aspects of web development. Sign up to one or more SitePoint newsletters at <http://www.sitepoint.com/newsletter/>.

The SitePoint Podcast

Join the SitePoint Podcast team for news, interviews, opinion, and fresh thinking for web developers and designers. We discuss the latest web industry topics, present guest speakers, and interview some of the best minds in the industry. You can catch up on the latest and previous podcasts at <http://www.sitepoint.com/podcast/>, or subscribe via iTunes.

Your Feedback

If you're unable to find an answer through the forums, or if you wish to contact us for any other reason, the best place to write is books@sitepoint.com. We have a well-staffed email support system set up to track your inquiries, and if our support team members can't answer your question, they'll send it straight to us. Suggestions for improvements, as well as notices of any mistakes you may find, are especially welcome.

Acknowledgments

Alexis Goldstein

Thank you to Lisa Lang, Russ Weakley, and Louis Simoneau. Your attention to detail, responsiveness, and impressive technical expertise made this book an absolute joy to work on. Thank you to my co-authors, Louis and Estelle, who never failed to impress me with their deep knowledge, vast experience, and uncanny ability to find bugs in the latest browsers. A special thank you to Estelle for the encouragement, for which I am deeply grateful. Finally, thank you to my girlfriend Tabatha, who now knows more about HTML5's JavaScript APIs than most of my nerdy friends. Thank you for your patience, your feedback, and all your support. You help me take things less seriously, which, as anyone who knows me knows, is a monumental task. Thank you for always making me laugh.

Louis Lazaris

Thank you to my wife for putting up with my odd work hours while I took part in this great project. Thanks to my talented co-authors, Estelle and Alexis, for gracing me with the privilege of having my name alongside theirs, and, of course, to our expert reviewer Russ for his great technical insight during the writing process. And special thanks to the talented staff at SitePoint for their super-professional handling of this project and everything that goes along with such an endeavor.

Estelle Weyl

Thank you to the entire open source community. With the option to “view source,” I have learned from every developer who opted for markup rather than plugins. I would especially like to thank Jen Mei Wu and Sandi Watkins, who helped point me in the right direction when I began my career. Thank you to Dave Gregory and Laurie Voss who have always been there to help me find the words when they escaped me. Thank you to Stephanie Sullivan for brainstorming over code into the wee hours of the morning. And thank you to my developer friends at Opera, Mozilla, and Google for creating awesome browsers, providing us with the opportunity to not just play with HTML5 and CSS, but also to write this book.

Conventions Used in This Book

You'll notice that we've used certain typographic and layout styles throughout the book to signify different types of information. Look out for the following items:

Code Samples

Code in this book will be displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park. The birds
were singing and the kids were all back at school.</p>
```

If the code is to be found in the book's code archive, the name of the file will appear at the top of the program listing, like this:

example.css

```
.footer {
  background-color: #CCC;
  border-top: 1px solid #333;
}
```

If only part of the file is displayed, this is indicated by the word *excerpt*:

example.css (*excerpt*)

```
border-top: 1px solid #333;
```

If additional code is to be inserted into an existing example, the new code will be displayed in bold:

```
function animate() {
  new_variable = "Hello";
}
```

Where existing code is required for context, rather than repeat all the code, a vertical ellipsis will be displayed:

```
function animate() {
  :
  return new_variable;
}
```

Some lines of code are intended to be entered on one line, but we've had to wrap them because of page constraints. A ➞ indicates a line break that exists for formatting purposes only, and should be ignored:

```
URL.open("http://www.sitepoint.com/blogs/2007/05/28/user-style-she
➞ets-come-of-age/");
```

Tips, Notes, and Warnings



Hey, You!

Tips will give you helpful little pointers.



Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.



Make Sure You Always ...

... pay attention to these important points.



Watch Out!

Warnings will highlight any gotchas that are likely to trip you up along the way.

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 page¹.

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 `articles`, `sections`, `headers`, `footers`, 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.



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 study⁶ conducted on the Yahoo network, published in October 2010, users with JavaScript disabled amounted to around 1% of total traffic worldwide. Another study⁷ 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 `divs` 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/ymn/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 `sections` 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 post¹² 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 footers 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 reusable.” 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.

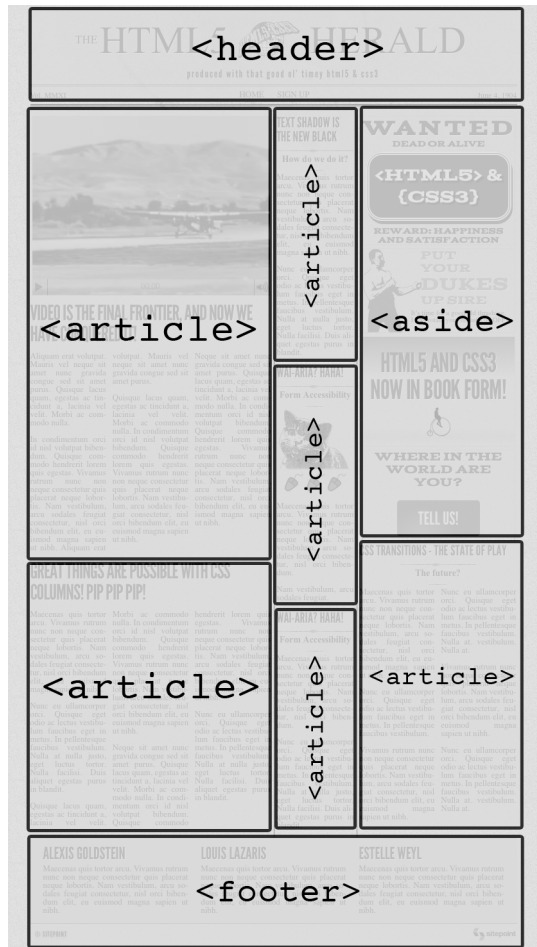


Figure 2.2. *The HTML5 Herald*, broken into structural HTML5 elements

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

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 3

More HTML5 Semantics

Our sample site is coming along nicely. We've given it some basic structure, along the way learning more about marking up content using HTML5's new elements.

In this chapter, we'll discuss even more new elements, along with some changes and improvements to familiar elements. We'll also add some headings and basic text to our project, and we'll discuss the potential impact of HTML5 on SEO and accessibility.

Before we dive into that, though, let's take a step back and examine a few new—and a little tricky—concepts that HTML5 brings to the table.

A New Perspective on Types of Content

For layout and styling purposes, developers have become accustomed to thinking of elements in an HTML page as belonging to one of two categories: block and inline. Although elements are still rendered as either block or inline by browsers, the HTML5 spec takes the categorization of content a step further. The specification now defines a set of more granular **content models**. These are broad definitions about the kind of content that should be found inside a given element. Most of the

time they'll have little impact on the way you write your markup, but it's worth having a passing familiarity with them, so let's have a quick look.

Metadata content

This category is what it sounds like: data that's not present on the page itself, but affects the page's presentation or includes other information *about* the page. This includes elements like `title`, `link`, `meta`, and `style`.

Flow content

Flow content includes just about every element that's used in the body of an HTML document, including elements like `header`, `footer`, and even `p`. The only elements *excluded* are those that have no effect on the document's flow: `script`, `link`, and `meta` elements in the page's head, for example.

Sectioning content

This is the most interesting—and for our purposes, most relevant—type of content in HTML5. In the last chapter, we often found ourselves using the generic term “section” to refer to a block of content that could contain a heading, footer, or aside. In fact, what we were actually referring to was **sectioning content**. In HTML5, this includes `article`, `aside`, `nav`, and `section`. We'll talk about sectioning content and how it can affect the way you write your markup in more detail very shortly.

Heading content

This type of content defines the header of a given section, and includes the various levels of heading (`h1`, `h2`, and so on), as well as the new `hgroup` element, which we'll cover a bit later.

Phrasing content

This category is roughly the equivalent to what you're used to thinking of as *inline* content, it includes elements like `em`, `strong`, `cite`, and the like.

Embedded content

This one's fairly straightforward, and includes elements that are, well, *embedded* into a page, such as `img`, `object`, `embed`, `video`, `canvas`, and others.

Interactive content

This category includes any content with which users can interact. It consists mainly of form elements, as well as links and other elements that are interactive only when certain attributes are present.

As you might gather from reading the list above, some elements can belong to more than one category. There are also some elements that fail to fit into *any* category. Don't worry if this seems confusing; just remember that these distinctions exist—that should be more than enough.

The Document Outline

In previous versions of HTML, you could draw up an outline of any given document by looking at the various levels of headings (h1 through to h6) contained in the page. Each time a new level of heading was added, you'd go one step deeper into the hierarchy of your outline. For example, take this markup:

```
<h1>Title</h1>
:
<h2>Subtitle</h2>
:
<h3>Another level</h3>
:
<h2>Another subtitle</h2>
```

This would produce the document outline shown in Figure 3.1.

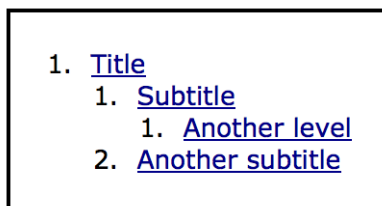


Figure 3.1. A simple document outline

It was preferred that each page have a single h1 element, with other headings following sequentially.

In order to make content easier to syndicate and more portable, the HTML5 specification provides a clear algorithm for constructing the outline of an HTML document. Each element that falls under the category of “sectioning content” creates a new node in the document outline. Heading (h1–h6) elements within a block of sectioning content also create “implied” sections—this is actually what was happening in our simple outline above.

This all sounds more complicated than it is. To start to gain an understanding of it, let’s look at how the above example could be rewritten using some additional HTML5 elements:

```
<section>
  <h1>Title</h1>
  :
  <article>
    <h1>Article Title</h1>
    :
    <h2>Article Subtitle</h2>
    :
  </article>
  <article>
    <h1>Another subtitle</h1>
    :
  </article>
</section>
```

This results in exactly the same document outline as above: each piece of sectioning content (the `article` elements in this example) creates a new branch in the document tree, and so can have its own h1. This way, each section has its own mini document outline.

The advantage of the new outlining algorithm is that it allows us to move an entire section to a completely different document while preserving the same markup. Beforehand, a post’s title on that post’s page might have been an h1, but the same post’s title on the home page or a category page listing might have been an h2 or h3. Now, you can just keep the same markup, as long as the headings are grouped together in a sectioning content element.



Testing Document Outlines

Getting a document's outline right in HTML5 can be tricky at first. If you're having trouble, you can use a handy JavaScript bookmarklet called `h5o`¹ to show the outline of any document you're viewing with the HTML5 outline algorithm. The resulting display will reveal your document's hierarchy in accordance with the HTML5 standard, so you can make corrections as needed.

To install it in your browser, download the HTML file from the site and open it in your browser; then drag the link to your favorites or bookmarks bar. Now you can use the `h5o` link to display a document outline for any page you're viewing.

It's important to note that the old way of coding and structuring content, with a single `h1` on each page, is still valid HTML5. Your pages will still be valid, even though you'll miss out on the portability and syndication benefits.



Understanding Sectioning Roots

Distinct from—but similar to—sectioning content, HTML5 also defines a type of element called a **sectioning root**. These include `blockquote`, `body`, `details`, `fieldset`, `figure`, and `td`. What makes the sectioning root elements distinct is that, although they may individually have their own outlines, the sectioning content and headings inside these elements do *not* contribute to the overall document outline (with the exception of `body`, the outline of which *is* the document's outline).

Breaking News

Now that we've got a solid handle on HTML5's content types and document outlines, it's time to dive back into *The HTML5 Herald* and add some headings for our articles.

For brevity, we'll deal with each section individually. Let's add a title and subtitle to our header, just above the navigation:

```
<header>

  <hgroup>
    <h1>The HTML5 Herald</h1>
```

¹ <http://code.google.com/p/h5o/>

```

    <h2>Produced With That Good Ol' Timey HTML5 & CSS3</h2>
  </hgroup>
  <nav>
  :
  </nav>

</header>

```

The hgroup Element

You'll notice we have introduced three elements into our markup: the title of the website, which is marked up with the customary h1 element; a tagline immediately below the primary page title, marked up with an h2; and a new HTML5 element that wraps our title and tagline, the hgroup element.

To understand the purpose of the hgroup element, consider again how a page's outline is built. Let's take our heading markup without the hgroup element:

```

<h1>The HTML5 Herald</h1>
<h2>Produced With That Good Ol' Timey HTML5 & CSS3</h2>

```

This would produce the document outline shown in Figure 3.2.

1. [The HTML5 Herald](#)
 1. [Produced With That Good Ol' Timey HTML5 & CSS3](#)

Figure 3.2. A subtitle generates an unwanted node in the document outline

The h2 element creates a new, implicit section: all content that follows is logically grouped under a subsection created by that tagline—and that's not what we want at all. Furthermore, if we have additional headings (for example, for article titles) that use h2, those new headings will be hierarchically on the same level as our tagline; this is also incorrect, as shown in Figure 3.3.

1. [The HTML5 Herald](#)
 1. [Produced With That Good Ol' Timey HTML5 & CSS3](#)
 2. [An Article Title](#)

Figure 3.3. Other headlines in the content wrongly appear grouped with the tagline

Well, we could mark up subsequent headings starting with h3, right? But again, this causes problems in our document's outline. Now, the headings beginning with h3 will become subsidiary to our tagline, as Figure 3.4 shows.

1. [The HTML5 Herald](#)
 1. [Produced With That Good Ol' Timey HTML5 & CSS3](#)
 1. [An Article Title](#)

Figure 3.4. Using further nested heading levels fails to solve the problem

That's also undesirable; we want the new headings to be subsections of our primary heading, the h1 element.

What if, instead, we opted to mark up our tagline using a generic element like a p or span:

```
<h1>HTML5 Herald</h1>
<p id="tagline">Produced With That Good Ol' Timey HTML5 & CSS3
➡</p>
```

While this does avoid cluttering up the document outline with a superfluous branch, it's a little lacking in terms of semantics. You might be thinking that the id attribute helps define the element's meaning by using a value of tagline. But the id attribute cannot be used by the browser to infer meaning for the element in question—it adds nothing to the document's semantics.

This is where the hgroup element comes in. The hgroup element tells the user agent that the headings nested within it form a composite heading (a heading group, as it were), with the h1 being the primary parent element. This prevents our document

outline from becoming jumbled, and it helps us avoid using nonsemantic elements in our page.

So any time you want to include a subheading without affecting the document's outline, just wrap the headings in an `hgroup` element; this will resolve the problem without resorting to undesirable methods. Figure 3.5 shows the outline produced for the header, with the `hgroup` wrapping the two headings.



Figure 3.5. `hgroup` to the rescue

Much better!

More New Elements

In addition to the structural elements we saw in Chapter 2 and the `hgroup` element we've just covered, HTML5 introduces a number of new semantic elements. Let's examine some of the more useful ones.

The `figure` and `figcaption` Elements

The `figure` and `figcaption` elements are another pair of new HTML5 elements that contribute to the improved semantics in HTML5. The `figure` element is explained in the spec as follows:

The element can [...] be used to annotate illustrations, diagrams, photos, code listings, etc, that are referred to from the main content of the document, but that could, without affecting the flow of the document, be moved away from that primary content, e.g. to the side of the page, to dedicated pages, or to an appendix.

Think of charts, graphs, images to accompany text, or example code. All those types of content might be good places to use `figure` and potentially `figcaption`.

The `figcaption` element is simply a way to mark up a caption for a piece of content that appears inside of a `figure`.

In order to use the `figure` element, the content being placed inside it must have some relation to the main content in which the `figure` appears. If you can completely remove it from a document, and the document's content can still be fully understood, you probably shouldn't be using `figure`; you might, however, need to use `aside` or another alternative. Likewise, if the image or listing forms part of the flow of the document, and the text would need rewording if you moved it, it's probably best to use another option.

Let's look at how we'd mark up a `figure` inside an article:

```
<article>
  <hgroup>
    <h1>WAI-ARIA</h1>
    <h2>Web App Accessibility</h2>
  </hgroup>

  <p>Lorem ipsum dolor ... </p>

  <p>As you can see in <a href="#fig1">Figure 1</a>,

  <figure id="fig1">
    <figcaption>Screen Reader Support for WAI-ARIA</figcaption>
    
  </figure>

  <p>Lorem ipsum dolor ... </p>
</article>
```

The mark Element

The `mark` element “indicates a part of the document that has been highlighted due to its likely relevance to the user's current activity.” Admittedly, there are very few uses we can think of for the `mark` element. The most common is in the context of a search, where the keywords that were searched for are highlighted in the results.

Avoid confusing `mark` with `em` or `strong`; those elements add contextual importance, whereas `mark` separates the targeted content based on a user's current browsing or search activity.

For example, if a user has arrived at an article on your site from a Google search for the word “HTML5,” you might highlight words in the article using the `mark` element, like this:

```
<h1>Yes, You Can Use <mark>HTML5</mark> Today!</h1>
```

The `mark` element can be added to the document either using server-side code, or JavaScript once the page has loaded.

The progress and meter Elements

Two new elements added in HTML5 allow for marking up of data that’s being measured or gauged in some way. The difference between them is fairly subtle: `progress` is used to describe the current status of a changing process that’s headed for completion, regardless of whether the completion state is defined. The traditional download progress bar is a perfect example of `progress`.

The `meter` element, meanwhile, represents an element whose range is known, meaning it has definite minimum and maximum values. The spec gives the examples of disk usage, or a fraction of a voting population—both of which have a definite maximum value. Therefore, it’s likely you wouldn’t use `meter` to indicate an age, height, or weight—all of which normally have unknown maximum values.

Let’s first look at `progress`. The `progress` element can have a `max` attribute to indicate the point at which the task will be complete, and a `value` attribute to indicate the task’s status. Both of these attributes are optional. Here’s an example:

```
<h1>Your Task is in Progress</h1>
<p>Status: <progress min="0" max="100" value="0"><span>0</span>%
➡</progress></p>
```

This element would best be used (along with some JavaScript) to dynamically change the value of the percentage as the task progresses. You’ll notice that the code includes `` tags, isolating the number value; this facilitates targeting the number directly from your script when you need to update it.

The `meter` element has six associated attributes. In addition to `max` and `value`, it also allows use of `min`, `high`, `low`, and `optimum`.

The `min` and `max` attributes reference the lower and upper boundaries of the range, while `value` indicates the current specified measurement. The `high` and `low` attributes indicate thresholds for what is considered “high” or “low” in the context. For example, your grade on a test can range from 0% to 100% (`max`), but anything below 60% is considered low and anything above 85% is considered high. `optimum` refers to the ideal value. In the case of a test score, the value of `optimum` would be 100.

Here’s an example of `meter`, using the premise of disk usage:

```
<p>Total current disk usage: <meter value="63" min="0" max="320"
↳low="10" high="300" title="gigabytes">63 GB</meter>
```

The time Element

Dates and times are invaluable components of web pages. Search engines are able to filter results based on time, and in some cases, a specific search result can receive more or less weight by a search algorithm depending on when it was first published.

The `time` element has been specifically designed to deal with the problem of humans reading dates and times differently from machines. Take the following example:

```
<p>We'll be getting together for our next developer conference on
↳12 October of this year.</p>
```

While humans reading this paragraph will understand when the event will take place, it would be less clear to a machine attempting to parse the information.

Here’s the same paragraph with the `time` element introduced:

```
<p>We'll be getting together for our next developer conference on
↳<time datetime="2011-10-12">12 October of this year</time>.</p>
```

The `time` element also allows you to express dates and times in whichever format you like while retaining an unambiguous representation of the date and time behind the scenes, in the `datetime` attribute. This value could then be converted into a localized or preferred form using JavaScript, or by the browser itself, though currently no browsers implement any special handling of the `time` element.

If you want to include a time along with the date, you would do it like this:

```
<time datetime="2011-10-12T16:24:34.014Z">12 October of this year.
➡</time>
```

In the above example, the T character is used to indicate the start of the time. The format is HH:MM:SS with milliseconds after the decimal point. The Z character is optional and indicates that the time zone is Coordinated Universal Time (UTC). To indicate a time zone offset (instead of UTC), you would append it with a plus or minus, like this:

```
<time datetime="2011-10-12T16:24:34.014-04:00">12 October of
this year</time>
```

In addition to the `datetime` attribute shown in the above examples, the `time` element allows use of the `pubdate` attribute. This is a Boolean attribute, and its existence indicates that the content within the closest ancestor `article` element was published on the specified date. If there's no `article` element, the `pubdate` attribute would apply to the entire document.

For example, in the header of *The HTML5 Herald*, the issue's publication date is a perfect candidate for the `time` element with a `pubdate` attribute:

[index.html \(excerpt\)](#)

```
<p id="issue"><time datetime="1904-06-04" pubdate>June 4, 1904
➡</time></p>
```

Because this element indicates the publication date of our newspaper, we've added the `pubdate` attribute. Any other dates referred to on the page—in the text of articles, for example—would omit this attribute.

The `time` element has some associated rules and guidelines:

- You should not use `time` to encode unspecified dates or times (for example, “during the ice age” or “last winter”).
- The date represented cannot be “BC” or “BCE” (before the common era); it must be a date on the Gregorian Calendar.
- The `datetime` attribute has to be a valid date string.

- If the `time` element lacks a `datetime` attribute, the element's text content (whatever appears between the opening and closing tags) needs to be a valid date string.

The uses for the `time` element are endless: calendar events, publication dates (for blog posts, videos, press releases, and so forth), historic dates, transaction records, article or content updates, and much more.

Changes to Existing Features

While new elements and APIs have been the primary focus of HTML5, this latest iteration of web markup has also brought with it changes to existing elements. For the most part, any changes that have been made have been done with backwards compatibility in mind, to ensure that the markup of existing content is still usable.

We've already considered some of the changes (the doctype declaration, character encoding, content types, and the document outline, for example). Let's look at other significant changes introduced in the HTML5 spec.

The Word “Deprecated” is Deprecated

In previous versions of HTML and XHTML, elements that were no longer recommended for use (and so removed from the spec), were considered “deprecated.” In HTML5, there is no longer any such thing as a deprecated element; the term now used is “obsolete.”

This may seem like an insignificant change, but the difference is important: while a deprecated element would be removed from the specification, an obsolete element will remain there. This is so that browser makers still have a standard way of rendering these elements consistently, even if their use is no longer recommended. For example, you can view information in the W3C's specification on frames (an obsolete feature) at <http://dev.w3.org/html5/spec/Overview.html#frames>.

Block Elements Inside Links

Although most browsers handled this situation just fine in the past, it was never actually valid to place a block-level element inside an `a` element. Instead, to produce valid HTML, you'd have to use multiple `a` elements and style the group to appear as a single block.

In HTML5, you're now permitted to wrap almost anything—other than form elements or other links—in an a element without having to worry about validation errors.

Bold Text

A few changes have been made in the way that bold text is semantically defined in HTML5. There are essentially two ways to make text bold in most browsers: using the `b` element, or using the `strong` element.

Although the `b` element was never deprecated, before HTML5 it was discouraged in favor of `strong`. The `b` element previously was a way of saying “make this text appear in boldface.” Since HTML markup is supposed to be all about the meaning of the content, leaving the presentation to CSS, this was unsatisfactory.

In HTML5, the `b` element has been redefined to represent a section of text that is “stylistically offset from the normal prose without conveying any extra importance.”

The `strong` element, meanwhile, still conveys more or less the same meaning. In HTML5, it represents “strong importance for its contents.” Interestingly, the HTML5 spec allows for nesting of `strong` elements. So, if an entire sentence consisted of an important warning, but certain words were of even greater importance, the sentence could be wrapped in one `strong` element, and each important word could be wrapped in its own nested `strong`.

Italicized Text

Along with the modifications to the `b` and `strong` elements, changes have been made in the way the `i` element is defined in HTML5.

Previously, the `i` element was used to simply render italicized text. As with `b`, this definition was unsatisfactory. In HTML5, the definition has been updated to “a span of text in an alternate voice or mood, or otherwise offset from the normal prose.” So the appearance of the text has nothing to do with the semantic meaning, although it may very well still be italic—that's up to you.

An example of content that can be offset using `i` tags might be an idiomatic phrase from another language, such as *reductio ad absurdum*, a latin phrase meaning “reduction to the point of absurdity.” Other examples could be text representing a dream sequence in a piece of fiction, or the scientific name of a species in a journal article.

The `em` element is unchanged, but its definition has been expanded to clarify its use. It still refers to text that’s emphasized, as would be the case colloquially. For example, the following two phrases have the exact same wording, but their meanings change because of the different use of `em`:

```
<p>Harry’s Grill is the best <em>burger</em> joint in town.</p>  
<p>Harry’s Grill <em>is</em> the best burger joint in town.</p>
```

In the first sentence, because the word “burger” is emphasized, the meaning of the sentence focuses on the type of “joint” being discussed. In the second sentence, the emphasis is on the word “is,” thus moving the sentence focus to the question of whether Harry’s Grill really is the best of all burger joints in town.

Neither `i` nor `em` should be used to mark up a publication title; instead, use `cite` (see the section called “A `cite` for Sore Eyes”).

Of all the four elements discussed here (`b`, `i`, `em`, and `strong`), the only one that gives contextual importance to its content is the `strong` element.

Big and Small Text

The `big` element was previously used to represent text displayed in a large font. The `big` element is now obsolete and should not be used. The `small` element, however, is still valid, but has a different meaning.

Previously, `small` was intended to describe “text in a small font.” In HTML5, it represents “side comments such as small print.” Some examples where `small` might be used include information in footer text, fine print, and terms and conditions. The `small` element should only be used for short runs of text.

Although the presentational implications of `small` have been removed from the definition, text inside `small` tags will more than likely still appear in a smaller font than the rest of the document.

For example, the footer of *The HTML5 Herald* includes a copyright notice. Since this is essentially legal fine print, it’s perfect for the `small` element:

```
<small>&copy; SitePoint Pty. Ltd.</small>
```

A cite for Sore Eyes

The `cite` element is another one that's been redefined in HTML5, accompanied by a fair bit of controversy. In HTML4, the `cite` element represented “a citation or a reference to other sources.” Within the scope of that definition, the spec permitted a person's name to be marked up with `cite` (in the case of a quotation attributed to an individual, for example).

HTML5 expressly forbids the use of `cite` for a person's name, seemingly going against the principle of backwards compatibility. Now the spec describes `cite` as “the title of a work,” and gives a whole slew of examples, including a book, a song, a TV show, and a theatre production.

Some notable web standards advocates (including Jeremy Keith and Bruce Lawson) have opposed this new definition forbidding people's names within `cite`. For more information on the ongoing debate, see the page on this topic on the WHATWG Wiki.²

Description (not Definition) Lists

The existing `d1` (definition list) element, along with its associated `dt` (term) and `dd` (description) children, has been redefined in the HTML5 spec. Previously, in addition to terms and definitions, the spec allowed the `d1` element to mark up dialogue, but the spec now prohibits this.

In HTML5, these lists are no longer called “definition lists”; they're now the more generic-sounding “description lists.” They should be used to mark up any kind of name-value pairs, including terms and definitions, metadata topics and values, and questions and answers.

Other New Elements and Features

We've introduced you to and expounded upon some of the more practical new elements and features. Now, in this section, we'll touch on lesser-known elements, attributes, and features that have been added to the HTML5 spec.

² http://wiki.whatwg.org/wiki/Cite_element

The details Element

This new element helps mark up a section of the document that’s hidden, but can be expanded to reveal additional information. The aim of the element is to provide native support for a feature common on the Web—a collapsible box that has a title, and more info or functionality hidden away.

Normally this kind of widget is created using a combination of markup and scripting. The inclusion of it in HTML5 intends to remove the scripting requirements and simplify its implementation for web authors.

Here’s how it might look:

```
<details>
  <summary>Some Magazines of Note</summary>
  <ul>
    <li><cite>Bird Watchers Digest</cite></li>
    <li><cite>Rowers Weekly</cite></li>
    <li><cite>Fishing Monthly</cite></li>
  </ul>
</details>
```

The example above would cause the contents of the `summary` element to appear to the user, with the rest of the content hidden. Upon clicking `summary`, the hidden content appears.

If `details` lacks a defined `summary`, the user agent will define a default summary (for example, “Details”). If you want the hidden content to be visible by default, you can use the Boolean `open` attribute.

The `summary` element can only be used as a child of `details`, and it must be the first child, if used.

So far, `details` has little to no support in browsers. A couple of JavaScript-based polyfills are available, including one by Mathias Bynens.³

³ <http://mathiasbynens.be/notes/html5-details-jquery>

Customized Ordered Lists

Ordered lists, using the `ol` element, are quite common in web pages. HTML5 introduces a new Boolean attribute called `reversed` that, when present, reverses the order of the list items.

While we're on the topic of ordered lists, HTML5 has brought back the `start` attribute, deprecated in HTML4. The `start` attribute lets you specify with which number your list should begin.

Support is good for `start`, but `reversed` has yet to be implemented in most browsers.

Scoped Styles

The `style` element, used for embedding styles directly in your pages, now allows use of a Boolean attribute called `scoped`. Take the following code example:

```
<h1>Page Title</h1>
<article>
  <style scoped>
    h1 { color: blue; }
  </style>
  <h1>Article Title</h1>
  <p>Article content.</p>
</article>
```

Because the `scoped` attribute is present, the styles declared inside the `style` element will only apply to the parent element and its children (if cascading rules permit), instead of the entire document. This allows specific sections inside documents (like the `article` in the above example) to be easily portable along with their associated styles.

This is certainly a handy new feature, but as of this writing, no browser supports the `scoped` attribute. As a temporary solution, a jQuery-based polyfill is available at <https://github.com/thingsinjars/jQuery-Scoped-CSS-plugin>.

The `async` Attribute for Scripts

The `script` element now allows the use of the `async` attribute, which is similar to the existing `defer` attribute. Using `defer` specifies that the browser should wait until the page's markup is parsed before loading the script. The new `async` attribute

allows you to specify that a script should load asynchronously (meaning it should load as soon as it's available), without causing other elements on the page to delay while it loads. Both `defer` and `async` are Boolean attributes.

These attributes must only be used when the `script` element defines an external file. For legacy browsers, you can include both `async` and `defer` to ensure that one or the other is used, if necessary. In practice, both attributes will have the effect of not pausing the browser's rendering of the page while scripts are downloaded; however, `async` can often be more advantageous, as it will load the script “in the background” while other rendering tasks are taking place, and execute the script as soon as it's available.

The `async` attribute is particularly useful if the script you're loading has no other dependencies, and it would benefit the user experience if the script is loaded as soon as possible, rather than after the page loads.

Validating HTML5 Documents

In chapter two, we introduced you to a number of syntax changes in HTML5, and touched on some issues related to validation. Let's expand upon those concepts a little more so that you can better understand how validating pages has changed.

The HTML5 validator is no longer concerned with code style. You can use uppercase, lowercase, omit quotes from attributes, leave tags open, and be as inconsistent as you like, and your page will often still be valid.

So, you ask, what *does* count as an error for the HTML5 validator? It will alert you to incorrect use of elements, elements included where they shouldn't be, missing required attributes, incorrect attribute values, and the like. In short, the validator will let you know if your markup conflicts with the specification, so it's still an extremely valuable tool when developing your pages.

However, since many of us are accustomed to the stricter validation rules imposed on XHTML documents, let's go through some specifics. This way, you can understand what is considered valid in HTML5 that was invalid when checking XHTML-based pages:

- Some elements that were required in XHTML-based syntax are no longer required for a document to pass HTML5 validation; examples include the `html` and `body` elements.
- Void elements, or elements that stand alone and don't contain any content, are not required to be closed using a closing slash; examples include `<meta>` and `
`.
- Elements and attributes can be in uppercase, lowercase, or mixed case.
- Quotes are unnecessary around attribute values, unless multiple space-delimited values are used, or a URL appears as a value and contains a query string with an equals (=) character in it.
- Some attributes that were required in XHTML-based syntax are no longer required in HTML5; examples include the `type` attribute for the `script` element, and the `xmlns` attribute for the `html` element.
- Some elements that were deprecated and thus invalid in XHTML are now valid; one example is the `embed` element.
- Stray text that doesn't appear inside any element would invalidate an XHTML document; this is not the case in HTML5.
- Some elements that needed to be closed in XHTML can be left open without causing validation errors in HTML5; examples include `p`, `li`, and `dt`.
- The `form` element isn't required to have an `action` attribute.
- Form elements, such as `input`, can be placed as direct children of the `form` element; in XHTML, another element (such as `fieldset` or `div`) was required to wrap form elements.
- The `textarea` element is not required to have `rows` and `cols` attributes.
- The `target` attribute, deprecated and thus invalid in XHTML, is now valid in HTML5.
- Block elements can be placed inside `a` elements.

- The ampersand character (&) doesn't need to be encoded as `&` if it appears as text on the page.

That's a fairly comprehensive, though hardly exhaustive, list of differences between XHTML and HTML5 validation. Some are style choices, so you're encouraged to choose a style and be consistent. We outlined some preferred style choices in the previous chapter, and you're welcome to incorporate some if not all of those suggestions in your own HTML5 projects.



Lint Tools

If you want to validate your markup's syntax style using stricter guidelines, you can use an HTML5 **lint tool**, such as <http://lint.brihten.com/html/>. At the time of writing, it's still in development, but it works well. You can use it to check that your attributes and tags are lowercase, that void tags are self-closed, that Boolean attributes omit their value, that closing tags are never omitted—or any combination of these style rules. It can even ensure that your markup is indented consistently!

Summary

By now, we've gotten our heads around just about all the new semantic and syntactic changes in HTML5. Some of this information may be a little hard to digest straight away, but don't worry! The best way to become familiar with HTML5 is to use it—you can start with your next project. Try using some of the structural elements we covered in the last chapter, or some of the text-level semantics we saw in this chapter. If you're unsure about how exactly an element is meant to be used, go back and read the section about it, or better yet, read the specification itself. While the language is certainly drier than the text in this book (at least, we hope it is!), the specifications can give you a more complete picture of how a given element is intended to be used. Remember that the HTML5 specification is still in development, so some of what we've covered is still subject to change. The specifications will always contain the most up-to-date information.

In the next chapter, we'll look at a crucial segment of new functionality introduced in HTML5: forms and form-related features.

Chapter 4

HTML5 Forms

We’ve coded most of the page, and you now know most of what there is to know about new HTML5 elements and their semantics. But before we start work on the *look* of the site—which we do in Chapter 6—we’ll take a quick detour away from *The HTML5 Herald’s* front page to have a look at the sign-up page. This will illustrate what HTML5 has to offer in terms of web forms.

HTML5 web forms have introduced new form elements, input types, attributes, and other features. Many of these features we’ve been using in our interfaces for years: form validation, combo boxes, placeholder text, and the like. The difference is that where before we had to resort to JavaScript to create these behaviors, they’re now available directly in the browser; all you need to do is set an attribute in your markup to make them available.

HTML5 not only makes marking up forms easier on the developer, it’s also better for the user. With client-side validation being handled natively by the browser, there will be greater consistency across different sites, and many pages will load faster without all that redundant JavaScript.

Let’s dive in!

Dependable Tools in Our Toolbox

Forms are often the last thing developers include in their pages—many developers find forms just plain boring. The good news is that HTML5 injects a little bit more joy into coding forms. By the end of this chapter, we hope you'll look forward to employing form elements, as appropriate, in your markup.

Let's start off our sign-up form with plain, old-fashioned HTML:

register.html (excerpt)

```
<form id="register" method="post">
  <hgroup>
    <h1>Sign Me Up!</h1>
    <h2>I would like to receive your fine publication.</h2>
  </hgroup>

  <ul>
    <li>
      <label for="register-name">My name is:</label>
      <input type="text" id="register-name" name="name">
    </li>
    <li>
      <label for="address">My email address is:</label>
      <input type="text" id="address" name="address">
    </li>
    <li>
      <label for="url">My website is located at:</label>
      <input type="text" id="url" name="url">
    </li>
    <li>
      <label for="password">I would like my password to be:</label>
      <p>(at least 6 characters, no spaces)</p>
      <input type="password" id="password" name="password">
    </li>
    <li>
      <label for="rating">On a scale of 1 to 10, my knowledge of
➡HTML5 is:</label>
      <input type="text" name="rating" id="rating">
    </li>
    <li>
      <label for="startdate">Please start my subscription on:
➡</label>
      <input type="text" id="startdate" name="startdate">
```

```

    </li>
    <li>
      <label for="quantity">I would like to receive <input
➡type="text" name="quantity" id="quantity"> copies of <cite>
➡The HTML5 Herald</cite>.</label>
    </li>
    <li>
      <label for="upsell">Also sign me up for <cite>The CSS3
➡Chronicle</cite></label>
      <input type="checkbox" id="upsell" name="upsell">
    </li>
    <li>
      <input type="submit" id="register-submit" value="Send Post
➡Haste">
    </li>
  </ul>
</form>

```

This sample registration form uses form elements that have been available since the earliest versions of HTML. This form provides clues to users about what type of data is expected in each field via the `label` and `p` elements, so even your users on Netscape 4.7 and IE5 (kidding!) can understand the form. It works, but it can certainly be improved upon.

In this chapter we're going to enhance this form to include HTML5's features. HTML5 provides new input types specific to email addresses, URLs, numbers, dates, and more. In addition to those new input types, HTML5 also introduces attributes that can be used with both new and existent input types. These allow you to provide placeholder text, mark fields as required, and declare what type of data is acceptable—all without JavaScript.

We'll cover all the newly added input types later in the chapter. Before we do that, let's take a look at the new form attributes HTML5 provides.

HTML5 Form Attributes

For years, developers have written (or copied and pasted) snippets of JavaScript to validate the information users entered into form fields: what elements are required, what type of data is accepted, and so on. HTML5 provides us with several attributes that allow us to dictate what is an acceptable value, and inform the user of errors, all without the use of any JavaScript.

Browsers that support these HTML5 attributes will compare data entered by the user against regular expression patterns provided by the developer (you). Then they check to see if all required fields are indeed filled out, enable multiple values if allowed, and so on. Even better, including these attributes won't harm older browsers; they'll simply ignore the attributes they don't understand. In fact, you can use these attributes and their values to power your scripting fallbacks, instead of hardcoding validation patterns into your JavaScript code, or adding superfluous classes to your markup. We'll look at how this is done a bit later; for now, let's go through each of the new attributes.

The required Attribute

The Boolean `required` attribute tells the browser to only submit the form if the field in question is filled out correctly. Obviously, this means that the field can't be left empty, but it also means that, depending on other attributes or the field's type, only certain types of values will be accepted. Later in the chapter, we'll be covering different ways of letting browsers know what kind of data is expected in a form.

If a required field is empty or invalid, the form will fail to submit, and focus will move to the first invalid form element. Opera, Firefox, and Chrome provide the user with error messages; for example, "Please fill out this field" or "You have to specify a value" if left empty, and "Please enter an email address" or "xyz is not in the format this page requires" when the data type or pattern is wrong.



Out of focus?

Time for a quick refresher: a form element is **focused** either when a user clicks on the field with their mouse, or tabs to it with their keyboard. For `input` elements, typing with the keyboard will enter data into that element.

In JavaScript terminology, the `focus` event will fire on a form element when it receives focus, and the `blur` event will fire when it *loses* focus.

In CSS, the `:focus` pseudo-class can be used to style elements that currently have focus.

The `required` attribute can be set on any input type except `button`, `range`, `color`, and `hidden`, all of which generally have a default value. As with other Boolean at-

tributes we've seen so far, the syntax is either simply `required`, or `required="required"` if you're using XHTML syntax.

Let's add the `required` attribute to our sign-up form. We'll make the name, email address, password, and subscription start date fields required:

register.html (excerpt)

```
<ul>
  <li>
    <label for="register-name">My name is:</label>
    <input type="text" id="register-name" name="name"
    ➤required aria-required="true">
  </li>
  <li>
    <label for="email">My email address is:</label>
    <input type="text" id="email" name="email"
    ➤required aria-required="true">
  </li>
  <li>
    <label for="url">My website is located at:</label>
    <input type="text" id="url" name="url">
  </li>
  <li>
    <label for="password">I would like my password to be:</label>
    <p>(at least 6 characters, no spaces)</p>
    <input type="password" id="password" name="password"
    ➤required aria-required="true">
  </li>
  <li>
    <label for="rating">On a scale of 1 to 10, my knowledge of
    ➤HTML5 is:</label>
    <input type="text" name="rating" type="range">
  </li>
  <li>
    <label for="startdate">Please start my subscription on:
    ➤</label>
    <input type="text" id="startdate" name="startdate"
    ➤required aria-required="true">
  </li>
  <li>
    <label for="quantity">I would like to receive <input
    ➤type="text" name="quantity" id="quantity"> copies of <cite>
    ➤The HTML5 Herald</cite></label>
  </li>
```

```

<li>
  <label for="upsell">Also sign me up for <cite>The CSS3
  ➔Chronicle</cite></label>
  <input type="checkbox" id="upsell" name="upsell">
</li>
<li>
  <input type="submit" id="register-submit" value="Send Post
  ➔Haste">
</li>
</ul>

```

For improved accessibility, whenever the required attribute is included, add the ARIA attribute `aria-required="true"`. Many screen readers lack support for the new HTML5 attributes, but many *do* have support for WAI-ARIA roles, so there's a chance that adding this role could let a user know that the field is required—see Appendix B for a brief introduction to WAI-ARIA.

Figure 4.1, Figure 4.2, and Figure 4.3 show the behavior of the required attribute when you attempt to submit the form.



Figure 4.1. The required field validation message in Firefox 4

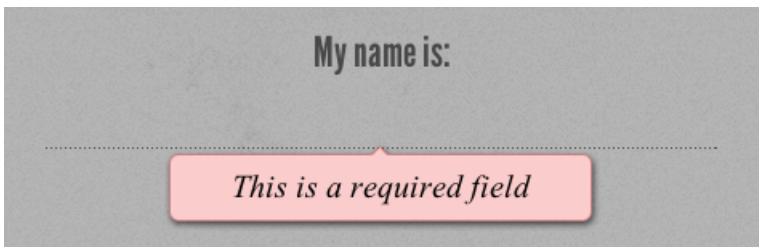


Figure 4.2. How it looks in Opera ...



Figure 4.3. ... and in Google Chrome

Styling Required Form Fields

You can style required form elements with the `:required` pseudo-class. You can also style valid or invalid fields with the `:valid` and `:invalid` pseudo-classes. With these pseudo-classes and a little CSS magic, you can provide visual cues to sighted users indicating which fields are required, and also give feedback for successful data entry:

```
input:required {
  background-image: url('../images/required.png');
}
input:focus:invalid {
  background-image: url('../images/invalid.png');
}
input:focus:valid {
  background-image: url('../images/valid.png');
}
```

We're adding a background image (an asterisk) to required form fields. We've also added separate background images to valid and invalid fields. The change is only apparent when the form element has focus, to keep the form from looking too cluttered.



Beware Default Styles

Note that Firefox 4 applies its own styles to invalid elements (a red shadow), as shown in Figure 4.1 earlier. You may want to remove the native drop shadow with the following CSS:

```
:invalid { box-shadow: none; }
```



Backwards Compatibility

Older browsers mightn't support the `:required` pseudo-class, but you can still provide targeted styles using the attribute selector:

```
input:required,


```

You can also use this attribute as a hook for form validation in browsers without support for HTML5. Your JavaScript code can check for the presence of the `required` attribute on empty elements, and fail to submit the form if any are found.

The placeholder Attribute

The `placeholder` attribute allows a short hint to be displayed inside the form element, space permitting, telling the user what data should be entered in that field. The placeholder text disappears when the field gains focus, and reappears on blur if no data was entered. Developers have provided this functionality with JavaScript for years, but in HTML5 the `placeholder` attribute allows it to happen natively, with no JavaScript required.

For *The HTML5 Herald's* sign-up form, we'll put a placeholder on the website URL and start date fields:

[register.html \(excerpt\)](#)

```
<li>
    <label for="url">My website is located at:</label>
    <input type="text" id="url" name="url"
    ➤placeholder="http://example.com">
</li>
:
<li>
    <label for="startdate">Please start my subscription on:</label>
    <input type="text" id="startdate" name="startdate" required
    ➤aria-required="true" placeholder="1911-03-17">
</li>
```

Because support for the `placeholder` attribute is still restricted to the latest crop of browsers, you shouldn't rely on it as the only way to inform users of requirements. If your hint exceeds the size of the field, describe the requirements in the input's `title` attribute or in text next to the input element.

Currently, Safari, Chrome, Opera, and Firefox 4 support the `placeholder` attribute.

Polyfilling Support with JavaScript

Like everything else in this chapter, it won't hurt nonsupporting browsers to include the `placeholder` attribute.

As with the `required` attribute, you can make use of the `placeholder` attribute and its value to make older browsers behave as if they supported it—all by using a little JavaScript magic.

Here's how you'd go about it: first, use JavaScript to determine which browsers lack support. Then, in those browsers, use a function that creates a “faux” placeholder. The function needs to determine which form fields contain the `placeholder` attribute, then temporarily grab that attribute's content and put it in the `value` attribute.

Then you need to set up two event handlers: one to clear the field's value on focus, and another to replace the placeholder value on blur if the form control's value is still `null` or an empty string. If you do use this trick, make sure that the value of your `placeholder` attribute isn't one that users might actually enter, and remember to clear the faux placeholder when the form is submitted. Otherwise, you'll have lots of “(XXX) XXX-XXXX” submissions!

Let's look at a sample JavaScript snippet (using the jQuery JavaScript library for brevity) to progressively enhance our form elements using the `placeholder` attribute.



jQuery

In the code examples that follow, and throughout the rest of the book, we'll be using the jQuery¹ JavaScript library. While all the effects we'll be adding could be accomplished with plain JavaScript, we find that jQuery code is generally more readable; thus, it helps to illustrate what we want to focus on—the HTML5 APIs—rather than spending time explaining a lot of hairy JavaScript.

¹ <http://jquery.com/>

Here's our placeholder polyfill:

register.html (excerpt)

```
<script>
if(!Modernizr.input.placeholder) {

    $("input[placeholder], textarea[placeholder]").each(function() {
        if($(this).val()==""){
            $(this).val($(this).attr("placeholder"));
            $(this).focus(function(){
                if($(this).val()==$(this).attr("placeholder")) {
                    $(this).val("");
                    $(this).removeClass('placeholder');
                }
            });
            $(this).blur(function(){
                if($(this).val()==""){
                    $(this).val($(this).attr("placeholder"));
                    $(this).addClass('placeholder');
                }
            });
        }
    });

    $('form').submit(function(){
        // first do all the checking for required
        // element and form validation.
        // Only remove placeholders before final submission
        var placeheld = $(this).find('[placeholder]');
        for(var i=0; i<placeheld.length; i++){
            if$(placeheld[i]).val() ==
            ➡$(placeheld[i]).attr('placeholder')) {
                // if not required, set value to empty before submitting
                $(placeheld[i]).attr('value','');
            }
        }
    });
}
</script>
```

The first point to note about this script is that we're using the Modernizr² JavaScript library to detect support for the placeholder attribute. There's more information

² <http://www.modernizr.com/>

about Modernizr in Appendix A, but for now it's enough to understand that it provides you with a whole raft of `true` or `false` properties for the presence of given HTML5 and CSS3 features in the browser. In this case, the property we're using is fairly self-explanatory. `Modernizr.input.placeholder` will be `true` if the browser supports `placeholder`, and `false` if it doesn't.

If we've determined that `placeholder` support is absent, we grab all the `input` and `textarea` elements on the page with a `placeholder` attribute. For each of them, we check that the value isn't empty, then replace that value with the value of the `placeholder` attribute. In the process, we add the `placeholder` class to the element, so you can lighten the color of the font in your CSS, or otherwise make it look more like a native placeholder. When the user focuses on the input with the faux placeholder, the script clears the value and removes the class. When the user removes focus, the script checks to see if there is a value. If not, we add the placeholder text and class back in.

This is a great example of an HTML5 polyfill: we use JavaScript to provide support *only for those browsers that lack native support*, and we do it by *leveraging the HTML5 elements and attributes already in place*, rather than resorting to additional classes or hardcoded values in our JavaScript.

The pattern Attribute

The `pattern` attribute enables you to provide a regular expression that the user's input must match in order to be considered valid. For any input where the user can enter free-form text, you can limit what syntax is acceptable with the `pattern` attribute.

The regular expression language used in patterns is the same Perl-based regular expression syntax as JavaScript, except that the `pattern` attribute must match the entire value, not just a subset. When including a `pattern`, you should always indicate to users what is the expected (and required) pattern. Since browsers currently show the value of the `title` attribute on hover like a tooltip, include pattern instructions that are more detailed than placeholder text, and which form a coherent statement.



The Skinny on Regular Expressions

Regular expressions are a feature of most programming languages that allow developers to specify patterns of characters and check to see if a given string matches the pattern. Regular expressions are famously indecipherable to the uninitiated. For instance, one possible regular expression to check if a string is formatted as an email address looks like this: `[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}`.

A full tutorial on the syntax of regular expressions is beyond the scope of this book, but there are plenty of great resources and tutorials available online if you'd like to learn. Alternately, you can search the Web or ask around on forums for a pattern that will serve your purposes.

For a simple example, let's add a `pattern` attribute to the password field in our form. We want to enforce the requirement that the password be at least six characters long, with no spaces:

register.html (excerpt)

```
<li>
  <label for="password">I would like my password to be:</label>
  <p>(at least 6 characters, no spaces)</p>
  <input type="password" id="password" name="password" required
  ➔pattern="\S{6,}">
</li>
```

`\S` refers to “any nonwhitespace character,” and `{6,}` means “at least six times.” If you wanted to stipulate the maximum amount of characters, the syntax would be, for example, `\S{6,10}` for between six and ten characters.

As with the `required` attribute, the `pattern` attribute will prevent the form being submitted if the pattern isn't matched, and will provide an error message.

If your pattern is not a valid regular expression, it will be ignored for the purposes of validation. Note also that similar to the `placeholder` and `required` attributes, you can use the value of this attribute to provide the basis for your JavaScript validation code for nonsupporting browsers.

The disabled Attribute

The Boolean `disabled` attribute has been around longer than HTML5, but it has been expanded on, to a degree. It can be used with any form control except the new output element—and unlike previous versions of HTML, HTML5 allows you to set the `disabled` attribute on a `fieldset` and have it apply to all the form elements contained in that `fieldset`.

Generally, form elements with the `disabled` attribute have the content grayed out in the browser—the text is lighter than the color of values in enabled form controls. Browsers will prohibit the user from focusing on a form control that has the `disabled` attribute set. This attribute is often used to disable the submit button until all fields are correctly filled out, for example.

You can employ the `:disabled` pseudo-class in your CSS to style disabled form controls.

Form controls with the `disabled` attribute aren't submitted along with the form; so their values will be inaccessible to your form processing code on the server side. If you want a value that users are unable to edit, but can still see and submit, use the `readonly` attribute.

The readonly Attribute

The `readonly` attribute is similar to the `disabled` attribute: it makes it impossible for the user to edit the form field. Unlike `disabled`, however, the field *can* receive focus, and its value is submitted with the form.

In a comments form, we may want to include the URL of the current page or the title of the article that is being commented on, letting the user know that we are collecting this data without allowing them to change it:

```
<label for="about">Article Title</label>
<input type="text" name="about" id="about" readonly>
```

The multiple Attribute

The `multiple` attribute, if present, indicates that multiple values can be entered in a form control. While it has been available in previous versions of HTML, it only applied to the `select` element. In HTML5, it can be added to `email` and `file` input

types as well. If present, the user can select more than one file, or include several comma-separated email addresses.

At the time of writing, multiple file input is only supported in Chrome, Opera, and Firefox.



Spaces or Commas?

You may notice that the iOS touch keyboard for email inputs includes a space. Of course, spaces aren't permitted in email addresses, but some browsers allow you to separate multiple emails with spaces. Firefox 4 and Opera both support multiple emails separated with either commas or spaces. WebKit has no support for the space separator, even though the space is included in the touch keyboard.

Soon, all browsers will allow extra whitespace. This is how most users will likely enter the data; plus, this allowance has recently been added to the specification.

The form Attribute

Not to be confused with the `form` element, the `form attribute` in HTML5 allows you to associate form elements with forms in which they're not nested. This means you can now associate a fieldset or form control with any other form in the document. The `form` attribute takes as its value the `id` of the `form` element with which the fieldset or control should be associated.

If the attribute is omitted, the control will only be submitted with the form in which it's nested.

The autocomplete Attribute

The `autocomplete` attribute specifies whether the form, or a form control, should have autocomplete functionality. For most form fields, this will be a drop-down that appears when the user begins typing. For password fields, it's the ability to save the password in the browser. Support for this attribute has been present in browsers for years, though it was never in the specification until HTML5.

By default, `autocomplete` is on. You may have noticed this the last time you filled out a form. In order to disable it, use `autocomplete="off"`. This is a good idea for sensitive information, such as a credit card number, or information that will never need to be reused, like a CAPTCHA.

Autocompletion is also controlled by the browser. The user will have to turn on the autocomplete functionality in their browser for it to work at all; however, setting the autocomplete attribute to off overrides this preference.

The datalist Element and the list Attribute

Datalists are currently only supported in Firefox and Opera, but they are very cool. They fulfill a common requirement: a text field with a set of predefined autocomplete options. Unlike the `select` element, the user can enter whatever data they like, but they'll be presented with a set of suggested options in a drop-down as they type.

The `datalist` element, much like `select`, is a list of options, with each one placed in an `option` element. You then associate the `datalist` with an input using the `list` attribute on the input. The `list` attribute takes as its value the `id` attribute of the `datalist` you want to associate with the input. One `datalist` can be associated with several input fields.

Here's what this would look like in practice:

```
<label for="favcolor">Favorite Color</label>
<input type="text" list="colors" id="favcolor" name="favcolor">

<datalist id="colors">
  <option value="Blue">
  <option value="Green">
  <option value="Pink">
  <option value="Purple">
</datalist>
```

In supporting browsers, this will display a simple text field that drops down a list of suggested answers when focused. Figure 4.4 shows what this looks like.

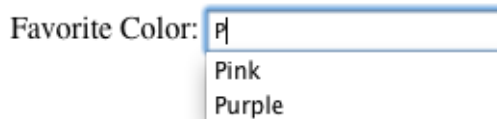


Figure 4.4. The `datalist` element in action in Firefox

The autofocus Attribute

The Boolean `autofocus` attribute specifies that a form control should be focused as soon as the page loads. Only one form element can have `autofocus` in a given page.

HTML5 New Form Input Types

You're probably already familiar with the `input` element's `type` attribute. This is the attribute that determines what kind of form input will be presented to the user. If it is omitted—or, in the case of new input types and older browsers, not understood—it still works: the `input` will default to `type="text"`. This is the key that makes HTML5 forms usable today. If you use a new input type, like `email` or `search`, older browsers will simply present users with a standard text field.

Our sign-up form currently uses four of the ten input types you're familiar with: `checkbox`, `text`, `password`, and `submit`. Here's the full list of types that were available before HTML5:

- `button`
- `checkbox`
- `file`
- `hidden`
- `image`
- `password`
- `radio`
- `reset`
- `submit`
- `text`

HTML5 gives us input types that provide for more data-specific UI elements and native data validation. HTML5 has a total of 13 new input types:

- `search`
- `email`
- `url`
- `tel`
- `datetime`
- `date`

- month
- week
- time
- datetime-local
- number
- range
- color

Let's look at each of these new types in detail, and see how we can put them to use.

Search

The search input type (`type="search"`) provides a search field—a one-line text input control for entering one or more search terms. The spec states:

The difference between the text state and the search state is primarily stylistic: on platforms where search fields are distinguished from regular text fields, the search state might result in an appearance consistent with the platform's search fields rather than appearing like a regular text field.

Many browsers style search inputs in a manner consistent with the browser or the operating system's search boxes. Some browsers have added the ability to clear the input with the click of a mouse, by providing an **x** icon once text is entered into the field. You can see this behavior in Chrome on Mac OS X in Figure 4.5.

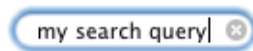


Figure 4.5. The search input type is styled to resemble the operating system's search fields

Currently, only Chrome and Safari provide a button to clear the field. Opera 11 displays a rounded corner box without a control to clear the field, but switches to display a normal text field if any styling, such as a background color, is applied.

While you can still use `type="text"` for search fields, the new search type is a visual cue as to where the user needs to go to search the site, and provides an interface the user is accustomed to. *The HTML5 Herald* has no search field, but here's an example of how you'd use it:

```
<form id="search" method="get">
  <input type="search" id="s" name="s">
  <input type="submit" value="Search">
</form>
```

Since `search`, like all the new input types, appears as a regular text box in nonsupporting browsers, there's no reason not to use it when appropriate.

Email Addresses

The `email` type (`type="email"`) is, unsurprisingly, used for specifying one or more email addresses. It supports the Boolean `multiple` attribute, allowing for multiple, comma-separated email addresses.

Let's change our form to use `type="email"` for the registrant's email address:

register.html (excerpt)

```
<label for="email">My email address is</label>
<input type="email" id="email" name="email">
```

If you change the input type from `text` to `email`, as we've done here, you'll notice no visible change in the user interface; the input still looks like a plain text field. However, there are differences behind the scenes.

The change becomes apparent if you're using an iOS device. When you focus on the email field, the iPhone, iPad, and iPod will all display a keyboard optimized for email entry (with a shortcut key for the `@` symbol), as shown in Figure 4.6.



Figure 4.6. The `email` input type provides a specialized keyboard on iOS devices

Firefox, Chrome, and Opera also provide error messaging for `email` inputs: if you try to submit a form with content unrecognizable as one or more email addresses, the browser will tell you what is wrong. The default error messages are shown in Figure 4.7.



Figure 4.7. Error messages for incorrectly formatted email addresses on Firefox 4 (left) and Opera 11 (right)



Custom Validation Messages

Don't like the error messages provided? In some browsers, you can set your own with `.setCustomValidity(errorMessage)`. `setCustomValidity` takes as its only parameter the error message you want to provide. You can pass an empty string to `setCustomValidity` if you want to remove the error message entirely.

Unfortunately, while you can change the *content* of the message, you're stuck with its appearance, at least for now.

URLs

The `url` input (`type="url"`) is used for specifying a web address. Much like `email`, it will display as a normal text field. On many touch screens, the on-screen keyboard displayed will be optimized for web address entry, with a forward slash (/) and a “.com” shortcut key.

Let's update our registration form to use the `url` input type:

register.html (excerpt)

```
<label for="url">My website is located at:</label>
<input type="url" id="url" name="url">
```

Opera, Firefox, and WebKit support the `url` input type, reporting the input as invalid if the URL is incorrectly formatted. Only the general format of a URL is validated,

so, for example, `q://example.xyz` will be considered valid, even though `q://` isn't a real protocol and `.xyz` isn't a real top-level domain. As such, if you want the value entered to conform to a more specific format, provide information in your label (or in a placeholder) to let your users know, and use the `pattern` attribute to ensure that it's correct—we'll cover `pattern` in detail later in this chapter.



WebKit

When we refer to WebKit in this book, we're referring to browsers that use the WebKit rendering engine. This includes Safari (both on the desktop and on iOS), Google Chrome, the Android browser, and a number of other mobile browsers. You can find more information about the WebKit open source project at <http://www.webkit.org/>.

Telephone Numbers

For telephone numbers, use the `tel` input type (`type="tel"`). Unlike the `url` and `email` types, the `tel` type doesn't enforce a particular syntax or pattern. Letters and numbers—indeed, any characters other than new lines or carriage returns—are valid. There's a good reason for this: all over the world countries have different types of valid phone numbers, with various lengths and punctuation, so it would be impossible to specify a single format as standard. For example, in the USA, `+1(415)555-1212` is just as well understood as `415.555.1212`.

You can encourage a particular format by including a placeholder with the correct syntax, or a comment after the input with an example. Additionally, you can stipulate a format by using the `pattern` attribute or the `setCustomValidity` method to provide for client-side validation.

Numbers

The `number` type (`type="number"`) provides an input for entering a number. Usually, this is a “spinner” box, where you can either enter a number or click on the up or down arrows to select a number.

Let's change our quantity field to use the `number` input type:

register.html (excerpt)

```
<label for="quantity">I would like to receive <input type="number"
➡name="quantity" id="quantity"> copies of <cite>The HTML5 Herald
➡</cite></label>
```

Figure 4.8 shows what this looks like in Opera.



Figure 4.8. The number input seen in Opera

The number input has `min` and `max` attributes to specify the minimum and maximum values allowed. We highly recommend that you use these, otherwise the up and down arrows might lead to different (and very odd) values depending on the browser.



When is a number not a number?

There will be times when you may think you want to use `number`, when in reality another input type is more appropriate. For example, it might seem to make sense that a street address should be a number. But think about it: would you want to click the spinner box all the way up to 34154? More importantly, many street numbers have non-numeric portions: think 24½ or 36B, neither of which work with the `number` input type.

Additionally, account numbers may be a mixture of letters and numbers, or have dashes. If you know the pattern of your number, use the `pattern` attribute. Just remember not to use `number` if the range is extensive or the number could contain non-numeric characters and the field is required. If the field is optional, you might want to use `number` anyway, in order to prompt the number keyboard as the default on touchscreen devices.

If you do decide that `number` is the way to go, remember also that the `pattern` attribute is unsupported in the `number` type. In other words, if the browser supports the `number` type, that supersedes any `pattern`. That said, feel free to include a `pattern`, in case the browser supports `pattern` but not the `number` input type.

You can also provide a `step` attribute, which determines the increment by which the number steps up or down when clicking the up and down arrows. The `min`, `max`, and `step` attributes are supported in Opera and WebKit.

On many touchscreen devices, focusing on a number input type will bring up a number touch pad (rather than a full keyboard).

Ranges

The range input type (`type="range"`) displays a slider control in browsers that support it (currently Opera and WebKit). As with the number type, it allows the `min`, `max`, and `step` attributes. The difference between number and range, according to the spec, is that the exact value of the number is unimportant with range. It's ideal for inputs where you want an imprecise number; for example, a customer satisfaction survey asking clients to rate aspects of the service they received.

Let's change our registration form to use the range input type. The field asking users to rate their knowledge of HTML5 on a scale of 1 to 10 is perfect:

register.html (excerpt)

```
<label for="rating">On a scale of 1 to 10, my knowledge of HTML5  
is:</label>  
<input type="range" min="1" max="10" name="rating" type="range">
```

The `step` attribute defaults to 1, so it's not required. Figure 4.9 shows what this input type looks like in Safari.

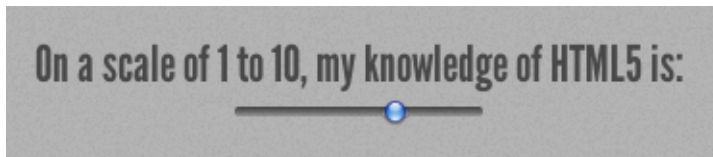


Figure 4.9. The range input type in Chrome

The default value of a range is the midpoint of the slider—in other words, halfway between the minimum and the maximum.

The spec allows for a reversed slider (with values from right to left instead of from left to right) if the maximum specified is less than the minimum; however, currently no browsers support this.

Colors

The `color` input type (`type="color"`) provides the user with a color picker—or at least it does in Opera (and, surprisingly, in the built-in browser on newer BlackBerry smartphones). The color picker should return a hexadecimal RGB color value, such as `#FF3300`.

Until this input type is fully supported, if you want to use a color input, provide placeholder text indicating that a hexadecimal RGB color format is required, and use the `pattern` attribute to restrict the entry to only valid hexadecimal color values.

We don't use color in our form, but, if we did, it would look a little like this:

```
<label for="clr">Color: </label>
<input id="clr" name="clr" type="text" placeholder="#FFFFFF"
  ➔pattern="#(?:[0-9A-Fa-f]){6}|[0-9A-Fa-f]{3}" required>
```

The resulting color picker is shown in Figure 4.10. Clicking the **Other...** button brings up a full color wheel, allowing the user to select any hexadecimal color value.



Figure 4.10. Opera's color picker control for the `color` input type

WebKit browsers support the color input type as well, and can indicate whether the color is valid, but don't provide a color picker ... yet.

Dates and Times

There are several new date and time input types, including `date`, `datetime`, `datetime-local`, `month`, `time`, and `week`. All date and time inputs accept data formatted according to the ISO 8601 standard.³

³ http://en.wikipedia.org/wiki/ISO_8601

date

This comprises the date (year, month, and day), but no time; for example, 2004-06-24.

month

Only includes the year and month; for example, 2012-12.

week

This covers the year and week number (from 1 to 52); for example, 2011-W01 or 2012-W52.

time

A time of day, using the military format (24-hour clock); for example, 22:00 instead of 10.00 p.m.

datetime

This includes both the date and time, separated by a “T”, and followed by either a “Z” to represent UTC (Coordinated Universal Time), or by a time zone specified with a + or - character. For example, “2011-03-17T10:45-5:00” represents 10:45am on the 17th of March, 2011, in the UTC minus 5 hours time zone (Eastern Standard Time).

datetime-local

Identical to `datetime`, except that it omits the time zone.

The most commonly used of these types is `date`. The specifications call for the browser to display a date control, yet at the time of writing, only Opera does this by providing a calendar control.

Let’s change our subscription start date field to use the `date` input type:

[register.html](#) (*excerpt*)

```
<label for="startdate">Please start my subscription on:</label>
<input type="date" min="1904-03-17" max="1904-05-17"
  ➤id="startdate" name="startdate" required aria-required="true"
  ➤placeholder="1911-03-17">
```

Now, we’ll have a calendar control when we view our form in Opera, as shown in Figure 4.11. Unfortunately, it’s unable to be styled with CSS at present.

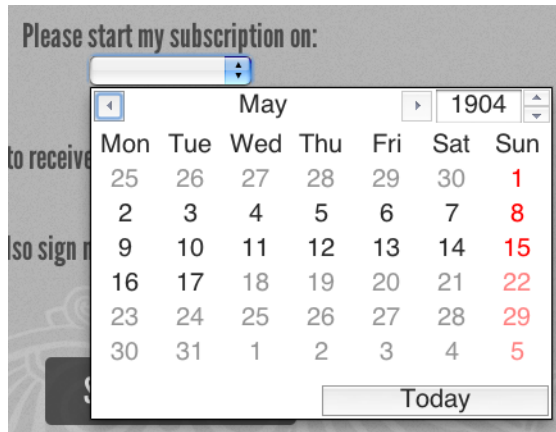


Figure 4.11. Opera's date picker for date, datetime, datetime-local, week, and month input types

For the month and week types, Opera displays the same date picker, but only allows the user to select full months or weeks. In those cases, individual days are unable to be selected; instead, clicking on a day selects the whole month or week.

Currently, WebKit provides *some* support for the date input type, providing a user interface similar to the number type, with up and down arrows. Safari behaves a little oddly when it comes to this control; the default value is the very first day of the Gregorian calendar: 1582-10-15. The default in Chrome is 0001-01-01, and the maximum is 275760-09-13. Opera functions more predictably, with the default value being the current date. Because of these oddities, we highly recommend including a minimum and maximum when using any of the date-based input types (all those listed above, except time). As with number, this is done with the min and max attributes.

The placeholder attribute we added to our start date field earlier is made redundant in Opera by the date picker interface, but it makes sense to leave it in place to guide users of other browsers.

Eventually, when all browsers support the UI of all the new input types, the placeholder attribute will only be relevant on text, search, URL, telephone, email, and password types. Until then, placeholders are a good way to hint to your users what kind of data is expected in those fields—remember that they'll just look like regular text fields in nonsupporting browsers.



Dynamic Dates

In our example above, we hardcoded the `min` and `max` values into our HTML. If, for example, you wanted the minimum to be the day after the current date (this makes sense for a newspaper subscription start date), this would require updating the HTML every day. The best thing to do is dynamically generate the minimum and maximum allowed dates on the server side. A little PHP can go a long way:

```
<?php
function daysFromNow($days){
    $added = ($days * 24 * 3600) + time();
    echo(date("Y-m-d", $added));
}
?>
```

In our markup where we had static dates, we now dynamically create them with the above function:

```
<li>
    <label for="startdate">Please start my subscription on:
    ↪</label>
    <input type="date" min="<?php daysFromNow(1); ?>"
    ↪max="<?php daysFromNow(60); ?>" id="startdate"
    ↪name="startdate" required aria-required="true"
    ↪placeholder="1911-03-17">
</li>
```

This way, the user is limited to entering dates that make sense in the context of the form.

You can also include the `step` attribute with the `date` and `time` input types. For example, `step="6"` on month will limit the user to selecting either January or July. On `time` and `datetime` inputs, the `step` attribute must be expressed in seconds, so `step="900"` on the `time` input type will cause the input to step in increments of 15 minutes.

Other New Form Controls in HTML5

We've covered the new values for the `input` element's `type` attribute, along with some attributes that are valid on most form elements. But HTML5 web forms still have more to offer us! There are four new form elements in HTML5: `output`, `keygen`, `progress`, and `meter`. We covered `progress` and `meter` in the last chapter, since they're often useful outside of forms, so let's take a look at the other two elements.

The `output` Element

The purpose of the `output` element is to accept and display the result of a calculation. The `output` element should be used when the user can see the value, but not directly manipulate it, and when the value can be derived from other values entered in the form. An example use might be the total cost calculated after shipping and taxes in a shopping cart.

The `output` element's value is contained between the opening and closing tags. Generally, it will make sense to use JavaScript in the browser to update this value. The `output` element has a `for` attribute, which is used to reference the `ids` of form fields whose values went into the calculation of the `output` element's value.

It's worth noting that the `output` element's name and value are submitted along with the form.

The `keygen` Element

The `keygen` element is a control for generating a public-private keypair⁴ and for submitting the public key from that key pair. Opera, WebKit, and Firefox all support this element, rendering it as a drop-down menu with options for the length of the generated keys; all provide different options, though.

The `keygen` element introduces two new attributes: the `challenge` attribute specifies a string that is submitted along with the public key, and the `keytype` attribute specifies the type of key generated. At the time of writing, the only supported `keytype` value is `rsa`, a common algorithm used in public-key cryptography.

⁴ http://en.wikipedia.org/wiki/Public-key_cryptography

Changes to Existing Form Controls and Attributes

There have been a few other changes to form controls in HTML5.

The `form` Element

Throughout this chapter, we've been talking about attributes that apply to various form field elements; however, there are also some new attributes specific to the `form` element itself.

First, as we've seen, HTML5 provides a number of ways to natively validate form fields; certain input types such as `email` and `url`, for example, as well as the `required` and `pattern` attributes. You may, however, want to use these input types and attributes for styling or semantic reasons without preventing the form being submitted. The new Boolean `novalidate` attribute allows a form to be submitted without native validation of its fields.

Next, forms no longer need to have the `action` attribute defined. If omitted, the form will behave as though the `action` were set to the current page.

Lastly, the `autocomplete` attribute we introduced earlier can also be added directly to the `form` element; in this case, it will apply to all fields in that form unless those fields override it with their own `autocomplete` attribute.

The `optgroup` Element

In HTML5, you can have an `optgroup` as a child of another `optgroup`, which is useful for multilevel select menus.

The `textarea` Element

In HTML 4, we were required to specify a `textarea` element's size by specifying values for the `rows` and `cols` attributes. In HTML5, these attributes are no longer required; you should use CSS to define a `textarea`'s width and height.

New in HTML5 is the `wrap` attribute. This attribute applies to the `textarea` element, and can have the values `soft` (the default) or `hard`. With `soft`, the text is submitted without line breaks other than those actually entered by the user, whereas `hard` will

submit any line breaks introduced by the browser due to the size of the field. If you set the `wrap` to `hard`, you need to specify a `cols` attribute.

In Conclusion

As support for HTML5 input elements and attributes grows, sites will require less and less JavaScript for client-side validation and user interface enhancements, while browsers handle most of the heavy lifting. Legacy user agents are likely to stick around for the foreseeable future, but there is no reason to avoid moving forward and using HTML5 web forms, with appropriate polyfills and fallbacks filling the gaps where required.

In the next chapter, we'll continue fleshing out *The HTML5 Herald* by adding what many consider to be HTML5's killer feature: native video and audio.

Chapter 5

HTML5 Audio and Video

No book on HTML5 would be complete without an examination of the new `video` and `audio` elements. These ground-breaking new elements have already been utilized on the Web, albeit in a limited capacity, but more and more developers and content creators are starting to incorporate them into their projects.

For *The HTML5 Herald*, we're going to be placing a `video` element in the first column of our three-column layout. But before we explore the details of the `video` element and its various attributes and associated elements, let's take a brief look at the state of video on the Web today.

For the most part, this chapter will focus on the `video` element, since that's what we're using in our sample project. However, the `audio` element behaves relatively identically: almost all the attributes and properties that we'll be using for video also apply to audio. Where there are exceptions, we'll be sure to point them out.

A Bit of History

Up until now, multimedia content on the Web has, for the most part, been placed in web pages by means of third-party plugins or applications that integrate with the

web browser. Some examples of such software include QuickTime, RealPlayer, and Silverlight.

By far the most popular way to embed video and audio on web pages is by means of Adobe's Flash Player plugin. The Flash Player plugin was originally developed by Macromedia and is now maintained by Adobe as a result of their 2005 buy-out of the company. The plugin has been available since the mid-90s, but did not really take off as a way to serve video content until well into the 2000s.

Before HTML5, there was no standard way to embed video into web pages. A plugin like Adobe's Flash Player is controlled solely by Adobe, and is not open to community development.

The introduction of the `video` and `audio` elements in HTML5 resolves this problem and makes multimedia a seamless part of a web page, the same as the `img` element. With HTML5, there's no need for the user to download third-party software to view your content, and the video or audio player is easily accessible via scripting.

The Current State of Play

Unfortunately, as sublime as HTML5 video and audio sounds in theory, it's less simple in practice. A number of factors need to be considered before you decide to include HTML5's new multimedia elements on your pages.

First, you'll need to understand the state of browser support. At the time of writing, the only browsers with a significant market share that don't support native HTML5 video and audio are Internet Explorer 8 and earlier. Unfortunately, this is still a sizable slice of most sites' audiences.

The other major browser makers offer HTML5 video support in versions now in wide use (Chrome 3+, Safari 4+, and Firefox 3.5+). The last version of Chrome without HTML5 video support (version 2) has a nonexistent market share, and the same is true for the nonsupporting versions of Safari and Opera.

Although IE's market share is significant, you can still use HTML5 video on your pages today. Later on, we'll show you how the new `video` element has been designed with backwards compatibility in mind, so that users of nonsupporting browsers will still have access to your multimedia content.

Video Container Formats

Video on the Web is based on container formats and codecs. A **container** is a wrapper that stores all the necessary data that comprises the video file being accessed, much like a ZIP file wraps or contains files. Some examples of well-known video containers include Flash Video (**.flv**), MPEG-4 (**.mp4** or **.m4v**), and AVI (**.avi**).

The video container houses data, including a video track, an audio track with markers that help synchronize the audio and video, language information, and other bits of metadata that describe the content.

The video container formats relevant to HTML5 are MPEG-4, Ogg, and WebM.

Video Codecs

A video **codec** defines an algorithm for encoding and decoding a multimedia data stream. A codec can encode a data stream for transmission, storage, or encryption, or it can decode it for playback or editing. For the purpose of HTML5 video, we're concerned with the decoding and playback of a video stream. The video codecs that are most pertinent to HTML5 video are H.264, Theora, and VP8.

Audio Codecs

An audio codec in theory works the same as a video codec, except it's concerned with the streaming of sound, rather than video frames. The audio codecs that are most pertinent to HTML5 video are AAC and Vorbis.

What combinations work in current browsers?

It would be nice if browser support allowed us to choose a single container, video codec, and audio codec to create a standard way of embedding video using the new **video** element in HTML5. Unfortunately, it's not quite that simple—although things are improving.

In Table 5.1, we've outlined video container and codec support in the most popular browser versions. This chart only includes browser versions that offer support for the HTML5 video element.

Table 5.1. Browser support for HTML5 video

Container/Video Codec/Audio Codec	Firefox	Chrome	IE	Opera	Safari	iOS Safari	Android
Ogg/Theora/Vorbis	3.5+	3+	—	10.5+	—	—	—
MP4/H.264/AAC	—	3–11	9+	—	4+	4+	2.1+ ^a
WebM/VP8/Vorbis	4+	6+	9+ ^b	10.6+	—	—	2.3+

^a Versions of Android prior to 2.3 require JavaScript to play the video.

^b IE9 supports playback of WebM video with a VP8 codec when the user has installed a VP8 codec on Windows.

Opera Mini and Opera Mobile currently offer no support for HTML5 video, but Opera has announced there are plans to include support in upcoming releases.¹



Licensing Issues

While the new **video** element itself is free to use in any context, the containers and codecs are not always as simple. For example, while the Theora and VP8 (WebM) codecs are not patent-encumbered, the H.264 codec is patent-encumbered and licensing for it is provided by the MPEG-LA group.

Currently, for H.264, if your video is provided to your users for free, there's no requirement for you to pay royalties. However, detailed licensing issues are far beyond the scope and intent of this book, so just be aware that you may need to do some research before using any particular video format when including HTML5 video in your pages.

The Markup

After all that necessary business surrounding containers, codecs, browser support, and licensing issues, it's time to examine the markup of the **video** element and its associated attributes.

The simplest way to include HTML5 video in a web page is as follows: `<video src="example.webm"></video>`

¹ <http://my.opera.com/operamobile/blog/2010/12/04/developing-opera-mobile-for-android/>

But, as you’ve probably figured out from the preceding sections, this will only work in a limited number of browsers. It is, however, the minimum code required to have HTML5 video working to some extent. In a perfect world, it would work everywhere—the same way the `img` element works everywhere—but that’s a little way off just yet.

Similar to the `img` element, the `video` element should also include `width` and `height` attributes:

```
<video src="example.webm" width="375" height="280"></video>
```

Even though the dimensions can be set in the markup, they’ll have no effect on the aspect ratio of the video. For example, if the video in the above example was actually 375×240 and the markup was as shown above, the video would be centered vertically inside the 280-pixel space specified in the HTML. This stops the video from stretching unnecessarily and looking distorted.

The `width` and `height` attributes accept integers only, and their values are always in pixels. Naturally, these values can be overridden via scripting or CSS.

Enabling Native Controls

No embedded video would be complete without giving the user the ability to play, pause, stop, seek through the video, or adjust the volume. HTML5’s `video` element includes a `controls` attribute that does just that:

```
<video src="example.webm" width="375" height="280" controls></video>
```

`controls` is a Boolean attribute, so no value is required. Its inclusion in the markup tells the browser to make the controls visible and accessible to the user.

Each browser is responsible for the look of the built-in video controls. Figure 5.1 to Figure 5.4 show how these controls differ in appearance from browser to browser.



Figure 5.1. The native video controls in Firefox 4

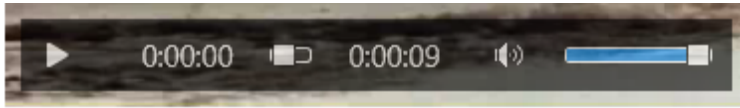


Figure 5.2. ... in IE9



Figure 5.3. ... in Opera 11



Figure 5.4. ... and in Chrome

The autoplay Attribute

We'd love to omit reference to this particular attribute, since its use will be undesirable for the most part. However, there are cases where it can be appropriate. The Boolean `autoplay` attribute does exactly what it says: it tells the web page to play the video as soon as possible.

Normally, this is a bad practice; most of us know too well how jarring it can be if a website starts playing video or audio as soon as it loads—especially if our speakers are turned up. Usability best practices dictate that sounds and movement on web pages should only be triggered when requested by the user. But this doesn't mean that the `autoplay` attribute can never be used.

For example, if the page in question contains nothing but a video—that is, the user clicked on a link to a page for the sole purpose of viewing a specific video—it may be acceptable to allow it to play automatically, depending on the video's size, any surrounding content, and the audience.

Here's how you'd do that:

```
<video src="example.webm" width="375" height="280" controls
➔autoplay></video>
```



Autoplaying on the iPhone

Safari on the iPhone will ignore the `autoplay` attribute; all video will wait for the user to press the play button before starting. This is sensible, given that mobile bandwidth is often limited.

The loop Attribute

Another attribute that you should think twice about before using is the Boolean `loop` attribute. Again, it's fairly self-explanatory: according to the spec, this attribute, when present, will tell the browser to “seek back to the start of the media resource upon reaching the end.”

So if you created a web page whose sole intention was to annoy its visitors, it might contain code like this:

```
<video src="example.webm" width="375" height="280" controls  
  ↪autoplay loop></video>
```

Autoplay and an infinite loop! We just need to remove the native controls and we'd have a trifecta of worst practices.

Of course, there are some situations where `loop` can be useful: imagine a browser-based game, in which ambient sounds and music should play continuously as long as the page is open.

The preload Attribute

In contrast to the two previous attributes, `preload` could definitely come in handy in a number of cases. The `preload` attribute accepts one of three values:

auto

A value of `auto` indicates that the video and its associated metadata will start loading before the video is played. This way, the browser can start playing the video more quickly when the user requests it.

none

A value of `none` indicates that the video shouldn't load in the background before the user presses play.

metadata

This works like `none`, except that any metadata associated with the video (for example, its dimensions, duration, and the like) can be preloaded, even though the video itself won't be.

This particular attribute does not have a spec-defined default in cases where it's omitted; each browser decides which of those three values should be the default state, which makes sense. It allows desktop browsers to preload the video and/or metadata automatically (having no real adverse effect) while permitting mobile browsers to default to either `metadata` or `none`, as many mobile users have restricted bandwidth and will prefer to have the choice of downloading the video or not.

The poster Attribute

When you go to view a video on the Web, normally a single frame of the video will display in order to provide a teaser of its content. The `poster` attribute makes it easy to choose such a teaser. This attribute, similar to `src`, will point to an image file on the server by means of a URL.

Here's how our video element would look with a `poster` attribute defined:

```
<video src="example.webm" width="375" height="280"  
➔poster="teaser.jpg" controls></video>
```

Although the `poster` attribute is useful, there's a bug in iOS3 (corrected in iOS4) that prevents playback of the video if this attribute is present. If you know that many of your visitors use iOS 3.x, you should either avoid using the `poster` attribute, or remove it for those devices specifically.

The audio Attribute

The `audio` attribute controls the default state of the audio track for the video element, and currently accepts only a single possible value: `muted`. The spec states that other values are likely to be added in the future, for specifying the default audio track or volume, for example.

A value of `muted` will cause the video's audio track to default to muted, potentially overriding any user preferences. This will only control the default state of the element—the user interacting with the controls, or JavaScript can change this.

Here it is added to our video element:

```
<video src="example.webm" width="375" height="280"
  ▶poster="teaser.jpg" audio="muted"></video>
```

Adding Support for Multiple Video Formats

As we discussed earlier, using a single container format to serve your video is not currently an option, even though that's really the ultimate idea behind having the video element, and one which we hope will be realized in the future. To allow inclusion of multiple video formats, the video element allows source elements to be defined so that you can allow every user agent to display the video using the format of its choice. These elements serve the same function as the src attribute on the video element; so if you're providing source elements, there's no need to specify a src for your video.

Taking current browser support into consideration, here's how we might declare our source elements:

```
<source src="example.mp4" type="video/mp4">
<source src="example.webm" type="video/webm">
<source src="example.ogv" type="video/ogg">
```

The source element (oddly enough) takes a src attribute that specifies the location of the video file. It also accepts a type attribute that specifies the container format for the resource being requested. This latter attribute allows the browser to determine if it can play the file in question, thus preventing the browser from unnecessarily downloading an unsupported format.

The type attribute also allows a codec parameter to be specified, which defines the video and audio codecs for the requested file. Here's how our source elements will look with the codecs specified:

```
<source src="example.mp4"
  ▶type='video/mp4; codecs="avc1.42E01E, mp4a.40.2"'>
<source src="example.webm" type='video/webm; codecs="vp8, vorbis"'>
<source src="example.ogv" type='video/ogg; codecs="theora, vorbis"'>
```

You'll notice that the syntax for the `type` attribute has been slightly modified to accommodate the container and codec values. The double quotes surrounding the values have been changed to single quotes, and another set of nested double quotes is included specifically for the codecs.

This can be a tad confusing at first glance, but in most cases you'll just be copying and pasting those values once you have a set method for encoding the videos (which we'll touch on later in this chapter). The important point is that you define the correct values for the specified file to ensure that the browser can determine which (if any) file it will be able to play.

source Order

In our example above, the MP4/H.264/AAC container/codecs combination is included first. This is to ensure that the video will play on the iPad. On that device, a bug causes only the first source element to be recognized. It's safe to assume that this bug is fixed in subsequent versions of the iPad, but for now it's necessary to include the MP4/H.264 file first to ensure compatibility.

The first source element will be recognized by IE9, Safari, and older versions of Chrome, so that covers quite a large chunk of our HTML5-ready audience.

The next element in the list defines the WebM/VP8/Vorbis container/codecs combination. This is supported by later versions of Chrome that will eventually drop support for H.264. In addition to Chrome, WebM video will also play in Firefox 4 and Opera 10.6.

Finally, the third source element declares the Ogg/Theora/Vorbis container/codecs combination, which is supported by Firefox 3.5 and Opera 10.5. Although other browsers also support this combination, they'll be using the other formats since they appear ahead of this one in the source order. The browsers that support only this combination are older versions of browsers whose current versions support other formats, so it will be possible to drop this format once those versions become sufficiently rare.

These three source elements are placed as children of the `video` element, so with our three file formats declared, our code will now look like this:

index.html (excerpt)

```
<video width="375" height="280" poster="teaser.jpg" audio="muted">
  <source src="example.mp4" type='video/mp4;
  ➤codecs="avc1.42E01E, mp4a.40.2"'>
  <source src="example.webm" type='video/webm;
  ➤codecs="vp8, vorbis"'>
  <source src="example.ogv" type='video/ogg;
  ➤codecs="theora, vorbis"'>
</video>
```

You'll notice that our code above is now without the `src` attribute on the `video` element. As well as being redundant, it would also override any video files defined in the source elements.

What about Internet Explorer 6–8?

The three source elements that we included inside our `video` element will cover all modern browsers. But we're yet to ensure that our video will play for a potentially large portion of our audience. As mentioned earlier, a significant percentage of users are still using browsers without native support for HTML5 video. Most of those users are on some version of Internet Explorer prior to version 9.

In keeping with the principle of graceful degradation, the HTML5 `video` element has been designed so that older browsers can access the video by some alternate means. Older browsers that fail to recognize the `video` element will simply ignore it, along with its source children. But if the `video` element contains content that the browser recognizes as valid HTML, it will read and display that content instead.

What kind of content can we serve to those nonsupporting browsers? According to Adobe,² 99% of users have the Flash plugin installed on their systems. Additionally, most of those instances of the Flash plugin are version 9 or later, which offer support for the MPEG-4 video container format. To allow Internet Explorer 6 to 8 (and other older browsers lacking support for HTML5 video) to play the video, we declare an embedded Flash video to use as a fallback. Here's the completed code for the video on *The HTML5 Herald*, with the Flash fallback code included:

² http://www.adobe.com/products/player_census/flashplayer/version_penetration.html

```

<video width="375" height="280" poster="teaser.jpg" audio="muted">
  <source src="example.mp4" type='video/mp4;
  ↳codecs="avc1.42E01E, mp4a.40.2"'>
  <source src="example.webm" type='video/webm;
  ↳codecs="vp8, vorbis"'>
  <source src="example.ogv" type='video/ogg;
  ↳codecs="theora, vorbis"'>
  <!-- fallback to Flash: -->
  <object width="375" height="280" type="application/x-shockwave-
  ↳flash" data="mediaplayer-5.5/player.swf">
    <param name="movie" value="mediaplayer-5.5/player.swf">
    <param name="allowFullScreen" value="true">
    <param name="wmode" value="transparent">
    <param name="flashvars" value="controlbar=over&image=
  ↳images/teaser.jpg&file=example.mp4">
    <!-- fallback image -->
    
  </object>
</video>

```

We'll avoid going into a detailed discussion of how this newly added code works (this isn't a Flash book, after all!), but here are a few points to note about this addition to our markup:

- The `width` and `height` attributes on the `object` element should be the same as those on the `video` element.
- We're using the open source JW Player by LongTail Video³ to play the file; you can use whichever video player you prefer.
- The Flash video code has a fallback of its own—an image file that displays if the code for the Flash video fails to work.
- The fourth `param` element defines the file to be used (**example.mp4**); as mentioned, most instances of the Flash player now support video playback using the MPEG-4 container format, so there's no need to encode another video format.

³ <http://www.longtailvideo.com/players/jw-flv-player/>

- HTML5-enabled browsers that support HTML5 video are instructed by the spec to ignore any content inside the `video` element that's not a source tag, so the fallback is safe in all browsers.

The last thing we'll mention here is that, in addition to the Flash fallback content, you could also provide an optional “download video” link that allows the user to get a local copy of the video and view it at their leisure. This would ensure that nobody is left without a means to view the video.

MIME Types

If you find that you've followed our instructions closely and your videos still won't play from your server, the issue could be related to the content-type information being sent.

Content-type, also known as the MIME type, tells the browser what kind of content they're looking at. Is this a text file? If so, what kind? HTML? JavaScript? Is this a video file? The content-type answers these questions for the browser. Every time your browser requests a page, the server sends “headers” to your browser before sending any files. These headers tell your browser how to interpret the file that follows. Content-type is an example of one of the headers the server sends to the browser.

The MIME type for each video file that you include via the `source` element is the same as the value of the `type` attribute (minus any codec information). For the purpose of HTML5 video, we're concerned with three MIME types. To ensure that your server is able to play all three types of video file, place the following lines of code in your `.htaccess` file (or the equivalent if you're using a web server other than Apache):

```
AddType video/ogg .ogg
AddType video/mp4 .mp4
AddType video/webm .webm
```

If this fails to fix your problem, you may have to talk to your host or server administrator to find out if your server is using the correct MIME types. To learn more

about configuring other types of web servers, read the excellent article “Properly Configuring Server MIME Types” from the Mozilla Developer Network.⁴



What's an .htaccess file?

An **.htaccess** file provides a way to make configuration changes on a per-directory basis when using the Apache web server. The directives in an **.htaccess** file apply to the directory it lives in and all subdirectories. For more on **.htaccess** files, see the Apache documentation.⁵

Encoding Video Files for Use on the Web

The code we've presented for *The HTML5 Herald* is virtually bullet-proof, and will enable the video to be viewable by nearly everyone that sees the page. Because we need to encode our video in at least two formats (possibly three, if we want to), we need an easy way to encode our original video file into these HTML5-ready formats. Fortunately, there are some online resources and desktop applications that allow you to do exactly that.

Miro Video Converter⁶ is free software with a super-simple interface that offers the ability to encode your video into all the necessary formats for HTML5 video. It's available for Mac and Windows.⁷

Simply drag a file to the window, or browse for a file in the customary way. A drop-down box offers options for encoding your video in Theora, WebM, or MPEG-4 format. There's also an option for MP3 and a number of presets for device-specific video output.

There are a number of other options for encoding HTML5 video, but this should suffice to help you create the two (or three) files necessary for embedding video that 99% of users can view.

⁴ https://developer.mozilla.org/en/properly_configuring_server_mime_types

⁵ <http://httpd.apache.org/docs/1.3/howto/htaccess.html>

⁶ <http://www.mirovideoconverter.com/>

⁷ Linux users can use FFmpeg, [<http://ffmpeg.org/>] the command-line utility on which Miro Video Converter is based.

Creating Custom Controls

There's another huge benefit to using HTML5 video compared to the customary method of embedding video with a third-party technology. With HTML5 video, the `video` element becomes a real part of the web page, rather than an inaccessible plugin. It's as much a part of the web page as an `img` element or any other native HTML element would be. This means we can target the `video` element and its various components using JavaScript—and we can even style the `video` element with CSS.

As previously mentioned, each browser that supports HTML5 video embeds a native set of controls to help the user access the video content. These controls have a different appearance in each browser, which may vex those concerned with a site's branding. No problem: by using the JavaScript API provided by the `video` element, we can create our own custom controls and link them up to the video's behavior.

Custom controls are created using whichever elements you want—images, plain HTML and CSS, or even elements drawn using the Canvas API—the choice is yours. To harness this API, create your own custom controls, insert them into the page, and then use JavaScript to convert those otherwise static graphic elements into dynamic, fully functioning video controls.

Some Markup and Styling to Get Us Started

For our sample site, we're going to build a very simple set of video controls to demonstrate the power of the new HTML5 video API. To start off, Figure 5.5 shows a screenshot of the set of controls we'll be using to manipulate the video.

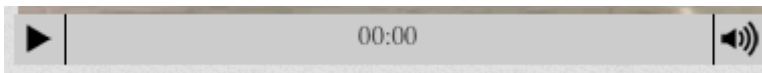


Figure 5.5. The simple set of video controls we'll be building

Both of those buttons have alternate states: Figure 5.6 shows how the controls will look if the video is playing and the sound has been muted.

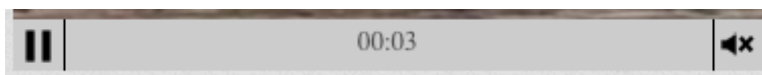


Figure 5.6. Our controls again, this time with the sound muted and the video playing

Our controls have three components:

- play/pause button
- timer that counts forward from zero
- mute/unmute button

In most cases, your custom video controls should have all the features of the default controls that various browsers natively provide. If your set of controls introduces fewer or inferior features, it's likely you'll end up frustrating your users.

For the purpose of introducing the API, rather than trying to mimic what the browsers natively do, we want to introduce the important parts of the video API gradually; this will allow you to get your feet wet while establishing a foundation from which to work.

We'll be creating a very simple, yet usable, set of controls for our video. The main feature missing from our set of controls is the seek bar that lets the user “scrub” through the video to find a specific part. This means there will be no way to go back to the start of the video aside from refreshing the page. Other than that, the controls will function adequately—they'll allow the user to play, pause, mute, or unmute the video.

Here's the HTML we'll be using to represent the different parts of the video controls:

[index.html](#) (*excerpt*)

```
<div id="controls" class="hidden">
  <a id="playPause">Play/Pause</a>
  <span id="timer">00:00</span>
  <a id="muteUnmute">Mute/Unmute</a>
</div>
```

We'll avoid going into the CSS in great detail, but here's a summary of what we've done (you can view the demo page's source in the code archive if you want to see how it's all put together):

- The text in the play/pause and mute/unmute buttons is removed from view using the `text-indent` property.
- A single CSS sprite image is used as a background image to represent the different button states (play, pause, mute, unmute).

- CSS classes are being used to represent the different states; those classes will be added and removed using JavaScript.
- The controls wrapper element is absolutely positioned and placed to overlay the bottom of the video.
- We've given the controls a default opacity level of 50%, but on hover the opacity increases to 100% (we'll be talking more about opacity in Chapter 6).
- By default, the controls wrapper element is set to `display: none` using a class of `hidden`, which we'll remove with JavaScript.

If you're following along building the example, go ahead and style the three elements however you like. The appearance of the controls is really secondary to what we're accomplishing here, so feel free to fiddle until you have a look you're happy with.

Introducing the Media Elements API

Let's go through the steps needed to create our custom controls, and in the process we'll introduce you to some aspects of the video API. Afterwards, we'll summarize some other methods and attributes from the API that we won't be using in our controls, so you can have a good overview of what the API includes.

In order to work with our new custom controls, we'll first cache them by placing them into JavaScript variables. Here are the first few lines of our code:

js/videoControls.js (excerpt)

```
var videoEl = $('video')[0],
    playPauseBtn = $('#playPause'),
    vidControls = $('#controls'),
    muteBtn = $('#muteUnmute'),
    timeHolder = $('#timer');
```

Of course, caching our selections isn't necessary, but it's always best practice (for maintainability and performance) to work with cached objects, rather than needlessly repeating the same code to target various elements on the page.

The first line is targeting the `video` element itself. We'll be using this `videoEl` variable quite a bit when using the API—since most API methods need to be called from the media element. The next four lines of code should be fairly familiar to you

if you took note of the HTML that comprises our controls. Those are the four elements on the page that we'll be manipulating based on user interaction.

Our first task is make sure the native controls are hidden. We could do this easily by simply removing the `controls` attribute from the HTML. But since our custom controls are dependent on JavaScript, visitors with JavaScript disabled would be deprived of any way of controlling the video. So we're going to remove the `controls` attribute in our JavaScript, like this:

js/videoControls.js (excerpt)

```
videoEl.removeAttribute("controls");
```

The next step is to make our own custom controls visible. As mentioned earlier, we've used CSS to remove our controls from view by default. By using JavaScript to enable the visibility of the custom controls, we ensure that the user will never see two sets of controls.

So our next chunk of code will look like this:

js/videoControls.js (excerpt)

```
videoEl.addEventListener('canplaythrough', function () {  
  vidControls.removeClass("hidden");  
}, false);
```

This is the first place we've used a feature from the HTML5 video API. First, take note of the `addEventListener` method. This method does exactly what its name implies: it listens for the specified event occurring on the targeted element.



But `addEventListener` isn't cross-browser!

If you're familiar with cross-browser JavaScript techniques, you probably know that the `addEventListener` method isn't cross-browser. In this case, it poses no problem. The only browsers in use that lack support for `addEventListener` are versions of Internet Explorer prior to version 9—and those browsers have no support for HTML5 video anyway.

All we have to do is use Modernizr (or some equivalent JavaScript) to detect support for the HTML5 video API, and then only run the code for supporting browsers—all of which will support `addEventListener`.

In this case, we’re targeting the `video` element itself. The event we’re registering to be listened for is the `canplaythrough` event from the video API. According to the definition of this event in the spec:⁸

The user agent estimates that if playback were to be started now, the media resource could be rendered at the current playback rate all the way to its end without having to stop for further buffering.

There are other events we can use to check if the video is ready, each of which has its own specific purpose. We’ll touch on some of those other events later in this chapter. This particular one ensures continuous playback, so it’s a good fit for us as we’d like to avoid choppy playback.

Playing and Pausing the Video

When the `canplaythrough` event fires, a callback function is run. In that function, we’ve put a single line of code that removes the `hidden` class from the controls wrapper, so now our controls are visible. Now we want to add some functionality to our controls. Let’s bind a `click` event handler to our play/pause button:

js/videoControls.js (excerpt)

```
playPauseBtn.bind('click', function () {  
  if (videoEl.paused) {  
    videoEl.play();  
  } else {  
    videoEl.pause();  
  }  
});
```

When the button is clicked, we run an `if/else` block that’s using three additional features from the video API. Here’s a description of all three:

The `paused` attribute is being accessed to see if the video is currently in the “paused” state. This doesn’t necessarily mean the video has been paused by the user; it could equally just represent the start of the video, before it’s been played. So this attribute will return `true` if the video isn’t currently playing.

⁸ <http://www.whatwg.org/specs/web-apps/current-work/multipage/video.html#event-media-canplay-through>

Since we’ve now determined that the play/pause button has been clicked, and the video is not currently playing, we can safely call the `play()` method on the video element. This will play the video from its last paused location.

Finally, if the `paused` attribute doesn’t return `true`, the `else` portion of our code will fire, and this will trigger the `pause()` method on the video element, stopping the video.

You may have noticed that our custom controls have no “stop” button (customarily represented by a square icon). You could add such a button if you feel it’s necessary, but many video players don’t use it since the seek bar can be used to move to the beginning of the video. The only catch is that the video API has no “stop” method; to counter this, you can cause the video to mimic the traditional “stop” behavior by pausing it and then sending it to the beginning (more on this later).

You’ll notice that something’s missing from our `if/else` construct. Earlier, we showed you a couple of screenshots displaying the controls in their two states. We need to use JavaScript to alter the background position of our sprite image; we want to change the button from “play me” to “pause me.”

Here’s how we’ll do that:

`js/videoControls.js` (excerpt)

```
videoEl.addEventListener('play', function () {
  playPauseBtn.addClass("playing");
}, false);
videoEl.addEventListener('pause', function () {
  playPauseBtn.removeClass("playing");
}, false);
```

Here we have two more uses of the `addEventListener` method (you’ll need to get used to it if you’re going to use the video and audio APIs!). The first block is listening for play events. So if the click handler we wrote triggers the `play()` method (or if something else causes the video to play, such as some other code on the page), the play event will be detected by the listener and the callback function will run.

The same thing is happening in the second block of code, except that it’s listening for the pause event (not to be confused with the `paused` attribute).

If the element has been played, the first block will add the `class playing` to our play/pause button. This `class` will change the background position of the sprite on the play/pause button to make the “pause me” icon appear. Similarly, the second block of code will remove the `playing class`, causing the state of the button to go back to the default (the “play me” state).

You’re probably thinking, “why not just add or remove the `playing class` in the code handling the button click?” While this would work just fine for when the button is clicked (or accessed via the keyboard), there’s another behavior we need to consider here, demonstrated in Figure 5.7.

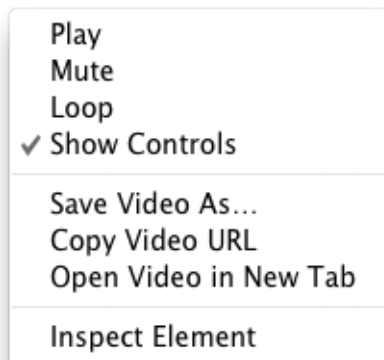


Figure 5.7. Some video controls are accessible via the context menu

The menu above appears when you bring up the `video` element’s context menu. As you can see, clicking the controls on the `video` element isn’t the only way to play/pause or mute/unmute the video.

To ensure that the button states are changed no matter how the `video` element’s features are accessed, we instead listen for `play` and `pause` events (and, as you’ll see in a moment, sound-related events) to change the states of the buttons.



Disabling the Context Menu

You may also be concerned that the `video` element's context menu has an option for **Save video as....** There's been discussion online about how easy it is to save HTML5 video, and this could affect how copyrighted videos will be distributed. Some content producers might feel like avoiding HTML5 video for this reason alone.

Whatever you choose to do, just recognize the realities associated with web video. Most users who are intent on copying and distributing copyrighted video will find ways to do it, regardless of any protection put in place. There are many web apps and software tools that can easily rip even Flash-based video. You should also be aware that even if you do disable the context menu on the `video` element, the user can still view the source of the page and find the location of the video file(s).

Some sites, like YouTube, have already implemented features to combat this when using HTML5 video. YouTube has a page that allows you to opt in to their HTML5 video trial.⁹ After opting in, when you view a video and open the `video` element's context menu, there's a custom context menu. The **"Save Video As..."** option is still present. But not so fast! If you choose this option, (as of this writing) you'll be "rickrolled."¹⁰ Sneaky!

YouTube also dynamically adds the `video` element to the page, so that you're unable to find the URL to the video file by poking around in the source.

So, realize that you do have options, and that it's possible to make it more difficult (but not impossible) for users to rip your copyrighted videos. But also recognize there are drawbacks to changing user expectations, in addition to the performance and maintainability issues associated with convoluting your scripts and markup for what could be little, if any, gain.

Muting and Unmuting the Video's Audio Track

The next bit of functionality we want to add to our script is the mute/unmute button. This piece of code is virtually the same as what was used for the play/pause button. This time, we've bound the `click` event to the mute/unmute button, following with a similar if/else construct:

⁹ <http://www.youtube.com/html5>

¹⁰ <http://en.wikipedia.org/wiki/Rickrolling>

js/videoControls.js (excerpt)

```
muteBtn.bind('click', function () {  
    if (videoEl.muted) {  
        videoEl.muted = false;  
    } else {  
        videoEl.muted = true;  
    }  
});
```

This block of code introduces a new part of the API: the `muted` attribute. After the mute button is clicked, we check to see the status of this attribute. If it's `true` (meaning the sound is muted), we set it to `false` (which unmutes the sound); if it's `false`, we set its status to `true`.

Again, we haven't done any button state handling here, for the same reasons mentioned earlier when discussing the play/pause buttons; the context menu allows for muting and unmuting, so we want to change the mute button's state depending on the actual muting or unmuting of the video, rather than the clicking of the button.

But unlike the play/pause button, we don't have the ability to listen for mute and unmute events. Instead, the API offers the `volumechange` event:

js/videoControls.js (excerpt)

```
videoEl.addEventListener('volumechange', function () {  
    if (videoEl.muted) {  
        muteBtn.addClass("muted");  
    } else {  
        muteBtn.removeClass("muted");  
    }  
}, false);
```

Again, we're using an event listener to run some code each time the specified event (in this case a change in volume) takes place. As you can probably infer from the name of this event, the `volumechange` event isn't limited to detecting muting and unmuting; it can detect volume changes.

Once we have detected the change in volume, we check the status of the video element's `muted` attribute, and we change the `class` on the mute/unmute button accordingly.

Responding When the Video Ends Playback

The code we've written so far will allow the user to play and pause the video, as well as mute and unmute the sound. All of this is done using our custom controls.

At this point, if you let the video play to the end, it will stop on the last frame. We think it's best to send the video back to the first frame, ready to be played again. This gives us the opportunity to introduce two new features of the API:

js/videoControls.js (excerpt)

```
videoEl.addEventListener('ended', function () {  
  videoEl.currentTime = 0;  
  videoEl.pause();  
}, false);
```

This block of code listens for the `ended` event, which tells us that the video has reached its end and stopped. Once we detect this event, we set the video's `currentTime` property to zero. This property represents the current playback position, expressed in seconds (with decimal fractions).

Which brings us to the next step in our code.

Updating the Time as the Video Plays

Now for the last step: we want our timer to update the current playback time as the video plays. We've already introduced the `currentTime` property; we can use it to update the content of our `#timeHolder` element. Here's how we do it:

js/videoControls.js (excerpt)

```
videoEl.addEventListener('timeupdate', function () {  
  timeHolder[0].innerHTML = secondsToTime(videoEl.currentTime);  
}, false);
```

In this case, we're listening for `timeupdate` events. The `timeupdate` event fires each time the video's time changes, which means even a fraction of a second's change will fire this event.

This alone would suffice to create a bare-bones timer. Unfortunately, the time would be unhelpful, and ugly on the eye because you'd see the time changing every millisecond to numerous decimal places, as shown in Figure 5.8.

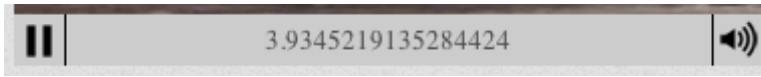


Figure 5.8. Using the `currentTime` property directly in our HTML is less than ideal

In addition, the timer in this state will not display minutes or hours, just seconds—which could end up being in the hundreds or thousands, depending on the length of the video. That's impractical, to say the least.

To format the seconds into a more user-friendly time, we've written a function called `secondsToTime()`, and called it from our `timeupdate` handler above. We don't want to show the milliseconds in this case, so our function rounds the timer to the nearest second. Here's the start of our function:

js/videoControls.js (excerpt)

```
var h = Math.floor(s / (60 * 60)),
    dm = s % (60 * 60),
    m = Math.floor(dm / 60),
    ds = dm % 60,
    secs = Math.ceil(ds);
```

After those five lines of code, the final variable `secs` will hold a rounded number of seconds, calculated from the number of seconds passed into the function.

Next, we need to ensure that a single digit amount of seconds or minutes is expressed using 05 instead of just 5. The next code block will take care of this:

js/videoControls.js (excerpt)

```
if (secs === 60) {
    secs = 0;
    m = m + 1;
}

if (secs < 10) {
    secs = "0" + secs;
}
```

```

if (m === 60) {
    m = 0;
    h = h + 1;
}

if (m < 10) {
    m = "0" + m;
}

```

Finally, we return a string that represents the current time of the video in its correct format:

[js/videoControls.js \(excerpt\)](#)

```

if (h === 0) {
    fulltime = m + ":" + secs;
} else {
    fulltime = h + ":" + m + ":" + secs;
}

return fulltime;

```

The `if/else` construct is included to check if the video is one hour or longer; if so, we'll format the time with two colons. Otherwise, the formatted time will use a single colon that divides minutes from seconds, which will be the case in most circumstances.

Remember where we're running this function. We've included this inside our `timeupdate` event handler. The function's returned result will become the content of the `timeHolder` element (which is the cached element with an `id` of `timer`):

[js/videoControls.js \(excerpt\)](#)

```
timeHolder[0].innerHTML = secondsToTime(videoEl.currentTime);
```

Because the `timeupdate` event is triggered with every fraction of a second's change, the content of the `timeHolder` element will change rapidly. But because we're rounding the value to the nearest second, the *visible* changes will be limited to a time update every second, even though technically the content of the timer element is changing more rapidly.

And that's it, our custom controls are done. The buttons work as expected and the timer runs smoothly. As we mentioned at the top, this isn't quite a fully functional set of controls. But you should at least have a good handle on the basics of interacting with HTML5 video from JavaScript, so have a tinker and see what else you can add.

Further Features of the Media Elements API

The API has much more to it than what we've covered here. Here's a summary of some events and attributes that you might want to use when building your own custom controls, or when working with video and audio elements.

One point to remember is that these API methods and properties can be used anywhere in your JavaScript—they don't need to be linked to custom controls. If you'd like to play a video when the mouse hovers over it, or use audio elements to play various sounds associated with your web application or game, all you need to do is call the appropriate methods.

Events

We've already seen the `canplaythrough`, `play`, `pause`, `volumechange`, `ended`, and `timeupdate` events. Here are some of the other events available to you when working with HTML5 video and audio:

canplay

This is similar to `canplaythrough`, but will fire as soon as the video is playable, even if it's just a few frames. (This contrasts with `canplaythrough`, as you'll remember, which only fires if the browser thinks it can play the video all the way to the end without rebuffering.)

error

This event is sent when an error has occurred; there's also an `error` attribute.

loadeddata

The first frame of the media has loaded.

loadedmetadata

This event is sent when the media's metadata has finished loading. This would include dimensions, duration, and any text tracks (for captions).

playing

This indicates that the media has begun to play. The difference between `playing` and `play` is that `play` will not be sent if the video loops and begins playing again, whereas `playing` will.

seeking

This is sent when a seek operation begins. It might occur when a user starts to move the seek bar to choose a new part of the video or audio.

seeked

This event fires when a seek operation is completed.

Attributes

In addition to the attributes we've already seen, here's a number of useful ones available to you:

playbackRate

The default playback rate is 1. This can be changed to speed up or slow down playback. This is naturally of practical use if you're creating a fast-forward or rewind button, or a slow-motion or slow-rewind button.

src

As its name implies, this attribute returns the URL that points to the video being played. This only works if you're using the `src` attribute on the `video` element.

currentSrc

This will return the value of the URL pointing to the video file being played, whether it's from the `video` element's `src` attribute or one of the source elements.

readyState

This attribute returns a numeric value from 0 to 4, with each state representing the readiness level of the media element. For example, a value of "1" indicates that the media's metadata is available. A value of "4" is virtually the same as the condition for firing the `canplaythrough` event, meaning the video is ready to play, and won't be interrupted by buffering or loading.

duration

This returns the length of the video in seconds.

buffered

This represents the time ranges of the video that have buffered and are available for the browser to play.

videoWidth, videoHeight

These values return the intrinsic dimensions of the video, the actual width and height as the video was encoded—not what’s declared in the HTML or CSS.

Those values can be accessed through the customary `width` and `height` attributes.

You can also access attributes that are able to be declared directly in the HTML, such as `preload`, `controls`, `autoplay`, `loop`, and `poster`.

What about audio?

Much of what we’ve discussed in relation to HTML5 video and its API also apply to the `audio` element, with the obvious exceptions being those related to visuals.

Similar to the `video` element, the `preload`, `autoplay`, `loop`, and `controls` attributes can be used (or not used!) on the `audio` element.

The `audio` element won’t display anything unless controls are present, but even if the element’s controls are absent, the element is still accessible via scripting. This is useful if you want your site to use sounds that aren’t tied to controls presented to the user. The `audio` element nests `source` tags, similar to `video`, and it will also treat any child element that’s not a `source` tag as fallback content for nonsupporting browsers.

As for codec/format support, Firefox, Opera, and Chrome all support Ogg/Vorbis; Safari, Chrome, and IE9 support MP3; and every supporting browser supports WAV. Safari also supports AIFF. At present, MP3 and Ogg/Vorbis will be enough to cover you for all supporting browsers.

Accessible Media

In addition to their status as first-class citizens of the page, making them intrinsically more keyboard accessible (using `tabindex`, for example), the HTML5 media elements also give you access to the `track` element to display captions or a transcript of the media file being played. Like `source` elements, `track` elements should be placed as children of the `video` or `audio` element.

The `track` element is still in flux, but if included as a child of the `video` element, it would look like the example shown here (similar to an example given in the spec):

```
<video src="brave.webm">
  <track kind="subtitles" src="brave.en.vtt" srclang="en"
  ↳label="English">
  <track kind="captions" src="brave.en.vtt" srclang="en"
  ↳label="English for the Hard of Hearing">
  <track kind="subtitles" src="brave.fr.vtt" srclang="fr"
  ↳label="Français">
  <track kind="subtitles" src="brave.de.vtt" srclang="de"
  ↳label="Deutsch">
</video>
```

The code here has four `track` elements, each referencing a text track for captions in a different language (or, in the case of the second one, alternate content in the same language).

The `kind` attribute can take one of five values: `subtitles`, `captions`, `descriptions`, `chapters`, and `metadata`. The `src` attribute is required, and points to an external file that holds the track information. The `srclang` attribute specifies the language. Finally, the `label` attribute gives a user-readable title for the track.

As of this writing, the `track` element is yet to be supported by any browser. For more info on this new element, see the W3C spec.¹¹

¹¹ <http://dev.w3.org/html5/spec/Overview.html#the-track-element>

It's Showtime

Video and audio on the Web have long been the stronghold of Flash, but, as we've seen, HTML5 is set to change that. While the codec and format landscape is presently fragmented, the promises of fully scriptable multimedia content, along with the performance benefits of running audio and video natively in the browser instead of in a plugin wrapper, are hugely appealing to web designers, developers, and content providers.

Because we have access to nearly foolproof fallback techniques, there's no reason not to start experimenting with these elements now. At the very least, we'll be better prepared when support is more universal.

We've now covered just about everything on HTML5 “proper” (that is, the bits that are in the HTML5 spec). In the next few chapters, we'll turn our attention to CSS3, and start to make *The HTML5 Herald* look downright *fancy*. After that, we'll finish by looking at the new JavaScript APIs that are frequently bundled with the term “HTML5.”

Chapter 6

Introducing CSS3

The content layer is done! Now it's time to make it pretty. The next four chapters focus on presentation. In this one, we'll start by covering some basics: we'll first do a quick overview of CSS selectors, and see what's been added to our arsenal in CSS3. Then, we'll take a look at a few new ways of specifying colors. We'll then dive into rounded corners, drop shadows, and text shadows—tips and tricks enabling us to style pages without having to make dozens of rounded-corner and text images to match our designs.

But first, we need to make sure older browsers recognize the new elements on our page, so that we can style them.

Getting Older Browsers on Board

As we mentioned back in Chapter 1, styling the new HTML5 elements in older versions of Internet Explorer requires a snippet of JavaScript called an HTML5 shiv. If you're using the Modernizr library detailed in Appendix A (which includes a similar piece of code), you'll be fine.

Even with this JavaScript in place, though, you're not quite ready to roll. IE6 through 8 will now be aware of these new elements, but they'll still lack any default styles. In fact, this will be the case for previous versions of other browsers as well; while they may allow arbitrary elements, they've no way of knowing, for example, that `article` should be block-level and `mark` should be inline. Because elements render as inline by default, it makes sense to tell these browsers which elements should be block-level.

This can be done with the following simple CSS rule:

css/styles.css (excerpt)

```
article, aside, figure, footer, header, hgroup, nav, section {  
    display: block;  
}
```

With this CSS and the required JavaScript in place, all browsers will start off on an even footing when it comes to styling HTML5 elements.

CSS3 Selectors

Selectors are at the heart of CSS. Without selectors to target elements on the page, the only way to modify the CSS properties of an element would be to use the element's `style` attribute and declare the styles inline. This, of course, is ugly, awkward, and unmaintainable. So we use selectors. Originally, CSS allowed the matching of elements by type, class, and/or id. This required adding `class` and `id` attributes to our markup to create hooks and differentiate between elements of the same type. CSS2.1 added pseudo-elements, pseudo-classes, and combinators. With CSS3, we can target almost any element on the page with a wide range of selectors.

In the descriptions that follow, we'll be including the selectors provided to us in earlier versions of CSS. They are included because, while we can now start using CSS3 selectors, all the selectors from previous versions of CSS are still supported. Even for those selectors that have been around for quite some time, it's worth going over them here, as browser support for many of them has only just reached the point of making them usable.

Relational Selectors

Relational selectors target elements based on their relationship to another element within the markup. All of these are supported in IE7+, Firefox, Opera, and WebKit:

Descendant (E F)

You should definitely be familiar with this one. The descendant selector targets any element F that is a descendant (child, grandchild, great grandchild, and so on) of an element E. For example, `ol li` targets `li` elements that are inside ordered lists. This would include `li` elements in a `ul` that's nested in an `ol`—which might not be what you want.

Child (E > F)

This selector matches any element F that is a *direct child* of element E—any further nested elements will be ignored. Continuing the above example, `ol > li` would only target `li` elements directly inside the `ol`, and would omit those nested inside a `ul`.

Adjacent Sibling (E + F)

This will match any element F that shares the same parent as E, and comes *directly after* E in the markup. For example, `li + li` will target all `li` elements except the first `li` in a given container.

General Sibling (E ~ F)

This one's a little trickier. It will match any element F that shares the same parent as any E and comes after it in the markup. So, `h1~h2` will match any `h2` that follows an `h1`, as long as they both share the same direct parent—that is, as long as the `h2` is not nested in any other element.

Let's look at a quick example:

```
<article>
  <header>
    <h1>Main title</h1>
    <h2>This subtitle is matched </h2>
  </header>
  <p> blah, blah, blah ...</p>
  <h2>This is not matched by h1~h2, but is by header~h2</h2>
  <p> blah, blah, blah ...</p>
</article>
```

The selector string `h1~h2` will match the first `h2`, because they’re both children, or direct descendants, of the header. The second `h2` doesn’t match, since its parent is `article`, not `header`. It would, however, match `header~h2`. Similarly, `h2~p` only matches the last paragraph, since the first paragraph precedes the `h2` with which it shares the parent `article`.



There Are No Backwards Selectors

You’ll notice that there’s no “parent” or “ancestor” selector, and there’s also no “preceding sibling” selector. This can be annoying sometimes, but there’s a reason for it: if the browser had to go backwards up the DOM tree, or recurse into sets of nested elements before deciding whether or not to apply a style, rendering would be exponentially slower and more demanding in terms of processing. See http://snook.ca/archives/html_and_css/css-parent-selectors for a more in-depth explanation of this issue.

Looking through the stylesheet for *The HTML5 Herald*, you’ll see a number of places where we’ve used these selectors. For example, when determining the overall layout of the site, we want the three-column divs to be floated left. To avoid this style being applied to any other divs nested inside them, we use the child selector:

css/styles.css (excerpt)

```
#main > div {  
  float: left;  
  overflow: hidden;  
}
```

As we add new styles to the site over the course of the next few chapters, you’ll be seeing a lot of these selector types.

Attribute Selectors

CSS2 introduced several attribute selectors. These allow for matching elements based on their attributes. CSS3 expands upon those attribute selectors, allowing for some targeting based on pattern matching.

E[attr]

Matches any element `E` that has the attribute `attr` with any value. We made use of this back in Chapter 4 to style required inputs—`input:required` works in

the latest browsers, but `input[required]` has the same effect and works in some slightly older ones.

E[attr=val]

Matches any element `E` that has the attribute `attr` with the exact, case-insensitive value `val`. While not new, it's helpful in targeting form input types—for instance, target checkboxes with `input[type=checkbox]`.

E[attr|=val]

Matches any element `E` whose attribute `attr` either has the value `val` or begins with `val-`. This is most commonly used for the `lang` attribute (as in `lang="en-us"`). For example, `p[lang|"en"]` would match any paragraph that has been defined as being in English, whether it be UK or US English.

E[attr~=val]

Matches any element `E` whose attribute `attr` has within its value the full word `val`, surrounded by whitespace. For example, `.info[title~=more]` would match any element with the class `info` that had a `title` attribute containing the word “more,” such as “Click here for more information.”

E[attr^=val] (IE8+, WebKit, Opera, Mozilla)

Matches any element `E` whose attribute `attr` starts with the value `val`. In other words, the `val` matches the beginning of the attribute value.

E[attr\$=val] (IE8+, WebKit, Opera, Mozilla)

Matches any element `E` whose attribute `attr` *ends* in `val`. In other words, the `val` matches the end of the attribute value.

E[attr*=val] (IE8+, WebKit, Opera, Mozilla)

Matches any element `E` whose attribute `attr` matches `val` anywhere within the attribute. In other words, the string `val` is matched anywhere in the attribute value. It is similar to `E[attr~=val]` above, except the `val` can be part of a word. Using the same example as above, `.fakelink[title~=info] {}` would match any element with the class `fakelink` that has a `title` attribute containing the string `info`, such as “Click here for more information.”

Pseudo-classes

It's likely that you're already familiar with some of the user interaction pseudo-classes, namely `:link`, `:visited`, `:hover`, `:active`, and `:focus`.



Key Points to Remember

1. The `:visited` pseudo-class may pose a security risk, and may not be fully supported in the future. In short, malicious sites can apply a style to a `visited` link, then use JavaScript to check the styles of links to popular sites. This allows the attacker to glimpse the user's browsing history without their permission. As a result, several browsers have begun limiting the styles that can be applied with `:visited`, and some others (notably Safari 5) have disabled it entirely.

The spec explicitly condones these changes, saying: "UAs [User Agents] may therefore treat all links as unvisited links, or implement other measures to preserve the user's privacy while rendering visited and unvisited links differently."

2. For better accessibility, add `:focus` wherever you include `:hover`, as not all visitors will use a mouse to navigate your site.
3. `:hover` can apply to any element on the page—not just links and form controls.
4. `:focus` and `:active` are relevant to links, form controls, and any element with a `tabindex` attribute.

While it's likely you've been using these basic pseudo-classes for some time, there are many other pseudo-classes available. Several of these have been in the specification for years, but weren't supported (or commonly known) until browsers started supporting the new HTML5 form attributes that made them more relevant.

The following pseudo-classes match elements based on attributes, user interaction, and form control state:

`:enabled`

A user interface element that's enabled.

`:disabled`

Conversely, a user interface element that's disabled.

:checked

Radio buttons or checkboxes that are selected or ticked.

:indeterminate

Form elements that are neither checked nor unchecked. This pseudo-class is still being considered, and may be included in the specification in the future.

:target

This selector singles out the element that is the target of the currently active intrapage anchor. That sounds more complicated than it is: you already know you can have links to anchors within a page by using the # character with the id of the target. For example, you may have `Skip to content` link in your page that, when clicked, will jump to the element with an id of content.

This changes the URL in the address bar to `thispage.html#content`—and the `:target` selector now matches the `#content` element, as if you had included, temporarily, the selector `#content`. We say “temporarily” because as soon as the user clicks on a different anchor, `:target` will match the new target.

:default

Applies to one or more UI elements that are the default among a set of similar elements.

:valid

Applies to elements that are valid, based on the `type` or `pattern` attributes (as we discussed in Chapter 4).

:invalid

Applies to empty required elements, and elements failing to match the requirements defined by the `type` or `pattern` attributes.

:in-range

Applies to elements with range limitations, where the value is within those limitations. This applies, for example, to `number` and `range` input types with `min` and `max` attributes.

:out-of-range

The opposite of `:in-range`: elements whose value is *outside* the limitations of their range.

:required

Applies to form controls that have the `required` attribute set.

:optional

Applies to all form controls that *do not* have the `required` attribute.

:read-only

Applies to elements whose contents are unable to be altered by the user. This is usually most elements other than form fields.

:read-write

Applies to elements whose contents are user-alterable, such as text input fields.

Browser support for these pseudo-classes is uneven, but improving fairly rapidly. Browsers that support form control attributes like `required` and `pattern` also support the associated `:valid` and `:invalid` pseudo-classes.

IE6 fails to understand `:hover` on elements other than links, and neither IE6 nor IE7 understand `:focus`. IE8 and earlier lack support for `:checked`, `:enabled`, `:disabled`, and `:target`. The good news is that IE9 *does* support these selectors.

While support is still lacking, JavaScript libraries such as jQuery can help in targeting these pseudo-classes in nonsupporting browsers.

Structural Pseudo-classes

So far, we've seen how we can target elements based on their attributes and states. CSS3 also enables us to target elements based simply on their location in the markup. These selectors are grouped under the heading structural pseudo-classes.¹

These might seem complicated right now, but they'll make more sense as we look at ways to apply them later on. These selectors are supported in IE9, as well as current and older versions of all the other browsers—but not in IE8 and below.

¹ <http://www.w3.org/TR/css3-selectors/#structural-pseudos>

:root

The root element, which is always the `html` element.

E F:nth-child(n)

The element `F` that is the `n`th child of its parent `E`.

E F:nth-last-child(n)

The element `F` that is the `n`th child of its parent `E`, counting backwards from the last one. `li:nth-last-child(1)` would match the last item in any list—this is the same as `li:last-child` (see below).

E:nth-of-type(n)

The element that is the `n`th element of its type in a given parent element.

E:nth-last-of-type(n)

Like `nth-of-type(n)`, except counting backwards from the last element in a parent.

E:first-child

The element `E` that is the first child `E` of its parent. This is the same as `:nth-child(1)`.

E:last-child

The element `E` that is the last child `E` of its parent, same as `:nth-last-child(1)`.

E:first-of-type

Same as `:nth-of-type(1)`.

E:last-of-type

Same as `:nth-last-of-type(1)`.

E:only-child

An element that's the only child of its parent.

E:only-of-type

An element that's the only one of its type inside its parent element.

E:empty

An element that has no children; this includes text nodes, so `<p>hello</p>` will not be matched.

E:lang(en)

An element in the language denoted by the two-letter abbreviation (en).

E:not(exception)

This is a particularly useful one: it will select elements that *don't* match the selector in the parentheses.

Selectors with the `:not` pseudo-class match everything to the left of the colon, and then exclude from that matched group the elements that also match what's to the right of the colon. The left-hand side matching goes first. For example, `p:not(.copyright)` will match all the paragraphs in a document first, and then exclude all the paragraphs from the set that also have the class of `copyright`. You can string several `:not` pseudo-classes together. `h2:not(header > h2):not(.logo)` will match all h2s on a page except those that are in a header and those that have a class of `logo`.

**What is n?**

There are four pseudo-classes that take an *n* parameter in parentheses: `:nth-child(n)`, `:nth-last-child(n)`, `:nth-of-type(n)`, and `:nth-last-of-type(n)`.

In the simplest case, *n* can be an integer. For example, `:nth-of-type(1)` will target the first element in a series. You can also pass one of the two keywords `odd` or `even`, targeting every other element. You can also, more powerfully, pass a number expression such as `:nth-of-type(3n+1)`. `3n` means every third element, defining the frequency, and `+1` is the offset. The default offset is zero, so where `:nth-of-type(3n)` would match the 3rd, 6th, and 9th elements in a series, `:nth-of-type(3n+1)` would match the 1st, 4th, 7th, and so on. Negative offsets are also allowed.

With these numeric pseudo-classes, you can pinpoint which elements you want to target without adding classes to the markup. The most common example is a table where every other row should be a slightly darker color to make it easier to read. We used to have to add `odd` or `even` classes to every `tr` to accomplish this. Now, we can simply declare `tr:nth-of-type(odd)` to target every odd line without touching the markup. You can even take it a step further with three-colored striped tables: target `:nth-of-type(3n)`, `:nth-of-type(3n+1)`, and `:nth-of-type(3n+2)` and apply a different color to each.

Pseudo-elements and Generated Content

In addition to pseudo-classes, CSS gives us access to **pseudo-elements**. Pseudo-elements allow you to target text that's part of the document, but not otherwise targetable in the document tree. Pseudo-classes generally reflect some attribute or state of the element that is not otherwise easily or reliably detectable in CSS. Pseudo-elements, on the other hand, represent some structure of the document that's outside of the DOM.

For example, all text nodes have a first letter and a first line, but how can you target them without wrapping them in a span? CSS provides the `::first-letter` and `::first-line` pseudo-elements that match the first letter and first line of a text node, respectively. These can alternatively be written with just a single colon: `:first-line` and `:first-letter`.



Why bother with the double colon?

The double colon is the correct syntax, but the single colon is better supported. IE6, IE7, and IE8 only understand the single-colon notation. All other browsers support both. Even though `:first-letter`, `:first-line`, `:first-child`, `:before`, and `:after` have been around since CSS2, these pseudo-elements in CSS3 have been redefined using double colons to differentiate them from pseudo-classes.

Generated Content

The `::before` and `::after` pseudo-elements don't refer to content that exists in the markup, but rather to a location where you can insert additional content, generated right there in your CSS. While this generated content doesn't become part of the DOM, it can be styled.

To generate content for a pseudo-element, use the `content` property. For example, let's say you wanted all external links on your page to be followed by the URL they point to in parentheses, to make it clear to your users that they'll be leaving your page. Rather than hardcoding the URLs into your markup, you can use the combination of an attribute selector and the `::after` pseudo-element:

```
a[href^=http]:after {
  content: " (" attr(href) ")";
}
```

`attr()` allows you to access any attributes of the selected element, coming in handy here for displaying the link’s target. And you’ll remember from the attribute selectors section that `a[href^=http]` means “any a element whose `href` attribute begins with `http`”; in other words, external links.

Here’s another example:

```
a[href$=pdf] {
  background: transparent url(pdficon.gif) 0 50% no-repeat;
  padding-left: 20px;
}
a[href$=pdf]:after {
  content: " (PDF)";
}
```

Those styles will add a PDF icon and the text “(PDF)” after links to PDFs. Remember that the `[attr$=val]` selector matches the *end* of an attribute—so `document.pdf` will match but `page.html` won’t.

::selection

The `::selection` pseudo-element matches text that is highlighted.

This is supported in WebKit, and with the `-moz` vendor prefix in Firefox. Let’s use it on *The HTML5 Herald*, to bring the selection background and text color in line with the monochrome style of the rest of the site:

css/styles.css (excerpt)

```
::-moz-selection{
  background: #484848;
  color:#fff;
}
::selection {
  background:#484848;
  color:#fff;
}
```

CSS3 Colors

We know you're probably chomping at the bit to put the *really* cool stuff from CSS3 into practice, but before we do there's one more detour we need to take. CSS3 brings with it support for some new ways of describing colors on the page. Since we'll be using these in examples over the next few chapters, it's important we cover them now.

Prior to CSS3, we almost always declared colors using the hexadecimal format (`#FFF`, or `#FFFFFF` for white). It was also possible to declare colors using the `rgb()` notation, providing either integers (0–255) or percentages. For example, white is `rgb(255,255,255)` or `rgb(100%,100%,100%)`. In addition, we had access to a few named colors, like `purple`, `lime`, `aqua`, `red`, and the like. While the color keyword list has been extended in the CSS3 color module² to include 147 additional keyword colors (that are generally well supported), CSS3 also provides us with a number of other options: `HSL`, `HSLA`, and `RGBA`. The most notable change with these new color types is the ability to declare semitransparent colors.

RGBA

RGBA works just like RGB, except that it adds a fourth value: **alpha**, the opacity level. The first three values still represent red, green, and blue. For the alpha value, 1 means fully opaque, 0 is fully transparent, and 0.5 is 50% opaque. You can use any number between 0 and 1, inclusively.

Unlike RGB, which can also be represented with hexadecimal notation as `#RRGGBB`, there is no hexadecimal notation for RGBA. There's been some discussion of including an eight-character hexadecimal value for RGBA as `#RRGGBBAA`, but this has yet to be added to the draft specification.

For an example, let's look at our registration form. We want the form to be a darker color, while still preserving the grainy texture of the site's background. To accomplish this, we'll use an RGBA color of `0,0,0,0.2`—in other words, solid black that's 80% transparent:

² <http://www.w3.org/TR/css3-color/>

css/styles.css (excerpt)

```
form {  
  :  
  background: rgba(0,0,0,0.2) url(../images/bg-form.png) no-repeat  
  ↪bottom center;  
}
```

Since Internet Explorer 8 and below lack support for RGBA, if you declare an RGBA color, make sure you *precede* it with a color IE can understand. IE will render the last color it can make sense of, so it will just skip the RGBA color. Other browsers will understand both colors, but thanks to the CSS cascade, they'll overwrite the IE color with the RGBA color as it comes later.

In the above example, we're actually fine with older versions of IE having no background color, because the color we're using is mostly transparent anyway.

HSL and HSLA

HSL stands for hue, saturation, and lightness. Unlike RGB, where you need to manipulate the saturation or brightness of a color by changing all three color values in concert, with HSL you can tweak either just the saturation, or the lightness, while keeping the same base hue. The syntax for HSL comprises integer for hue, and percentage values for saturation and lightness.³

Although monitors display colors as RGB, the browser simply converts the HSL value you give it into one the monitor can display.

The `hsl()` declaration accepts three values:

- The hue, in degrees from 0 to 359. Some examples: 0 = red, 60 = yellow, 120 = green, 180 = cyan, 240 = blue, and 300 = magenta. Of course, feel free to use everything in between.

³ A full exploration of color theory—along with what is meant by the terms saturation and lightness—is beyond the scope of this book. If you want to read more, Jason Beard's *The Principles of Beautiful Web Design* (SitePoint: Melbourne, 2010) [<http://www.sitepoint.com/books/design2/>] includes a great primer on color.

- The saturation, as a percentage. 100% is the norm for saturation. Saturation of 100% will be the full hue, and saturation of 0 will give you a shade of gray—essentially causing the hue value to be ignored.
- A percentage for lightness, with 50% being the norm. Lightness of 100% will be white, 50% will be the actual hue, and 0% will be black.

HSL also allows for an opacity value. For example, `hsla(300, 100%, 50%, 0.5)` is magenta with full saturation and normal lightness, which is 50% opaque.

HSL mimics the way the human eye perceives color, so it can be more intuitive for designers to understand—and, as mentioned above, it can make adjustments a bit quicker and easier. Feel free to use whatever syntax you're most comfortable with—but remember that if you need to support IE8 or below, you'll generally want to limit yourself to hexadecimal notation.

Let's sum up with a review of all the ways to write colors in CSS. A shade of dark red can be written as:

- `#800000`
- `maroon`
- `rgb(128,0,0)`
- `rgba(128,0,0,1.0)`
- `hsl(0,100%,13%)`
- `hsla(0,100%,13%,1.0)`

Opacity

In addition to specifying transparency with HSLA and RGBA colors, CSS3 also provides us with the `opacity` property. `opacity` sets the opaqueness of the element on which it's declared. Similar to alpha transparency, the opacity value is a floating point number between (and including) 0 and 1. An opacity value of 0 defines the element as fully transparent, whereas an opacity value of 1 means the element is fully opaque.

Let's look at an example:

```
div.halfopaque {  
  background-color: rgb(0, 0, 0);  
  opacity: 0.5;  
  color: #000000;  
}  
  
div.halfalpha{  
  background-color: rgba(0, 0, 0, 0.5);  
  color: #000000;  
}
```

While the two declaration blocks above may seem to be identical at first glance, there’s actually a key difference. While `opacity` sets the opacity value for an element *and all of its children*, a semitransparent RGBA or HSLA color has no impact on elements other than the one it’s declared on.

Looking at the example above, any text in the `halfopaque` div will also be 50% opaque (most likely making it difficult to read!); the text on the `halfalpha` div, though, will still be 100% opaque.

So, while the `opacity` property is a quick and easy solution for creating semitransparent elements, you should be aware of this consequence.

Putting It into Practice

Now that we’ve been through all the available CSS selectors and new color types, we’re ready to really start styling.

For the rest of the chapter, we’ll style a small section of *The HTML5 Herald* front page; this will demonstrate how to add rounded corners, text shadow, and box shadow.

In the right-hand sidebar of *The HTML5 Herald*’s front page are a series of whimsical advertisements—we marked them up as `article` elements within an `aside` way back in Chapter 2. The first of these is an old “Wanted” poster-style ad, advising readers to be on the look out for the armed and dangerous HTML5 and CSS3. The ad’s final appearance is depicted in Figure 6.1.



Figure 6.1. Our “Wanted” ad

You’ll notice that the dark gray box in the center of the ad has a double border with rounded corners, as well as a three-dimensional “pop” to it. The text that reads “<HTML5> & {CSS3}” also has a shadow that offsets it from the background. Thanks to CSS3, all these effects can be achieved with some simple code, and with no reliance on images or JavaScript. Let’s learn how it’s done.

The markup for the box is simply `<HTML5> & {CSS3}`. Other than the HTML entities, it’s as straightforward as it gets!

Before we can apply any styles to it, we need to select it. Of course, we could just add a `class` attribute to the markup, but where’s the fun in that? We’re here to learn CSS3, so we should try and use some fancy new selectors instead.

Our box isn’t the only `a` element on the page, but it might be the only `a` immediately following a paragraph in the sidebar. In this case, that’s good enough to single out the box. We also know how to add some pre-CSS3 styling for the basics, so let’s do that:

css/styles.css (excerpt)

```
aside p + a {
  display: block;
  text-decoration: none;
  border: 5px double;
  color: #ffffff;
  background-color: #484848;
  text-align: center;
```

```
font-size: 28px;  
margin: 5px 5px 9px 5px;  
padding: 15px 0;  
position: relative;  
}
```

Not bad! As Figure 6.2 shows, we're well on our way to the desired appearance. This will also be the appearance shown to IE8 and below except for the font styling, which we'll be adding in Chapter 9.



Figure 6.2. The basic appearance of our ad link, which will be seen by older browsers

Remember that IE6 lacks support for the adjacent sibling selector—so if you need to provide support to that browser, you can use a more common `id` or `class` selector.

This presentation is fine, and should be acceptable—no need for web pages to look identical in all browsers. Users with older versions of Internet Explorer will be unaware that they're missing anything. But we can still provide treats to better browsers. Let's go ahead and add a bit of polish.

Rounded Corners: `border-radius`

The `border-radius` property lets you create rounded corners without the need for images or additional markup. To add rounded corners to our box, we simply add:

```
-moz-border-radius: 25px;  
border-radius: 25px;
```

Safari, Chrome, Opera, IE9, and Firefox 4 support rounded corners without a vendor prefix (just `border-radius`). We still need to include the vendor-prefixed `-moz-border-radius` for Firefox 3.6 and earlier, though by the time you read this those versions may have dwindled enough for it to be no longer necessary.

Figure 6.3 shows what our link looks like with the addition of these properties.



Figure 6.3. Adding rounded corners to our link

The `border-radius` property is actually a shorthand. For our `a` element, the corners are all the same size and symmetrical. If we had wanted different-sized corners, we could declare up to four unique values—`border-radius: 5px 10px 15px 20px;`, for example. Just like padding, margin, and border, you can adjust each value individually:

```
border-top-left-radius: 5px;  
border-top-right-radius: 10px;  
border-bottom-right-radius: 15px;  
border-bottom-left-radius: 40px;
```

The `-moz-` prefixed form for older versions of Firefox uses a slightly different syntax:

```
-moz-border-radius-topleft: 5px;  
-moz-border-radius-topright: 10px;  
-moz-border-radius-bottomright: 15px;  
-moz-border-radius-bottomleft: 40px;
```

The resulting off-kilter box is shown in Figure 6.4.



Figure 6.4. It's possible to set the radius of each corner independently

When using the shorthand `border-radius`, the order of the corners is top-left, top-right, bottom-right, and bottom-left. You can also declare only two values, in which case the first is for top-left and bottom-right, and the second is for top-right and bottom-left. If you declare three values, the first refers to top-left, the second sets both the top-right and bottom-left, and the third is bottom-right.

We recommend using the shorthand—because it’s much shorter, and because until old versions of Firefox no longer require support, it avoids the need to use two different syntaxes.

You can also create asymmetrical corners with a different radius on each side. Rather than being circular, these will appear elliptical. If two values are supplied to any of the four longhand values, you’ll be defining the horizontal and vertical radii of a quarter ellipse respectively. For example, `border-bottom-left-radius: 20px 10px;` will create an elliptical bottom-left corner.

When using the shorthand for elliptical corners, separate the value of the horizontal and vertical radii with a slash. `border-radius: 20px / 10px;` will create four equal, elliptical corners, and `border-radius: 5px 10px 15px 20px / 10px 20px 30px 40px;` will create four unequal, elliptical corners. That last example will create corners seen in Figure 6.5. Interesting? Yes. Aesthetically pleasing? Not so much.



Figure 6.5. Four interesting, unequal, elliptical corners



With an Eye to the Future

When including prefixed properties, always follow with the correctly written, nonprefixed, standards-compliant syntax. This will ensure that your site is forward compatible!

There’s only one other element on *The HTML5 Herald* that uses rounded corners: the registration form’s submit button. Let’s round those corners now:

css/styles.ss (excerpt)

```
input[type=submit] {  
  -moz-border-radius: 10%;  
  border-radius: 10%;  
}
```

You'll note two things about the above CSS: we've used an attribute selector to target the `submit` input type, and we've used percentages instead of pixel values for the rounded corners. This will come in handy if we need to add more forms to the site later; other submit buttons might be smaller than the one on the registration page, and by using percentages, rounded corners will scale in proportion to the size of the button.

The `border-radius` property can be applied to all elements, except the `table` element when the `border-collapse` property is set to `collapse`.



What about older browsers?

Generally speaking, there's no need to provide an identical look in older browsers, but sometimes a client may insist on it. In the case of rounded corners, one common method is to dynamically generate four additional elements—one for each corner. You'd then use JavaScript to add four **spans** to all the elements you want rounded and, in your CSS, provide background images to each span corresponding to the relevant corner.

While methods like this one provide the desired look, they require JavaScript, additional markup, CSS, and/or images. Additionally, if there's a design change—for example, the color, radius, or border—the background images will need to be recreated. Fortunately, there are some JavaScript solutions that provide CSS3 decorations to older versions of IE without requiring additional images or markup; CSS3 PIE⁴ is one that's worth looking into.

⁴ <http://css3pie.com/>

Drop Shadows

CSS3 provides the ability to add drop shadows to elements using the `box-shadow` property. This property lets you specify the color, height, width, blur, and offset of one or multiple inner and/or outer drop shadows on your elements.

We usually think of drop shadows as an effect that makes an element look like it's "hovering" over the page and leaving a shadow; however, with such fine-grained control over all those variables, you can be quite creative. For our advertisement link, we can use a `box-shadow` with no blur to create the appearance of a 3D box.

The `box-shadow` property takes a comma-separated list of shadows as its value. Each shadow is defined by two to four size values, a color, and the key term `inset` for inset—or internal—shadows. If you fail to specify `inset`, the default is for the shadow to be drawn outside of the element:

Let's look at the shadow we're using on our element, so that we can break down what each value is doing:

css/styles.css (excerpt)

```
-webkit-box-shadow: 2px 5px 0 0 rgba(72,72,72,1);  
-moz-box-shadow: 2px 5px 0 0 rgba(72,72,72,1);  
box-shadow: 2px 5px 0 0 rgba(72,72,72,1);
```

The first value is the horizontal offset. A positive value will create a shadow to the right of the element, a negative value to the left. In our case, our shadow is two pixels to the right of the a.

The second value is the vertical offset. A positive value pushes the shadow down, creating a shadow on the bottom of the element. A negative value pushes the shadow up. In our case, the shadow is five pixels below the a.

The third value, if included, is the blur distance of the shadow. The greater the value, the more the shadow is blurred. Only positive values are allowed. Our shadow is not blurred, so we can either include a value of zero (0), or omit the value altogether.

The fourth value determines the spread distance of the shadow. A positive value will cause the shadow shape to expand in all directions. A negative value contracts

the shadow. Our shadow has no spread, so again we can either include a value of zero (0), or omit the value altogether.

The fifth value above is the color. You will generally want to declare the color of the shadow. If it's omitted, the spec states that it should default to the same as the color property of the element. Opera and Firefox support this default behavior, but WebKit doesn't, so be sure to include the color. In the example above, we used an RGBA color. In this particular design, the shadow is a solid color, so we could just have used the hex value. Most of the time, though, shadows will be partially transparent, so you'll be using RGBA or HSLA, usually.

The drop shadow created by these declarations is shown in Figure 6.6.



Figure 6.6. Adding a drop shadow to our box gives it the illusion of depth

By default, the shadow is a drop shadow—occurring on the outside of the box. You can create an inset shadow by adding the word `inset` to the start of your shadow declaration.

Opera, Firefox 4, and IE9 support the nonprefixed syntax. We're still including the `-moz-` prefix for Firefox 3.6 and earlier, and the `-webkit-` prefix for Safari and Chrome. However, current development versions of WebKit support the unprefixed version, and Firefox 4 will soon supplant the older versions, so the need for prefixing should wane.



Drop Shadows on IE6+

To include box shadows in IE6 through IE8, you have to use a proprietary filter like the one shown below. Be warned, though, that it's almost impossible to make it look the same as a CSS3 shadow. You should also be aware that filters have a significant impact on performance, so you should only use them if it's *absolutely* necessary for those older browsers to see your shadows. Moreover, these styles should be in a separate stylesheet targeted to earlier versions of IE with the help of conditional comments—otherwise they'll mess with your standard CSS3 shadows on IE9:

```
filter: shadow(color=#484848, direction=220, Strength=8);
filter:progid:DXImageTransform.Microsoft.dropshadow(OffX=2,
➤OffY=5, Color='#484848', Positive='true');
```



Nonrectangular shadows?

Drop shadows look good on rectangular elements, including those with rounded corners like ours. We're using the `border-radius` property on our element, so the shadow will follow the curve of the corners, which looks good.

Keep in mind, though, that the shadow follows the *edges* of your element, rather than the pixels of your content. So, if you try to use drop shadows on semitransparent images, you'll receive an ugly surprise: the shadow follows the rectangular borders of the image instead of the contour of the image's content.

To include more than one box shadow on an element, define a comma-separated list of shadows. When more than one shadow is specified, the shadows are layered front to back, as if the browser drew the last shadow first, and the previous shadow on top of that.

Like an element's outline, box shadows are *supposed* to be invisible in terms of the box model. In other words, they should have no impact on the layout of a page—they'll overlap other boxes and their shadows if necessary. We say “supposed to,” because there are bugs in some browsers, though these are few, and will likely be fixed fairly quickly.

Inset and Multiple Shadows

The registration form for *The HTML5 Herald* has what looks like a gradient background around the edges, but it's actually a few inset box shadows.

To create an inset box shadow, add the `inset` key term to your declaration. In our case, we have to include two shadows so that we cover all four sides: one shadow for the top left, and one for the bottom right:

css/styles.css (excerpt)

```
form {
  -webkit-box-shadow:
    inset 1px 1px 84px rgba(0,0,0,0.24),
    inset -1px -1px 84px rgba(0,0,0,0.24);
  -moz-box-shadow:
    inset 1px 1px 84px rgba(0,0,0,0.24),
    inset -1px -1px 84px rgba(0,0,0,0.24);
  box-shadow:
    inset 1px 1px 84px rgba(0,0,0,0.24),
    inset -1px -1px 84px rgba(0,0,0,0.24);
}
```

As you can see, to add multiple shadows to an element, you simply need to repeat the same syntax again, separated with a comma.



WebKit and Inset Shadows

Current versions of WebKit-based browsers suffer from *very* slow performance when rendering inset box shadows with a large `blur` value, like the one we're using on *The HTML5 Herald's* registration form.

Because WebKit supports both the `-webkit-` prefixed and unprefixed forms of the `box-shadow` property, we've had to omit both of these from the finished CSS. We could only include the `-moz-` prefixed property, so, unfortunately, Firefox is the sole beneficiary of our nice big inset shadow.

This bug has been fixed in the current development version of the WebKit engine, but it might be some time before the fix makes its way into releases of every WebKit-based browser. Therefore, if you're using inset shadows, be sure to do plenty of browser testing.

Text Shadow

Where `box-shadow` lets us add shadows to boxes, `text-shadow` adds shadows to individual characters in text nodes. Added in CSS2, `text-shadow` has been supported in Safari since version 1, and is now supported in all current browser releases except IE9.

The syntax of the `text-shadow` property is very similar to `box-shadow`, including prefixes, offsets, and the ability to add multiple shadows; the exceptions are that there's no spread, and inset shadows aren't permitted:

```
/* single shadow */
text-shadow: topOffset leftOffset blurRadius color;

/* multiple shadows */
text-shadow: topOffset1 leftOffset1 blurRadius1 color1,
             topOffset2 leftOffset2 blurRadius2 color2,
             topOffset3 leftOffset3 blurRadius3 color3;
```

Like `box-shadow`, when multiple shadows are declared, they're painted from front to back with the first shadow being the topmost. Text shadows appear behind the text itself. If a shadow is so large that it touches another letter, it will continue behind that character.

Our text has a semi-opaque shadow to the bottom right:

css/styles.css (excerpt)

```
text-shadow: 3px 3px 1px rgba(0, 0, 0, 0.5);
```

This states that the shadow extends three pixels below the text, three pixels to the right of the text, is slightly blurred (one pixel), and has a base color of black at 50% opacity.

With that style in place, our ad link is nearly complete, as Figure 6.7 shows. The finishing touch—a custom font—will be added in Chapter 9.



Figure 6.7. Our ad link is looking quite snazzy!

More Shadows

We now know how to create drop shadows on both block-level elements and text nodes. But so far, we’ve only styled a fraction of our page: only one link in one advertisement, in fact. Let’s do the rest of the shadows before moving on.

Looking back at the site design, we can see that all the `h1` elements on the page are uppercase and have drop shadows. The text is dark gray with a very subtle, solid-white drop shadow on the bottom right, providing a bit of depth.⁵ The tagline in the site header also has a drop shadow, but is all lowercase. The taglines for the articles, meanwhile, have no drop shadow.

We know from the section called “CSS3 Selectors” that we can target all these elements without using classes. Let’s target these elements without any additional markup:

css/styles.css (excerpt)

```
h1, h2 {
  text-transform: uppercase;
  text-shadow: 1px 1px #FFFFFF;
}
:not(article) > header h2 {
  text-transform: lowercase;
  text-shadow: 1px 1px #FFFFFF;
}
```

The first declaration targets all the `h1` elements and `h2` elements on the page. The second targets all the `h2` elements that are in a header, but only if that header is not nested in an `article` element.

⁵ See <http://twitter.com/#!/themaninblue/status/27210719975964673>.

Our text shadows are a solid white, so there's no need to use alpha transparent colors, or a blur radius.

Up Next

Now that we have shadows and rounded corners under our belt, it's time to have some more fun with CSS3. In the next chapter, we'll be looking at CSS3 gradients and multiple background images.

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.

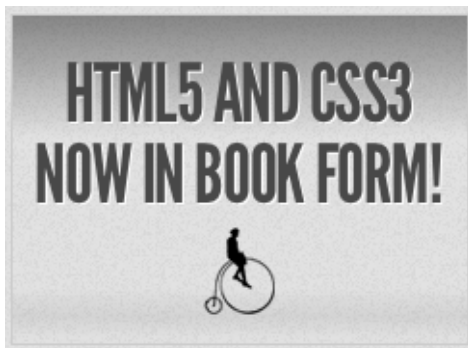


Figure 7.2. A linear gradient in *The HTML5 Herald*

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%);
```

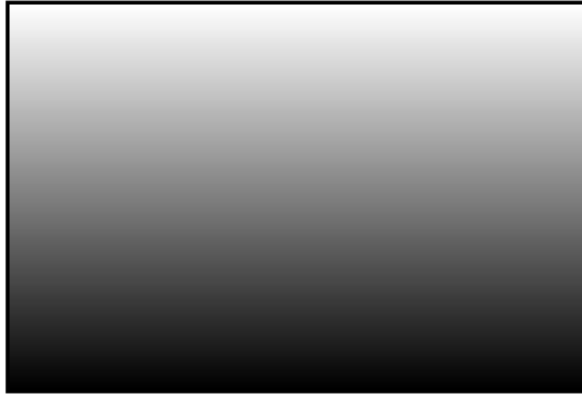


Figure 7.3. A white-to-black gradient from the top center to the bottom center of an element

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.

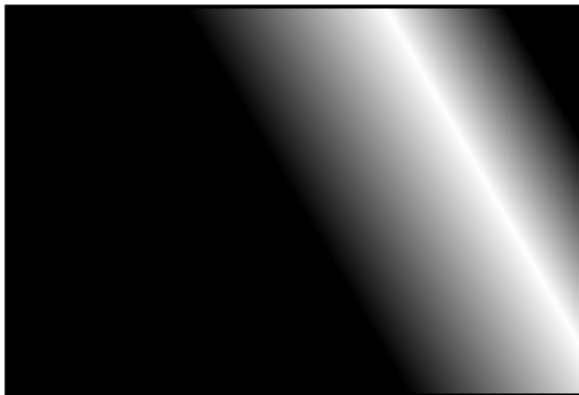


Figure 7.5. A gradient confined to a narrow band by offsetting the start and end color stops

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.

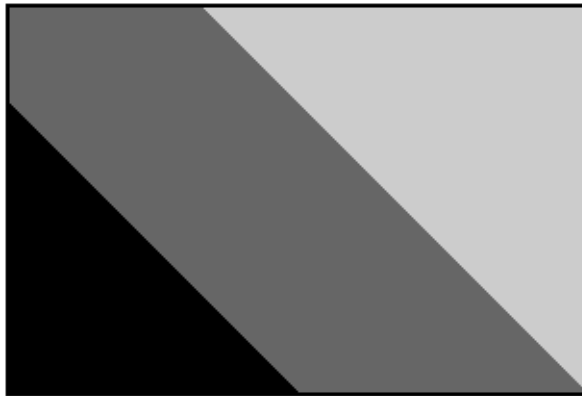


Figure 7.6. Careful placement of color stops can create striped backgrounds

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).

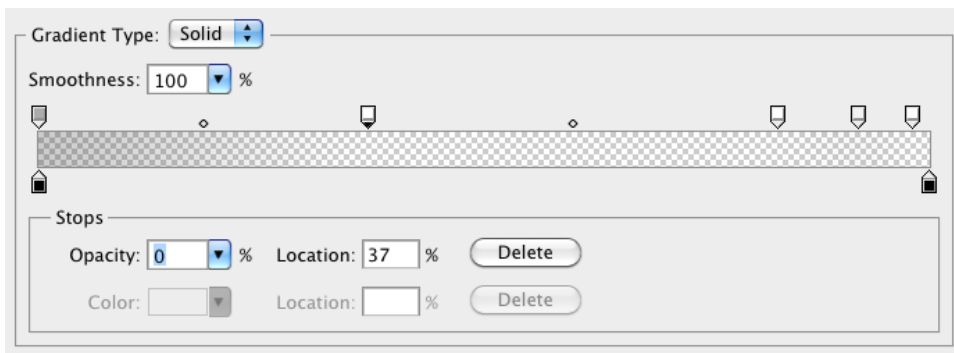


Figure 7.7. An example linear gradient in Photoshop

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) 100%);
}
```

```

        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.

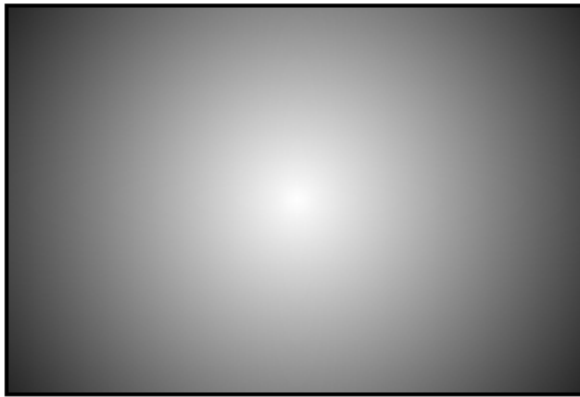


Figure 7.8. A simple, centered radial gradient

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.

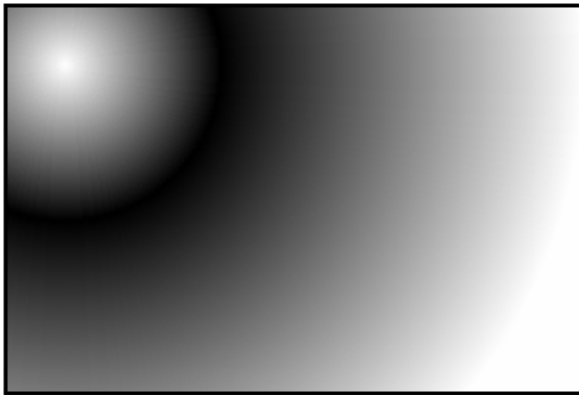


Figure 7.10. A radial gradient with a modified size and shape, and an extra color stop

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.

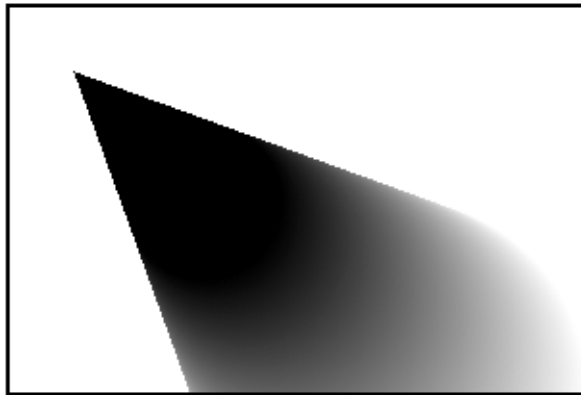


Figure 7.11. The older WebKit radial gradient syntax allowed for some interesting effects

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 rgb(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.

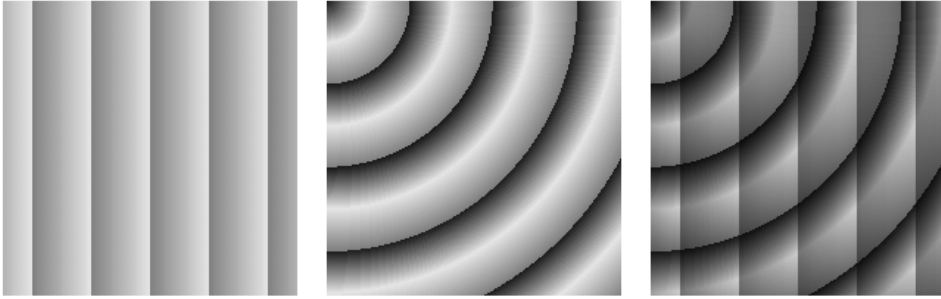


Figure 7.13. A few examples of repeating gradients

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 8

CSS3 Transforms and Transitions

Our page is fairly static. Actually, it's completely static. In Chapter 4 we learned a little about how to alter a form's appearance based on its state with the `:invalid` and `:valid` pseudo-classes. But what about really moving things around? What about changing the appearance of elements—rotating or skewing them?

For years, web designers have relied on JavaScript for in-page animations, and the only way to display text on an angle was to use an image. This is far from ideal. Enter CSS3: without a line of JavaScript or a single JPEG, you can tilt, scale, move, and even flip your elements with ease.

Let's see how it's done.

Transforms

Supported in Firefox 3.5+, Opera 10.5, WebKit since 3.2 (Chrome 1), and even Internet Explorer 9, the CSS3 `transform` property lets you translate, rotate, scale, or skew any element on the page. While some of these effects were possible using previously existing CSS features (like relative and absolute positioning), CSS3 gives you unprecedented control over many more aspects of an element's appearance.

We manipulate an element's appearance using **transform functions**. The value of the `transform` property is one or more transform functions, separated by spaces, which will be applied in the order they're provided. In this book, we'll cover all the two-dimensional transform functions. WebKit also supports the transformation of elements in 3D space—3D transforms—but that's beyond the scope of this book.

To illustrate how transforms work, we'll be working on another advertisement block from *The HTML5 Herald*, shown in Figure 8.1.



Figure 8.1. This block will serve to illustrate CSS3 transforms

Translation

Translation functions allow you to move elements left, right, up, or down. These functions are similar to the behavior of `position: relative;` where you declare `top` and `left`. When you employ a translation function, you're moving elements without impacting the flow of the document.

Unlike `position: relative`, which allows you to position an element either against its current position or against a parent or other ancestor, a translated element can only be moved relative to its current position.

The `translate(x,y)` function moves an element by `x` from the left, and `y` from the top:

```
-webkit-transform: translate(45px, -45px);  
-moz-transform: translate(45px, -45px);  
-ms-transform: translate(45px, -45px);  
-o-transform: translate(45px, -45px);  
transform: translate(45px, -45px);
```

If you only want to move an element vertically or horizontally, you can use the `translateX` or `translateY` functions:

```
-webkit-transform: translateX(45px);
-moz-transform: translateX(45px);
-ms-transform: translateX(45px);
-o-transform: translateX(45px);
transform: translateX(45px);

-webkit-transform: translateY(-45px);
-moz-transform: translateY(-45px);
-ms-transform: translateY(-45px);
-o-transform: translateY(-45px);
transform: translateY(-45px);
```

For our ad, let's say we want to move the word “dukes” over to the right when the user hovers over it, as if it had been punched by our mustachioed pugilist. In the markup, we have:

```
<h1>Put your <span>dukes</span> up sire</h1>
```

Let's apply the style whenever the `h1` is hovered over. This will make the effect more likely to be stumbled across than if it was only triggered by hovering over the span itself:

css/styles.css (excerpt)

```
#ad3 h1:hover span {
  color: #484848;
  -webkit-transform: translateX(40px);
  -moz-transform: translateX(40px);
  -ms-transform: translateX(40px);
  -o-transform: translateX(40px);
  transform: translateX(40px);
}
```

This works in most browsers, but you may have noticed that WebKit's not playing along. What gives? It turns out that WebKit will only allow you to transform block-level elements; inline elements are off-limits. That's easy enough to fix—we'll just add `display: inline-block;` to our span:

css/styles.css (excerpt)

```
#ad3 h1 span {  
  font-size: 30px;  
  color: #999999;  
  display: inline-block;  
  :  
}
```

The result is shown in Figure 8.2.

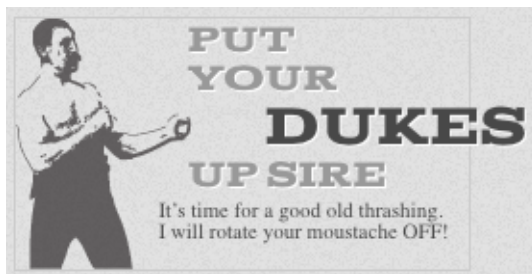


Figure 8.2. The result of our `translate` transform

It's nice, but we can still do better! Let's look at how we can scale our text to make it bigger as well.

Scaling

The `scale(x,y)` function scales an element by the defined factors horizontally and vertically, respectively. If only one value is provided, it will be used for both the x and y scaling. For example, `scale(1)` would leave the element the same size, `scale(2)` would double its proportions, `scale(0.5)` would halve them, and so on. Providing different values will distort the element, as you'd expect:

```
-webkit-transform: scale(1.5,0.25);  
-moz-transform: scale(1.5,0.25);  
-ms-transform: scale(1.5,0.25);  
-o-transform: scale(1.5,0.25);  
transform: scale(1.5,0.25);
```

As with `translate`, you can also use the `scalex(x)` or `scaley(y)` functions. These functions will scale only the horizontal dimensions, or only the vertical dimensions. They are the same as `scale(x,1)` and `scale(1,y)`, respectively.

A scaled element will grow outwards from or shrink inwards towards its center; in other words, the element's center will stay in the same place as its dimensions change. To change this default behavior, you can include the `transform-origin` property, which we'll be covering a bit later.

Let's add a `scale` transform to our span:

css/styles.css (excerpt)

```
#ad3 h1:hover span {
  color: #484848;
  -webkit-transform: translateX(40px) scale(1.5);
  -moz-transform: translateX(40px) scale(1.5);
  -ms-transform: translateX(40px) scale(1.5);
  -o-transform: translateX(40px) scale(1.5);
  transform: translateX(40px) scale(1.5);
}
```

Note that there's no need to declare a new transform—you provide it with a space-separated list of transform functions, so we just add our `scale` to the end of the list.

It's also worth remembering that scaling, like translation, has no impact on the document flow. This means that if you scale inline text, text around it won't reflow to accommodate it. Figure 8.3 shows an example of how this might be a problem. In cases like this, you might want to consider simply adjusting the element's height, width, or font-size instead of using a `scale` transform. Changing those properties will change the space allocated to the element by the browser.



Figure 8.3. Using the `scale` function on inline text can have unwanted results

In our example, however, we want the text to pop out of the ad without reflowing the surrounding text, so the `scale` does exactly what we need it to do. Figure 8.4 shows what our hover state looks like with the `scale` added to the existing translation.



Figure 8.4. Our ad now has plenty of pop

It's looking good, but there's still more to add.

Rotation

The `rotate()` function rotates an element around the point of origin (as with `scale`, by default this is the element's center), by a specified angle value. Generally, angles are declared in degrees, with positive degrees moving clockwise and negative moving counter-clockwise. In addition to degrees, values can be provided in grads, radians, or turns—but we'll just be sticking with degrees.

Let's add a `rotate` transform to our “dukes”:

```
#ad3 h1:hover span {
  color: #484848;
  -webkit-transform: rotate(10deg) translateX(40px) scale(1.5);
  -moz-transform: rotate(10deg) translateX(40px) scale(1.5);
  -ms-transform: rotate(10deg) translateX(40px) scale(1.5);
  -o-transform: rotate(10deg) translateX(40px) scale(1.5);
  transform: rotate(10deg) translateX(40px) scale(1.5);
}
```

We're rotating our `span` by ten degrees clockwise—adding to the effect of text that has just been dealt a powerful uppercut. We are declaring the rotation *before* the `translate` so that it's applied first—remember that transforms are applied in the order provided. Sometimes this will make no difference, but if an effect is behaving differently to what you'd like, it's worth playing with the order of your transforms.

The final transformed text is shown in Figure 8.5.



Figure 8.5. Our text has now been translated, scaled, and rotated—that's quite a punch

There's one more type of transform we're yet to visit. It won't be used on *The HTML5 Herald*, but let's take a look anyway.

Skew

The `skew(x,y)` function specifies a skew along the X and Y axes. As you'd expect, the `x` specifies the skew on the X axis, and the `y` specifies the skew on the Y axis. If the second parameter is omitted, the skew will only occur on the X axis:

```
-webkit-transform: skew(15deg, 4deg);
-moz-transform: skew(15deg, 4deg);
-ms-transform: skew(15deg, 4deg);
-o-transform: skew(15deg, 4deg);
transform: skew(15deg, 4deg);
```

Applying the above styles to a heading, for example, results in the skew shown in Figure 8.6.

A Skewed Perspective

Figure 8.6. Some text with a skew transform applied

As with `translate` and `scale`, there are axis-specific versions of the skew transform: `skewx()` and `skewy()`.

Changing the Origin of the Transform

As we hinted at earlier, you can control the origin from which your transforms are applied. This is done using the `transform-origin` property. It has the same syntax as the `background-position` property, and defaults to the center of the object (so that scales and rotations will be around the center of the box by default).

Let's say you were transforming a circle. Because the default `transform-origin` is the center of the circle, applying a `rotate` transform to a circle would have no visible effect—a circle rotated 90 degrees still looks exactly the same as it did before being rotated. However, if you gave your circle a `transform-origin` of `10% 10%`, you would notice the circle's rotation, as Figure 8.7 illustrates.

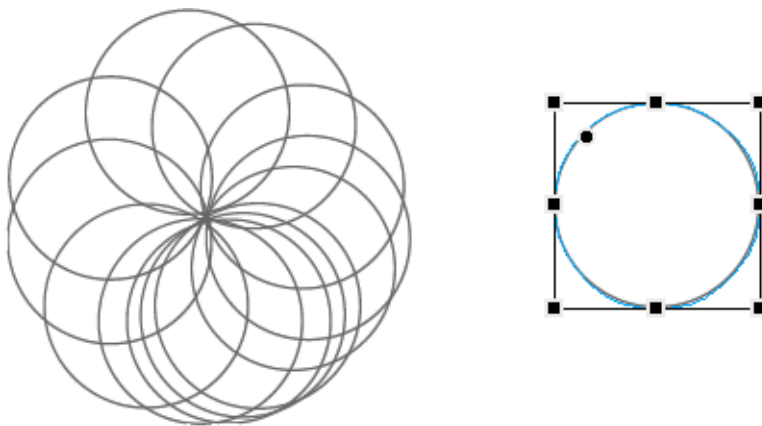


Figure 8.7. Rotating a circle only works if the `transform-origin` has been set

The `transform-origin` property is supported with vendor prefixes in WebKit, Firefox, and Opera:

```
-webkit-transform-origin: 0 0;  
-moz-transform-origin: 0 0;  
-o-transform-origin: 0 0;  
transform-origin: 0 0;
```

Support for Internet Explorer 8 and Earlier

While CSS3 transforms are unsupported in IE6, IE7, or IE8, you can mimic these effects with other CSS properties, including filters. To “translate,” use `position: relative;`, and `top` and `left` values:

```
.translate {
  position: relative;
  top: 200px;
  left: 200px;
}
```

You can also scale an element by altering its width and height. Remember, though, that while transformed elements still take up the space that they did before being scaled, altering a width or height alters the space allocated for the element and can affect the layout.

You can even use filters to rotate an element in Internet Explorer, but it's ugly:

```
.rotate {
  transform: rotate(15deg);
  filter: progid:DXImageTransform.Microsoft.Matrix(
    sizingMethod='auto expand', M11=0.9659258262890683,
    M12=-0.25881904510252074, M21=0.25881904510252074,
    M22=0.9659258262890683);
  -ms-filter: "progid:DXImageTransform.Microsoft.Matrix(
    M11=0.9659258262890683, M12=-0.25881904510252074,
    M21=0.25881904510252074, M22=0.9659258262890683,
    sizingMethod='auto expand')";
  zoom: 1;
}
```

This filter's syntax isn't worth going into here. If you want to rotate an element in Internet Explorer, go to <http://css3please.com/> for cross-browser code for a given rotation. Just edit any of the rotation values, and the other versions will be updated accordingly.

Transitions

As much fun as it's been to have a feature work in IE9, it's time to again leave that browser behind. While Opera, Firefox, and WebKit all support CSS transitions, IE is once again left in the dust.

Transitions allow the values of CSS properties to change over time, essentially providing simple animations. For example, if a link changes color on hover, you can have it gradually fade from one color to the other, instead of a sudden change.

Likewise, you can animate any of the transforms we’ve just seen, so that your pages feel more dynamic.

Animation has certainly been possible for some time with JavaScript, but native CSS transitions require much less processing on the client side, so they’ll generally appear smoother. Especially on mobile devices with limited computing power, this can be a lifesaver.

CSS transitions are declared along with the regular styles on an element. Whenever the target properties change, the browser will apply the transition. Most often, the change will be due to different styles applied to a hover state, for example. However, transitions will work equally well if the property in question is changed using JavaScript. This is significant: rather than writing out an animation in JavaScript, you can simply switch a property value and rely on the browser to do all the heavy lifting.

Here are the steps to create a simple transition using only CSS:

1. Declare the original state of the element in the default style declaration.
2. Declare the final state of your transitioned element; for example, in a hover state.
3. Include the transition functions in your default style declaration, using a few different properties: `transition-property`, `transition-duration`, `transition-timing-function`, and `transition-delay`. We’ll look at each of these and how they work shortly.

The important point to note is that the transition is declared in the default state. Currently, the transition functions need to include the vendor prefixes `-webkit-`, `-o-`, and `-moz-`, for WebKit, Opera, and Firefox, respectively.

This may be a lot to grasp, so let’s go over the various transition values. As we go, we’ll apply a transition to the transforms we added to our ad in the last section, so that the word “dukes” moves smoothly into its new position when hovered.

transition-property

The `transition-property` lists the CSS properties of the element that should be transitioned. Properties that can be made to transition include background, border, and box model properties. You can transition font sizes and weights, but not font

families. The W3C last updated the list of properties that can be transitioned in August 2010:

- background-color and background-position
- border-color, border-spacing, and border-width
- bottom, top, left, and right
- clip
- color
- crop
- font-size and font-weight
- height and width
- letter-spacing
- line-height
- margin
- max-height, max-width, min-height, and min-width
- opacity
- outline-color, outline-offset, and outline-width
- padding
- text-indent
- text-shadow
- vertical-align
- visibility
- word-spacing
- z-index

More properties are available to transition in some browsers, including the transform functions, but they're not (yet) in the proposed specifications. Note also that not all browsers support transitions on all the above properties at the time of writing.

You can provide any number of CSS properties to the `transition-property` declaration, separated by commas. Alternatively, you can use the keyword `all` to indicate that every supported property should be animated.

In the case of our ad, we'll apply the transition to the `transform` property:

```
#ad2 h1 span {  
  -webkit-transition-property: -webkit-transform;  
  -moz-transition-property: -moz-transform;
```

```
-o-transition-property: -o-transform;  
transition-property: transform;  
}
```

Note that we need to specify the prefixed forms of properties—you can’t animate `transform` in a browser that only understands `-moz-transform`, for example.

Because the list of properties that can transition is in flux, be careful what you include: it’s possible that a property that doesn’t animate at the time you’re writing your page eventually will, so be selective in the properties you specify, and only use `all` if you really want to animate every property.

So far these styles will have no effect; that’s because we still need to specify the duration of the transition.

transition-duration

The `transition-duration` property sets how long the transition will take. You can specify this either in seconds (s) or milliseconds (ms). We’d like our animation to be fairly quick, so we’ll specify 0.2 seconds, or 200 milliseconds:

```
-webkit-transition-duration: 0.2s;  
-moz-transition-duration: 0.2s;  
-o-transition-duration: 0.2s;  
transition-duration: 0.2s;
```

With those styles in place, our span will transition on hover. Notice that the “reverse” transition also takes place over the same duration—the element returns to its previous position.



Automatic Graceful Degradation

Although transitions are only supported in some browsers, the fact that they’re declared separately from the properties that are changing means that those changes will still be apparent in browsers without support for transitions. Those browsers will still apply the `:hover` (or other) state just fine, except that the changes will happen instantly rather than being transitioned over time.

transition-timing-function

The `transition-timing-function` lets you control the pace of the transition in even more granular detail. Do you want your animation to start off slow and get faster, start off fast and end slower, advance at an even keel, or some other variation? You can specify one of the key terms `ease`, `linear`, `ease-in`, `ease-out`, or `ease-in-out`. The best way to familiarize yourself with them is to play around and try them all. Most often, one will just feel right for the effect you're aiming to create. Remember to set a relatively long `transition-duration` when testing timing functions—if it's too fast, you won't be able to tell the difference.

In addition to those five key terms, you can also describe your timing function more precisely using the `cubic-bezier()` function. It accepts four numeric parameters; for example, `linear` is the same as `cubic-bezier(0.0, 0.0, 1.0, 1.0)`. If you've taken six years of calculus, the method of writing a cubic Bézier function might make sense; otherwise, it's likely you'll want to stick to the five basic timing functions. You can also look at online tools that let you play with different values, such as <http://www.netzgesta.de/dev/cubic-bezier-timing-function.html>.

For our transition, we'll use `ease-out`:

```
-webkit-transition-timing-function: ease-out;  
-moz-transition-timing-function: ease-out;  
-o-transition-timing-function: ease-out;  
transition-timing-function: ease-out;
```

This makes the transition fast to start with, becoming slower as it progresses. Of course, with a 0.2 second duration, the difference is barely perceptible.

transition-delay

Finally, by using the `transition-delay` property, it's also possible to introduce a delay before the animation begins. Normally, a transition begins immediately, so the default is 0. Include the number of milliseconds (ms) or seconds (s) to delay the transition:

```
-webkit-transition-delay: 250ms;  
-moz-transition-delay: 250ms;  
-o-transition-delay: 250ms;  
transition-delay: 250ms;
```



Negative Delays

Interestingly, a negative time delay that is less than the duration of the entire transition will cause it to start immediately, but it will start partway through the animation. For example, if you have a delay of -500ms on a 2s transition, the transition will start a quarter of the way through, and will last 1.5 seconds. This might be used to create some interesting effects, so it's worth being aware of.

The transition Shorthand Property

With four transition properties and three vendor prefixes, you could wind up with 16 lines of CSS for a single transition. Fortunately, as with other properties, there's a shorthand available. The `transition` property is shorthand for the four transition functions described above. Let's take another look at our transition so far:

```
#ad2 h1 span {  
  -webkit-transition-property: -webkit-transform, color;  
  -moz-transition-property: -moz-transform, color;  
  -o-transition-property: -o-transform, color;  
  transition-property: transform, color;  
  -webkit-transition-duration: 0.2s;  
  -moz-transition-duration: 0.2s;  
  -o-transition-duration: 0.2s;  
  transition-duration: 0.2s;  
  -webkit-transition-timing-function: ease-out;  
  -moz-transition-timing-function: ease-out;  
  -o-transition-timing-function: ease-out;  
  transition-timing-function: ease-out;  
}
```

Now let's combine all those values into a shorthand declaration:

css/styles.css (excerpt)

```
#ad2 h1 span {
  -webkit-transition: -webkit-transform 0.2s ease-out;
  -moz-transition: -moz-transform 0.2s ease-out;
  -o-transition: -o-transform 0.2s ease-out;
  transition: transform 0.2s ease-out;
}
```

Note that order of the values is important and must be as follows (though you don't need to specify all four values):

- 1.transition-property
- 2.transition-duration
- 3.transition-function
- 4.transition-delay

Multiple Transitions

The transition properties allow for multiple transitions in one call. For example, if we want to change the color at the same time as changing the rotation and size, we can.

Let's say instead of just transitioning the rotation, we transition the text's color property as well. We'd have to first include a color property in the transitioned style declaration, and then either the color property in the transition-property value list, or use the key term `all`:

```
transition-property: transform, color;
transition-duration: 0.2s;
transition-timing-function: ease-out;
```

You can also specify different durations and timing functions for each property being animated. Simply include each value in a comma-separated list, using the same order as in your transition-property:

```
transition-property: transform, color;
transition-duration: 0.2s, 0.1s;
transition-timing-function: ease-out, linear;
```

The above properties will apply an `ease-out` transition over 0.2 seconds to the `transform`, but a `linear` transition over 0.1 seconds to the `color`.

It's possible to specify multiple transitions when using the shorthand `transition` property also. In this case, specify all the values for each transition together, and separate each transition with commas:

```
transition: color 0.2s ease-out, transform 0.2s ease-out;
```

If you want to change both properties at the same rate and delay, you can include both property names or, since you are transitioning all the properties listed in the `hover` state anyway, you can employ the `all` keyword.

When using the `all` keyword, all the properties transition at the same rate, speed, and delay:

```
-webkit-transition: all 0.2s ease-out;  
-moz-transition: all 0.2s ease-out;  
-o-transition: all 0.2s ease-out;  
transition: all 0.2s ease-out;
```

If you don't want all your properties to transition at the same rate, or if you just want a select few to have a transition effect, include the various transition properties as a comma-separated list, including, at minimum, the `transition-property` and `transition-duration` for each.

Animations

Transitions animate elements over time; however, they're limited in what they can do. You can define starting and ending states, but there's no fine-grained control over any intermediate states. CSS animations, unlike transitions, allow you to control each step of an animation via **keyframes**. If you've ever worked with Flash, you're likely very familiar with the concept of keyframes; if not, don't worry, it's fairly straightforward. A keyframe is a snapshot that defines a starting or end point of any smooth transition. With CSS transitions, we're essentially limited to defining the first and last keyframes. CSS animations allow us to add any number of keyframes in between, to guide our animation in more complex ways.

At the time of this writing, only WebKit supports CSS animation. This means that support on the desktop is limited, but support on mobile devices is fairly good, as the default browsers on iOS and Android both run on WebKit. As we've already mentioned, the lack of powerful processors on many mobile devices make CSS animations a great alternative to weighty, CPU-intensive JavaScript animation.

Keyframes

To animate an element in CSS, you first create a named animation, then attach it to an element in that element's property declaration block. Animations in themselves don't do anything; in order to animate an element, you will need to associate the animation with that element.

To create an animation, use the `@keyframes` rule—or `@-webkit-keyframes` for current WebKit implementations—followed by a name of your choosing, which will serve as the identifier for the animation. Then, you can specify your keyframes.

For an animation called `myAnimation`, the `@keyframes` rule would look like this:

```
@-webkit-keyframes 'myAnimation'{  
  /* put animation keyframes here */  
}
```

Each keyframe looks like its own nested CSS declaration block. Instead of a selector, though, you use the keywords `from` or `to`, a percentage value, or a comma-separated list of percentage values. This value specifies how far along the animation the keyframe is located.

Inside each keyframe, include the desired properties and values. Between each keyframe, values will be smoothly interpolated by the browser's animation engine.

Keyframes can be specified in any order; it's the percentage values, rather than the order of the declarations, that determine the sequence of keyframes in the animation.

Here are a few simple animations:

```
@-webkit-keyframes 'appear' {  
  0% {  
    opacity: 0;  
  }
```

```
100% {  
    opacity: 1;  
}  
}  
  
@-webkit-keyframes 'disappear' {  
    to {  
        opacity: 0;  
    }  
    from {  
        opacity: 1;  
    }  
}  
  
@-webkit-keyframes 'appearDisappear' {  
    0%, 100% {  
        opacity: 0;  
    }  
    20%, 80% {  
        opacity: 1;  
    }  
}
```

The last animation is worth paying extra attention to: we've applied the same styles to 0% and 100%, and to 20% and 80%. In this case, it means the element will start out invisible (`opacity: 0;`), fade in to visible by 20% of the way through the duration, remain visible until 80%, then fade out.

We've created three animations, but they aren't attached to any elements. Once we have defined an animation, the next step is to apply it to one or more elements using the various animation properties.

Animation Properties

The animation properties supported by WebKit are as follows, with the `-webkit-` vendor prefix.

animation-name

This property is used to attach an animation (defined using the `@keyframes` syntax previously) to an element:

```
-webkit-animation-name: 'appear';
```

Note that the quotes around the animation name in both the property value and the `@keyframe` selector are optional. We recommend including them to keep your styles as legible as possible, and to avoid conflicts.

animation-duration

The `animation-duration` property defines the length of time, in seconds or milliseconds, an animation takes to complete one iteration (all the way through, from 0% to 100%):

```
-webkit-animation-duration: 300ms;
```

animation-timing-function

Like the `transition-timing-function` property, the `animation-timing-function` determines how the animation will progress over its duration. The options are the same as for `transition-timing-function`: `ease`, `linear`, `ease-in`, `ease-out`, `ease-in-out`, or `cubic-bezier`:

```
-webkit-animation-timing-function: linear;
```

animation-iteration-count

This property lets you define how many times the animation will play through. The value is generally an integer, but you can also use numbers with decimal points (in which case, the animation will end partway through a run), or the value `infinite` for endlessly repeating animations. If omitted, it will default to 1, in which case the animation will occur only once:

```
-webkit-animation-iteration-count: infinite;
```

animation-direction

When the animation iterates, you can use the `animation-direction` property with the value `alternate` to make every other iteration play the animation backwards. For example, in a bouncing ball animation, you could provide keyframes for the

falling ball, and then use `animation-direction: alternate;` to reverse it on every second play through:

```
-webkit-animation-direction: alternate;
```

The default is `normal`, so the animation will play forwards on each iteration.

When animations are played in reverse, timing functions are also reversed; for example, `ease-in` becomes `ease-out`.

animation-delay

Used to define how many milliseconds or seconds to wait before the browser begins the animation:

```
-webkit-animation-delay: 15s;
```

animation-fill-mode

The `animation-fill-mode` property defines what happens before the animation begins and after the animation concludes. By default, an animation won't affect property values outside of its runs, but with `animation-fill-mode`, we can override this default behavior. We tell the animation to “sit and wait” on the first keyframe until the animation starts, or stop on the last keyframe without reverting to the original values at the conclusion of the animation, or both.

The available values are `none`, `forwards`, `backwards`, or `both`. The default is `none`, in which case the animation proceeds and ends as expected, reverting to the initial keyframes when the animation completes its final iteration. When set to `forwards`, the animation continues to apply the values of the last keyframes after the animation ends. When set to `backwards`, the animation's initial keyframes are applied as soon as the animation style is applied to an element. As you'd expect, `both` applies both the `backwards` and `forwards` effects:

```
-webkit-animation-fill-mode: forwards;
```

animation-play-state

The `animation-play-state` property defines whether the animation is running or paused. A paused animation displays the current state of the animation statically.

When a paused animation is resumed, it restarts from the current position. This provides a simple way to control CSS animations from your JavaScript.

The Shorthand `animation` Property

Fortunately, there's a shorthand for all these animation properties. The `animation` property takes as its value a space-separated list of values for the longhand `animation-name`, `animation-duration`, `animation-timing-function`, `animation-delay`, `animation-iteration-count`, `animation-direction`, and `animation-fill-mode` properties:

```
.verbose {
  -webkit-animation-name: 'appear';
  -webkit-animation-duration: 300ms;
  -webkit-animation-timing-function: ease-in;
  -webkit-animation-iteration-count: 1;
  -webkit-animation-direction: alternate;
  -webkit-animation-delay: 5s;
  s-webkit-animation-fill-mode: backwards;
}

/* shorthand */
.concise {
  -webkit-animation: 'appear' 300ms ease-in 1 alternate 5s
  ➡backwards;
}
```

To declare multiple animations on an element, include a grouping for each animation name, with each shorthand grouping separated by a comma. For example:

```
.target {
  -webkit-animation:
    'animationOne' 300ms ease-in 0s backwards,
    'animationTwo' 600ms ease-out 1s forwards;
}
```

Moving On

With transforms, transitions, and animations, our site is looking more dynamic. Remember the old maxim, though: just because you can, doesn't mean you should. Animations were aplenty on the Web in the late nineties; a lot of us remember flashing banners and scrolling marquees, and to some extent, that problem still exists today. Use animations and transitions where it makes sense, enhancing the user experience—and skip it everywhere else.

We still have a few lessons to learn in CSS3 to make our website look more like an old-time newspaper. In the next chapter, we'll learn about creating columns without relying on `float`, and how to include fancy fonts that aren't installed by default on our users' computers.

Chapter 9

Embedded Fonts and Multicolumn Layouts

We've added quite a lot of decoration to *The HTML5 Herald*, but we're still missing some key components to give it that really old-fashioned feel. To look like a real newspaper, the text of the articles should be laid out in narrow columns, and we should use some suitably appropriate fonts for the period.

In this chapter, we'll finish up the look and feel of our website with CSS3 columns and `@font-face`.

Web Fonts with `@font-face`

Since the early days of the Web, designers have been dreaming of creating sites with beautiful typography. But, as we all know too well, browsers are limited to rendering text in the fonts the user has installed on their system. In practical terms, this has limited most sites to a handful of fonts: Arial, Verdana, Times, Georgia, and a few others.

Over the years, we have come up with a number of clever workarounds for this problem. We created JPEGs and PNGs for site titles, logos, buttons, and navigation elements. When those elements required additional states or variants, we created even more images, or image sprites to ensure the page stayed snappy and responsive. Whenever the design or text changed, all those images had to be recreated.

This can be an acceptable solution for some elements on a page, but it's just unrealistic to expect a designer to handcraft the title of every new article in Photoshop and then upload it to the site. So, for key page elements that need to change frequently, we were stuck with those same few fonts.

To fill this typographic void, some very cool font embedding scripts were created, like sIFR, based on Flash and JavaScript, and the canvas-based Cufón. While these methods have been a great stopgap measure, allowing us to include our own fonts, they had severe drawbacks. Sometimes they were tricky to implement, and they required that JavaScript be enabled and, in the case of sIFR, the Flash plugin be installed. In addition, they significantly slowed the page's download and rendering.

Fortunately, there's now a better way: `@font-face` is a pure CSS solution for embedding fonts—and it's supported on every browser with any kind of market share, from IE6 on up.

We'll be including two embedded fonts on *The HTML5 Herald* site: League Gothic from The League of Movable Type,¹ and Acknowledgement Medium by Ben Weiner of Reading Type.² The two fonts are shown, respectively, in Figure 9.1 and Figure 9.2.



Figure 9.1. League Gothic

¹ <http://www.theleagueofmoveabletype.com/>

² <http://www.readingtype.org/>

ACKNOWLEDGEMENT

Figure 9.2. Acknowledgement Medium

We'll now look at how we can embed these fonts and use them to power any of the text on our site, just as if they were installed on our users' machines.

Implementing @font-face

@font-face is one of several CSS **at-rules**, like @media, @import, @page, and the one we've just seen, @keyframes. At-rules are ways of encapsulating several rules together in a declaration to serve as instructions to the browser's CSS processor. The @font-face at-rule allows us to specify custom fonts that we can then include in other declaration blocks.

To include fonts using @font-face, you have to:

1. load the font file onto your servers in a variety of formats to support all the different browsers
2. name, describe, and link to that font in an @font-face rule
3. include the font's name in a font-family property value, just as you would for system fonts

You already know how to upload a file onto a server, so we'll discuss the details of the various file types in the next section. For now, we'll focus on the second and third steps so that you can become familiar with the syntax of @font-face.

Here's an example of an @font-face block:

```
@font-face {  
  font-family: 'fontName';  
  src: source;  
  font-weight: weight;  
  font-style: style;  
}
```

The font family and source are required, but the weight and style are optional.

You need to include a separate `@font-face` at-rule for every font you contain in your site. You'll also have to include a separate at-rule for each variation of the font—regular, thin, thick, italic, black, and so on. *The HTML5 Herald* will require two imported fonts, so we'll include two `@font-face` blocks:

css/styles.css (excerpt)

```
@font-face {  
  :  
}  
  
@font-face {  
  :  
}
```

The `font-family` part of the `@font-face` at-rule declaration is slightly different from the `font-family` property with which you're already familiar. In this case, we're *declaring* a name for our font, rather than assigning a font with a given name to an element. The font name can be anything you like—it's only a reference to a font file, so it needn't even correspond to the name of the font. Of course, it makes sense to use the font's name to keep your CSS readable and maintainable. It's good to settle on a convention and stick to it for all your fonts. For our two fonts, we'll use camel case:

css/styles.css (excerpt)

```
@font-face {  
  font-family: 'LeagueGothic';  
}  
  
@font-face {  
  font-family: 'AcknowledgementMedium';  
}
```

Declaring Font Sources

Now that we have a skeleton laid out for our `@font-face` rules, and we've given each of them a name, it's time to link them up to the actual font files. The `src` property can take several formats. Additionally, you can declare more than one source. If the browser fails to find the first source, it will try for the next one, and so on, until it either finds a source, or it runs out of options.

Let's add more formats to our League Gothic declaration:

css/styles.css (excerpt)

```
@font-face {
  font-family: 'LeagueGothicRegular';
  src: url('../fonts/League_Gothic-webfont.eot') format('eot'),
       url('../fonts/League_Gothic-webfont.woff') format('woff'),
       url('../fonts/League_Gothic-webfont.ttf') format('truetype'),
       url('../fonts/League_Gothic-webfont.svg#webfontFHZvtkso')
  ↪format('svg');
}
```

There are four font sources listed in the code block above. The first declaration is an EOT font declaration, a proprietary format for Internet Explorer, and the only file type understood by IE4–8.

Then we define the WOFF (Web Open Font Format, an emerging standard), OTF (OpenType), TTF (TrueType), and SVG (Scalable Vector Graphics) font files. While most desktop browsers will use one of the first three declarations, be sure to include the SVG format, which was originally the only format supported by the iPhone.³

Table 9.1 shows a breakdown of browser support for different formats. As you can see, there's no single format that's supported in every browser, so we need to provide a number of formats, as we did with video in Chapter 5.

Table 9.1. Browser support for font formats

	IE	Safari	Chrome	Firefox	Opera	iOS
@font-face	4+	3.1+	4+	3.5+	10+	3.2+
WOFF	9+	6+	6+	3.6+	11.1+	
OTF		3.1+	4+	3.5+	10+	4.2+
TTF	9+?	3.1+	4+	3.5+	10+	4.2+
SVG		3.1+	4+		10+	3.2+
EOT	4+					

³ The iPhone recently expanded support to include OTF in 4.2, but it still makes sense to include SVG for the time being.

Adding these extra font formats ensures support for every browser, but unfortunately it will cause problems in versions of IE older than IE9. Those browsers will see everything between the first `url(` and the last `)` as one URL, so will fail to load the font. At first, it would seem that we've been given the choice between supporting IE and supporting every other browser, but fortunately there's a solution. Detailed in a FontSpring blog post,⁴ it involves adding a query string to the end of the EOT URL. This tricks the browser into thinking that the rest of the `src` property is a continuation of that query string, so it goes looking for the correct URL and loads the font:

(excerpt)

```
@font-face {
  font-family: 'LeagueGothicRegular';
  src: url('../fonts/League_Gothic-webfont.eot?#iefix')
  ↳format('eot'),
      url('../fonts/League_Gothic-webfont.woff') format('woff'),
      url('../fonts/League_Gothic-webfont.ttf') format('truetype'),
      url('../fonts/League_Gothic-webfont.svg#webfontFHvztkso')
  ↳format('svg');
}
```

This syntax has one potential point of failure: IE9 has a feature called **compatibility mode**, in which it will attempt to render pages the same way IE7 or 8 would. This was introduced to prevent older sites appearing broken in IE9's more standards-compliant rendering. However, IE9 in compatibility mode doesn't reproduce the bug in loading the EOT font, so the above declaration will fail. To compensate for this, you can add an additional EOT URL in a separate `src` property:

(excerpt)

```
@font-face {
  font-family: 'LeagueGothicRegular';
  src: url('../fonts/League_Gothic-webfont.eot');
  src: url('../fonts/League_Gothic-webfont.eot?#iefix')
  ↳format('eot'),
      url('../fonts/League_Gothic-webfont.woff') format('woff'),
      url('../fonts/League_Gothic-webfont.ttf') format('truetype'),
```

⁴ <http://www.fontspring.com/blog/the-new-bulletproof-font-face-syntax>

```

        url('../fonts/League_Gothic-webfont.svg#webfontFHZvtkso')
    ↪format('svg');
}

```

This may be an unnecessary precaution, as generally a user would need to deliberately switch IE to compatibility mode while viewing your site for this issue to arise. Alternatively, you could also force IE out of compatibility mode by adding this meta element to your document's head:

```
<meta http-equiv="X-UA-Compatible" content="IE=Edge">
```

It's also possible to achieve the same result by adding an extra HTTP header; this can be done with a directive in your **.htaccess** file (or equivalent):

```

<IfModule mod_setenvif.c>
  <IfModule mod_headers.c>
    BrowserMatch MSIE ie
    Header set X-UA-Compatible "IE=Edge"
  </IfModule>
</IfModule>

```

Font Property Descriptors

Font property descriptors—including **font-style**, **font-variant**, **font-weight**, and others—can optionally be added to define the characteristics of the font face, and are used to match styles to specific font faces. The values are the same as the equivalent CSS properties:

```

@font-face {
  font-family: 'LeagueGothicRegular';
  src: url('../fonts/League_Gothic-webfont.eot');
  src: url('../fonts/League_Gothic-webfont.eot?#iefix')
  ↪format('eot'),
        url('../fonts/League_Gothic-webfont.woff') format('woff'),
        url('../fonts/League_Gothic-webfont.ttf') format('truetype'),
        url('../fonts/League_Gothic-webfont.svg#webfontFHZvtkso')
  ↪format('svg');
  font-weight: bold;
  font-style: normal;
}

```

Again, the behavior is different from what you'd expect. You are not telling the browser to make the font bold; rather, you're telling it that this *is* the bold variant of the font. This can be confusing, and the behavior can be quirky in some browsers.

However, there is a reason to use the `font-weight` or `font-style` descriptor in the `@font-face` rule declaration. You can declare several font sources for the same `font-family` name:

```
@font-face {  
  font-family: 'CoolFont';  
  font-style: normal;  
  src: url(fonts/CoolFontStd.ttf);  
}  
  
@font-face {  
  font-family: 'CoolFont';  
  font-style: italic;  
  src: url(fonts/CoolFontItalic.ttf);  
}  
  
.whichFont {  
  font-family: 'CoolFont';  
}
```

Notice that both at-rules use the same font-family name, but different font styles. In this example, the `.whichFont` element will use the **CoolFontStd.ttf** font, because it matches the style given in that at-rule. However, if the element were to inherit an italic font style, it would switch to using the **CoolFontItalic.ttf** font instead.

Unicode Range

Also available is the `unicode-range` descriptor, which is employed to define the range of Unicode characters supported by the font. If this property is omitted, the entire range of characters included in the font file will be made available.

We won't be using this on our site, but here's an example of what it looks like:

```
unicode-range: U+000-49F, U+2000-27FF, U+2900-2BFF, U+1D400-1D7FF;
```

Applying the Font

Once the font is declared using the `@font-face` syntax, you can then refer to it as you would any normal system font in your CSS: include it in a “stack” as the value of a `font-family` property. It’s a good idea to declare a fallback font or two in case your embedded font fails to load.

Let’s look at one example from *The HTML5 Herald*:

css/styles.css (excerpt)

```
h1 {  
  text-shadow: #fff 1px 1px;  
  font-family: LeagueGothic, Tahoma, Geneva, sans-serif;  
  text-transform: uppercase;  
  line-height: 1;  
}
```

Our two embedded fonts are used in a number of different places in our stylesheet, but you get the idea.

Legal Considerations

We’ve included the markup for two fonts on our site, but we’re yet to put the font files themselves in place. We found both of these fonts freely available online. They are both licensed as **freeware**—that is, they’re free to use for both personal and commercial use. Generally, you should consider this the only kind of font you should use for `@font-face`, unless you’re using a third-party service.

How is `@font-face` any different from using a certain font in an image file? By having a website on the Internet, your font source files are hosted on publicly available web servers, so in theory anyone can download them. In fact, in order to render the text on your page, the browser *has* to download the font files. By using `@font-face`, you’re distributing the font to everyone who visits your site. In order to include a font on your website, then, you need to be legally permitted to distribute the font.

Owning or purchasing a font doesn’t mean you have the legal right to redistribute it—in the same way that buying a song on iTunes doesn’t grant you the right to throw it up on your website for anyone to download. Licenses that allow you to

distribute fonts are more expensive (and rarer) than licenses allowing you to use a font on one computer for personal or even commercial use.

However, there are several websites that have free downloadable web fonts with Creative Commons,⁵ shareware, or freeware licensing. Alternatively, there are paid and subscription services that allow you to purchase or rent fonts, generally providing you with ready-made scripts or stylesheets that make them easy to use with `@font-face`.

A few sites providing web font services include Typekit,⁶ Typotheque,⁷ Webtype,⁸ Fontdeck,⁹ and Fonts.com¹⁰.

Google's web fonts directory¹¹ has a growing collection of fonts provided free of charge and served from Google's servers. It simply provides you with a URL pointing to a stylesheet that includes all the required `@font-face` rules, so all you need to do is add a `link` element to your document in order to start using a font.

When selecting a service, font selection and price are certainly important, but there are other considerations. Make sure that any service you choose to use takes download speed into consideration. Font files can be fairly large, potentially containing several thousand characters. Good services allow you to select character subsets, as well as font-style subsets, to decrease the file size. Bear in mind, also, that some services require JavaScript in order to function.

Creating Various Font File Types: Font Squirrel

If you have a font that you're legally allowed to redistribute, there'll be no need for you to use any of the font services above. You will, however, have to convert your font into the various formats required to be compatible with every browser on the market. So how do you go about converting your fonts into all these formats?

⁵ If you're unfamiliar with Creative Commons licenses, you can find out more at <http://creativecommons.org/>.

⁶ <http://typekit.com/>

⁷ <http://www.typotheque.com/>

⁸ <http://www.webtype.com/>

⁹ <http://fontdeck.com/>

¹⁰ <http://webfonts.fonts.com>

¹¹ <http://code.google.com/apis/webfonts/>

One of the easiest tools for this purpose is Font Squirrel’s @font-face generator.¹² This service allows you to select fonts from your desktop with a few clicks of your mouse and convert them to TTF, EOT, WOFF, SVG, SVGZ, and even a Base64 encoded version.¹³

By default, the **Optimal** option is selected for generating an @font-face kit; however, in some cases you can decrease the file sizes by choosing **Expert...** and creating a character subset. Rather than including every conceivable character in the font file, you can limit yourself to those you know will be used on your site.

For example, on *The HTML5 Herald* site, the Acknowledgement Medium font is used only in specific ad blocks and headings, so we need just a small set of characters. All the text set in this font is uppercase, so let’s restrict our font to uppercase letters, punctuation, and numbers, as shown in Figure 9.3.

The screenshot shows the 'Subsetting' section of the Font Squirrel's @font-face generator. It has three radio buttons: 'Basic Subsetting' (selected), 'Custom Subsetting...', and 'No Subsetting'. Below 'Basic Subsetting' is the text 'Western languages'. Below 'Custom Subsetting...' is 'Custom language support'. The 'Character Encoding' section has a checkbox for 'Mac Roman'. The 'Character Type' section has a grid of checkboxes: 'Lowercase' (unchecked), 'Uppercase' (checked), 'Numbers' (checked), 'Punctuation' (checked), 'Currency' (unchecked), 'Typographics' (unchecked), 'Math Symbols' (unchecked), 'Alt Punctuation' (unchecked), 'Lower Accents' (unchecked), 'Upper Accents' (unchecked), and 'Diacriticals' (unchecked).

Figure 9.3. Selecting a subset of characters in Font Squirrel’s @font-face generator

Figure 9.4 below shows how the file sizes of our subsetting fonts stack up against the default character sets. In our case, the uppercase-and-punctuation-only fonts are 25–30% smaller than the default character sets. Font Squirrel even lets you specify certain characters for your subset, so there’s no need to include all the letters of the alphabet if you know you won’t use them.

¹² <http://www.fontsquirrel.com/fontface/generator>

¹³ Base64 encoding is a way of including the entire contents of a font file directly in your CSS file. Sometimes this can provide performance benefits by avoiding an extra HTTP request, but that’s beyond the scope of this book. Don’t sweat it, though—the files generated by the default settings should be fine for most uses.









 acknowledgement-subset.eot	12 KB
 acknowledgement-subset.svg	25 KB
 acknowledgement-subset.ttf	20 KB
 acknowledgement-subset.woff	12 KB
 Acknowledgement-webfont.eot	29 KB
 Acknowledgement-webfont.svg	37 KB
 Acknowledgement-webfont.ttf	29 KB
 Acknowledgement-webfont.woff	16 KB

Figure 9.4. File sizes of subsetted fonts can be substantially smaller

For the League Gothic font, we'll need a more expanded character subset. This font is used for article titles, which are all uppercase like our ads, so we can again omit lowercase letters; however, we should consider that content for titles may include a wider range of possible characters. Moreover, users might use in-browser tools, or Google Translate, to translate the content on the page—in which case other characters might be required. So, for League Gothic, we'll go with the default **Basic Subsetting**—this will give you all the characters required for Western languages.

When employing `@font-face`, as a general rule minimize font file size as much as reasonably possible, while making sure to include enough characters so that a translated version of your site is still accessible.

Once you've uploaded your font for processing and selected all your options, press **Download Your Kit**. Font Squirrel provides a download containing: your font files with the extensions requested, a demo HTML file for each font face style, and a stylesheet from which you can copy and paste the code directly into your own CSS.



Font Squirrel's Font Catalogue

In addition to the `@font-face` generator, the Font Squirrel site includes a catalog of hand-picked free fonts whose licenses allow for web embedding. In fact, both of the fonts we're using on *The HTML5 Herald* can also be found on Font Squirrel, with ready-made `@font-face` kits to download without relying on the generator at all.

To target all browsers, make sure you've created TTF, WOFF, EOT, and SVG font file formats. Once you've created the font files, upload the web fonts to your server. Copy and paste the CSS provided, changing the paths to point to the folder where

you’ve put your fonts. Make sure the font-family name specified in the `@font-face` rule matches the one you’re using in your styles, and you’re good to go!



Troubleshooting @font-face

If your fonts are failing to display in any browser, the problem could very well be the path in your CSS. Check to make sure that the font file is actually where you expect it to be. Browser-based debugging tools—such as the Web Inspector in WebKit, Dragonfly in Opera, or the Firebug Firefox extension—will indicate if the file is missing.

If you’re sure that the path is correct and the file is where it’s supposed to be, make sure your server is correctly configured to serve up the fonts. Windows IIS servers won’t serve up files if they’re unable to recognize their MIME type, so try adding WOFF and SVG to your list of MIME types (EOT and TTF should be supported out of the box):

```
.woff  application/x-font-woff
.svg   image/svg+xml
```

Finally, some browsers require that font files be served from the same domain as the page they’re embedded on.



Browser-based Developer Tools

Safari, Chrome, and Opera all come standard with tools to help save you time as a web developer. Chrome and Opera already have these tools set up. Simply right-click (or control-click on a Mac) and choose **Inspect Element**. A panel will open up at the bottom of your browser, highlighting the HTML of the element you’ve selected. You’ll also see any CSS applied to that element.

While Safari comes with this tool, it needs to be manually enabled. To turn it on, go to **Safari > Preferences**, and then click the **Advanced** tab. Be sure that you check the **Show Develop menu in menu bar** checkbox.

Firefox comes without such a tool. Luckily, there’s a free Firefox plugin called Firebug that provides the same functionality. You can download Firebug at <http://getfirebug.com/>.

Other Considerations

Embedded fonts can improve performance and decrease maintenance time when compared to text as images. Remember, though, that font files can be big. If you need a particular font for a banner ad, it may make more sense (given the limited amount of text required) to simply create an image instead of including font files.

When pondering the inclusion of multiple font files on your site, consider performance. Multiple fonts will increase your site's download time, and font overuse can be tacky. Furthermore, the wrong font can make your content difficult to read. For body text, you should almost always stick to the usual selection of web-safe fonts.

Another factor worth considering is that browsers are unable to render the `@font-face` font until it has been downloaded entirely. They'll behave differently in how they display your content before the download is complete: some browsers will render the text in a system font, while others won't render any text at all.

This effect is referred to as a “flash of unstyled text,” or FOUT, a term coined by Paul Irish.¹⁴ To try to prevent this from happening (or to minimize its duration), make your file sizes as small as possible, Gzip them, and include your `@font-face` rules in CSS files as high up as possible in your markup. If there's a `script` above the `@font-face` declaration in the source, IE experiences a bug, whereby the page won't render *anything* until the font has downloaded—so be sure your fonts are declared above any scripts on your page.

Another option to mitigate `@font-face`'s impact on performance is to defer the font file download until after the page has rendered. This may be unviable for your designer or client, however, as it may result in a more noticeable FOUT, even if the page loads faster overall.¹⁵

Of course, we don't want to scare you away from using `@font-face`, but it's important that you avoid using this newfound freedom to run wild without regard for the consequences. Remember that there are trade-offs, so use web fonts where they're appropriate, and consider the available alternatives.

¹⁴ <http://paulirish.com/2009/fighting-the-font-face-fout/>

¹⁵ For more on `@font-face` and performance, as well as an example of how to “lazy load” your font files, see <http://www.stevesouders.com/blog/2009/10/13/font-face-and-performance/>.

CSS3 Multicolumn Layouts

Nothing says “newspaper” like a row of tightly packed columns of text. There’s a reason for this: newspapers break articles into multiple columns because lines of text that are too long are hard to read. Browser windows can be wider than printed books, and even as wide as some newspapers—so it makes sense for CSS to provide us with the ability to flow our content into columns.

You may be thinking that we’ve always been able to create column effects using the `float` property. But the behavior of floats is subtly different from what we’re after. Newspaper-style columns have been close to impossible to accomplish with CSS and HTML without forcing column breaks at fixed positions. True, you could break an article into `div`s, floating each one to make it look like a set of columns. But what if your content is dynamic? Your back-end code will need to figure out where each column should begin and end in order to insert the requisite `div` tags.

With CSS3 columns, the browser determines when to end one column and begin the next without requiring any extra markup. You retain the flexibility to change the number of columns as well as their width, without having to go back in and alter the page’s markup.

For now, we’re mostly limited to splitting content across a few columns, while controlling their widths and the gutters between them. As support broadens, we’ll be able to break columns, span elements across multiple columns, and more. Support for CSS3 columns is moderate: Firefox and WebKit have had support via vendor-prefixed properties for years, while Opera has just added support in 11.10 (without a vendor prefix), and IE still offers no support.

Almost all the content on the main page of *The HTML5 Herald* is broken into columns. Let’s dig deeper into the properties that make up CSS3 columns and learn how to create these effects on our site.

The `column-count` Property

The `column-count` property specifies the number of columns desired, and the maximum number of columns allowed. The default value of `auto` means that the element has one column. Our leftmost articles are broken into three columns, and the article below the ad blocks has two columns:

css/styles.css (excerpt)

```
#primary article .content {  
  -webkit-column-count: 3;  
  -moz-column-count: 3;  
  column-count: 3;  
}  
  
#tertiary article .content {  
  -webkit-column-count: 2;  
  -moz-column-count: 2;  
  column-count: 2;  
}
```

This is all we really need to create our columns. By default, the columns will have a small gap between them. The total width of the columns combined with the gaps will take up 100% of the width of the element.

Yet, there are a number of other properties we can use for more granular control.

The column-gap Property

The column-gap property specifies the width of the space between columns:

css/styles.css (excerpt)

```
#primary article .content,  
#tertiary article .content {  
  -webkit-column-gap: 10px;  
  -moz-column-gap: 10px;  
  column-gap: 10px;  
}
```

Declare the width in length units, such as `ems` or `pixels`, or use the term `normal`. It's up to the browser to determine what `normal` means, but the spec suggests `1em`. We've declared our gaps to be `10px` wide. The resulting columns are shown in Figure 9.5.



Figure 9.5. Our leftmost content area has articles split over three columns

The column-width Property

The `column-width` property is like having a `min-width` for your columns. The browser will include as many columns of at least the given width as it can to fill up the element—up to the value of the `column-count` property. If the columns need to be wider to fill up all the available space, they will be.

For example, if we have a parent that is 400 pixels wide, a 10-pixel column gap, and the `column-width` is declared as 150px, the browser can fit two columns:

$$(400\text{px width} - 10\text{px column gap}) \div 150\text{px width} = 2.6$$

The browser rounds down to two columns, making columns that are as large as possible in the allotted space; in this case that's 195px for each column—the total width minus the gap, divided by the number of columns. Even if the `column-count` were set to 3, there would still only be two columns, as there's not enough space to include three columns of the specified width. In other words, you can think of the `column-count` property as specifying the *maximum* column count.

The only situation in which columns will be narrower than the `column-width` is if the parent element itself is too narrow for a single column of the specified width. In this case, you'll have one column that fills the whole parent element.

It's a good idea to declare your `column-width` in `ems`, to ensure a minimum number of characters for each line in a column. Let's add a `column-width` of `9em` to our content columns:

css/styles.css (excerpt)

```
#primary article .content,
#tertiary article .content {
  :
  -webkit-column-width: 9em;
  -moz-column-width: 9em;
  column-width: 9em;
}
```

Now, if you increase the font size in your browser, you'll see that the number of columns is decreased as required to maintain a minimum width. This ensures readability, as shown in Figure 9.6.



Figure 9.6. Declaring a `column-width` in `ems` ensures a minimum number of characters on each line

The `columns` Shorthand Property

The `columns` shorthand property is a composite of the `column-width` and `column-count` properties. Declare the two parameters—the width of each column and the number of columns—as described above.

At the time of this writing, this compound property is only supported in WebKit, so you will need to at least continue providing separate properties for the `-moz-` implementation:

css/styles.css (excerpt)

```
#primary article .content {  
  -webkit-columns: 3 9em;  
  -moz-column-count: 3;  
  -moz-column-width: 9em;  
  columns: 3 9em;  
}
```

Rather than specifying different properties for `-webkit-` and `-moz-`, you might find it simpler to just stick with the separate `column-width` and `column-count` properties for now. It's up to you.

Columns and the `height` Property

With the above declarations—and no `height` specified on the element—browsers will balance the column heights automatically, so that the content in each column is approximately equal in height.

But what if a `height` is declared? When the `height` property is set on a multicolumn block, each column is allowed to grow to that height and no further before a new column is added. The browser starts with the first column and creates as many columns as necessary, creating only the first column if there is minimal text. Finally, if too little space is allocated, the content will overflow from the box—or be clipped if `overflow: hidden;` is set.

If you want to declare a `height` on your element, but would also like the content to be spread across your columns, you can use the `column-fill` property. When supported, and set to `balance`, the browser will balance the height of the columns as though there were no `height` declared.



Margins and Padding

Even with a `height` declared, columns may still not appear to have exactly the desired height, because of the bottom margins on paragraphs. WebKit currently splits margins and padding between columns, sometimes adding the extra spacing at the top of a following column. Firefox allows margins to go beyond the bottom of the box, rather than letting them show up at the top of the next column, which we think makes more sense.

As with the `column-width`, you may also want to declare your `height` in `ems` instead of `pixels`; this way, if your user increases the font size, they are less likely to have content clipped or overflowing.

Other Column Features

Beyond the core `count`, `width`, and `gap` properties, CSS3 provides us with a few additional features for laying out multicolumn content, some of which are yet to be supported.

The `column-rule` Property

Column **rules** are essentially borders between each column. The `column-rule` property specifies the color, style, and width of the column rules. The rule will appear in the middle of the column gap. This property is actually shorthand for the `column-rule-color`, `column-rule-style`, and `column-rule-width` properties.

The syntax for the value is exactly the same as for `border` and the related `border-width`, `border-style`, and `border-color` properties. The width can be any length unit, just like `border-width`, including the key terms of `medium`, `thick`, and `thin`. And the color can be any supported color value:

css/styles.css (excerpt)

```
-webkit-column-rule: 1px solid #CCCCCC;  
-moz-column-rule: 1px solid #CCCCCC;  
column-rule: 1px solid #CCCCCC;
```

Column Breaks

There are three column-breaking properties that allow developers to define where column breaks should appear. The `break-before`, `break-after`, and `break-inside`

properties take a limited number of key terms as values to define whether a column break can and should occur before, after, or inside an element, respectively. Rather than being applied to the same element on which we defined our primary column properties, they're applied to other elements nested inside it.

The values available are the same as for `page-break-after`, `page-break-before`, and `page-break-inside` in CSS 2.1: `auto`, `always`, `avoid`, `left`, and `right`. CSS3 also adds a few new possible values for these properties: `page`, `column`, `avoid-page`, and `avoid-column`. The `page` and `column` values function like `always`, and will force a break. The difference is that `page` will only force page breaks and `column` applies only to columns. This gives you a bit more flexibility in how you manage breaks. `avoid-page` and `avoid-column` are similar, except that they function like `avoid`.

For example, you might want to avoid a column break occurring immediately after an `h2` element in your content. Here's how you'd do that:

```
.columns {
  column-count: 3;
  column-gap: 5px;
}

.columns h2 {
  break-after: avoid;
}
```

The only browser engine that currently supports column breaks is WebKit. As well as being vendor-prefixed, the WebKit properties also take a different syntax from what's in the proposed specifications (note the addition of the word `column` to the property names):

```
-webkit-column-break-after: always;
-webkit-column-break-before: auto;
-webkit-column-break-inside: never;
```

Spanning Columns

The `column-span` property will make it possible for an element to span across several columns. If `column-span: all;` is set on an element, all content that comes *before* that element in the markup should be in columns *above* that element. All

content in columns appearing in the markup *after* the element should be in columns *below* the spanned element.

Currently, `column-span` is only supported in WebKit (as `-webkit-column-span`). Because it results in a very different appearance when it's unsupported, it's probably best to avoid using it for now—unless you can be sure that all your visitors will be using WebKit.

For example, for the first article on *The HTML5 Herald*, we could have applied the column properties to the `article` element rather than the `.content` `div`, and used `column-span` to ensure that the video spanned across the full width of the article. However, this would appear badly broken in browsers that support columns but not spanning—like Firefox—so we instead opted to separate the video from the column content.

Other Considerations

If you've been following along with our examples, you might notice that some of your blocks of text have ugly holes in them, like the one shown in Figure 9.7.

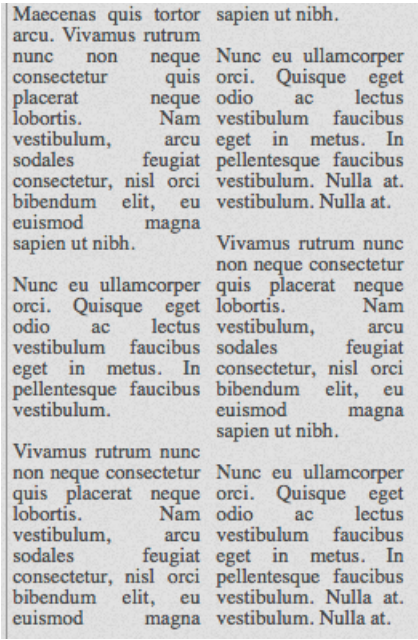


Figure 9.7. “Rivers” can appear in your text when your columns are too narrow

This problem occurs when text with `text-align: justify;` is set in very narrow columns—as we’re doing for *The HTML5 Herald*. This is because browsers don’t know how to hyphenate words in the same way that word processors do, so they space words out awkwardly to ensure that the left and right edges stay justified.

For *The HTML5 Herald*, we’ve used a JavaScript library called Hyphenator¹⁶ to hyphenate words and keep our text looking tidy. This may, however be unnecessary for your site—our columns are extremely narrow, as we’re trying to replicate an old-style newspaper. Few real-world sites would likely need justified columns that narrow, but if you ever come across this issue, it’s good to know that there are solutions available.

Progressive Enhancement

While columns still have limited browser support, there’s no harm including them in your sites unless your designer is a stickler for detail. Columns can be viewed as a progressive enhancement: making long lines easier to read. Those with browsers that lack support for columns will be none the wiser about what they’re missing. For example, *The HTML5 Herald* will have no columns when viewed in Internet Explorer 9, as Figure 9.8 shows—but the site certainly doesn’t look broken, it’s simply adapted to the capabilities of the browser.

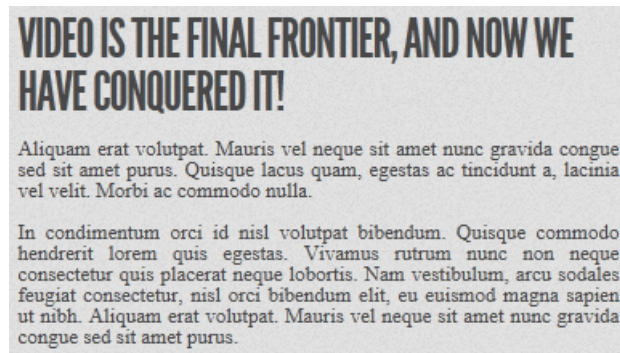


Figure 9.8. Our site has no columns when viewed in IE9—but that’s okay!

¹⁶ <http://code.google.com/p/hyphenator/>

If, however, columns are an important feature of your design, and must be provided to all visitors, there are scripts that can help, such as Columnizer,¹⁷ a jQuery plugin by Adam Wulf.

Media Queries

At this point, we've added a number of CSS3 enhancements to *The HTML5 Herald*. Along the way, we've filled in some knowledge gaps by presenting aspects of CSS3 that were outside the scope of our sample site. So while we're on the topic of columns, it's fitting that we introduce another CSS3 feature that's received much attention among designers targeting audiences on various devices.

In Chapter 1, we talked about the growth rate of mobile devices and the importance of considering the needs of mobile users. With CSS3 media queries, you can do just that—create a layout that resizes to accommodate different screen resolutions.

Media queries are at the heart of a recent design trend called **responsive web design**. This is when all page elements, including images and widgets, are designed and coded to resize and realign seamlessly and elegantly, depending on the capabilities and dimensions of the user's browser.

What are Media Queries?

Before CSS3, a developer could specify a media type for a stylesheet using the `media` attribute. So you might have come across a `link` element that looked like this:

```
<link rel="stylesheet" href="print.css" media="print">
```

Notice that the `media` type is specified as `print`. Acceptable values in addition to `print` include `screen`, `handheld`, `projection`, `all`, and a number of others you'll see less often, if ever. The `media` attribute allows you to specify which stylesheet to load based on the type of device the site is being viewed on. This has become a fairly common method for serving a print stylesheet.

With CSS3's media queries you can, according to the W3C spec, “extend the functionality of media types by allowing more precise labeling of style sheets.” This is done using a combination of media types and expressions that check for the presence

¹⁷ <http://welcome.totheinter.net/columnizer-jquery-plugin/>

of particular media features. So media queries let you change the presentation (the CSS) of your content for a wide variety of devices without changing the content itself (the HTML).

Syntax

Let's use the example from above, and implement a simple media query expression:

```
<link rel="stylesheet" href="style.css" media="screen and (color)">
```

This tells the browser that the stylesheet in question should be used for all screen devices that are in color. Simple—and it should cover nearly everyone in your audience. You can do the same using `@import`:

```
@import url(color.css) screen and (color);
```

Additionally, you can implement media queries using the `@media` at-rule, which we touched on earlier in this chapter when discussing `@font-face`. `@media` is probably the most well-known usage for media queries, and is the method you'll likely use most often:

```
@media handheld and (max-width: 380px) {
  /* styles go here */
}
```

In the example above, this expression will apply to all handheld devices that have a maximum display width of 380 pixels. Any styles within that block will apply only to the devices that match the expression.

Here are a few more examples of media queries using `@media`, so that you can see how flexible and varied the expressions can be. This style will apply only to screen-based devices that have a minimum device width (or screen width) of 320px and a maximum device width of 480px:

```
@media only screen and (min-device-width: 320px) and
➡(max-device-width: 480px) {
  /* styles go here */
}
```

Here's a slightly more complex example:

```
@media only screen and (-webkit-min-device-pixel-ratio: 1.5),
➡only screen and (min-device-pixel-ratio: 1.5) {
  /* styles go here */
}
```

In the above example, we use the `only` keyword, along with the `and` keyword in addition to a comma—which behaves like an `or` keyword. This code will specifically target the iPhone 4's higher resolution display, which could come in handy if you want that device to display a different set of images.

Flexibility of Media Queries

Using the above syntax, media queries allow you to change the layout of your site or application based on a wide array of circumstances. For example, if your site uses a two-column layout, you can specify that the sidebar column drop to the bottom and/or become horizontally oriented, or you can remove it completely on smaller resolutions. On small devices like smartphones, you can serve a completely different stylesheet that eliminates everything except the bare necessities.

Additionally, you can change the size of images and other elements that aren't normally fluid to conform to the user's device or screen resolution. This flexibility allows you to customize the user experience for virtually any type of device, while keeping the most important information and your site's branding accessible to all users.

Browser Support

Support for media queries is very good:

- IE9+
- Firefox 3.5+
- Safari 3.2+
- Chrome 8+
- Opera 10.6+
- iOS 3.2+
- Opera Mini 5+
- Opera Mobile 10+

■ Android 2.1+

The only area of concern is previous versions of Internet Explorer. There are two options for dealing with this: you can supply these versions of IE with a “default” stylesheet that’s served without using media queries, providing a layout suitable for the majority of screen sizes, or you can use a JavaScript-based polyfill. One such ready-made solution can be found at <http://code.google.com/p/css3-mediaqueries-js/>.

Thus, by taking advantage of CSS3 media queries, you can easily create a powerful way to target nearly every device and platform conceivable.

Further Reading

In a book like this, we can’t possibly describe every aspect of media queries. That could be another book in itself—and an important one at that. But if you’d like to look into media queries a little further, be sure to check out the following articles:

- “Responsive Web Design” on *A List Apart*¹⁸
- “How to Use CSS3 Media Queries to Create a Mobile Version of Your Site” on *Smashing Magazine*¹⁹
- For a more critical perspective, “CSS Media Query for Mobile is Fool’s Gold” on the Cloud Four blog²⁰

¹⁸ <http://www.alistapart.com/articles/responsive-web-design/>

¹⁹ <http://www.smashingmagazine.com/2010/07/19/how-to-use-css3-media-queries-to-create-a-mobile-version-of-your-website/>

²⁰ <http://www.cloudfour.com/css-media-query-for-mobile-is-fools-gold/>

Living in Style

We’ve now covered all the new features in CSS that went into making *The HTML5 Herald*—and quite a few that didn’t. While we haven’t covered *everything* CSS3 has to offer, we’ve mastered several techniques that you can use today, and a few that should be usable in the very near future. Remember to check the specifications—as these features are all subject to change—and keep up to date with the state of browser support. Things are moving quickly for a change, which is both a great boon and an additional responsibility for web developers.

Up next, we’ll switch gears to cover some of the new JavaScript APIs. While, as we’ve mentioned, these aren’t strictly speaking part of HTML5 or CSS3, they’re often bundled together when people speak of these new technologies. Plus, they’re a lot of fun, so why not get our feet wet?

Chapter 10

Geolocation, Offline Web Apps, and Web Storage

Much of what is loosely considered to be a part of “HTML5” isn’t, strictly speaking, HTML at all—it’s a set of additional APIs that provide a wide variety of tools to make our websites even better. We introduced the concept of an API way back in Chapter 1, but here’s a quick refresher: an API is an interface for programs. So, rather than a visual interface where a user clicks on a button to make something happen, an API gives your code a virtual “button” to press, in the form of a method it calls that gives it access to a set of functionality. In this chapter, we’ll walk you through a few of the most useful of these APIs, as well as give you a brief overview of the others, and point you in the right direction should you want to learn more.

With these APIs, we can find a visitor’s current location, make our website available offline as well as perform faster online, and store information about the state of our web application so that when a user returns to our site, they can pick up where they left off.



Here There be Dragons

A word of warning: as you know, the P in API stands for Programming—so there'll be some JavaScript code in the next two chapters. If you're fairly new to JavaScript, don't worry! We'll do our best to walk you through how to use these new features using simple examples with thorough explanations. We'll be assuming you have a sense of the basics, but JavaScript *is* an enormous topic. To learn more, SitePoint's *Simply JavaScript* by Kevin Yank and Cameron Adams is an excellent resource for beginners.¹ You may also find the Mozilla Developer Network's JavaScript Guide useful.²

As with all the JavaScript examples in this book so far, we'll be using the jQuery library in the interests of keeping the examples as short and readable as possible. We want to demonstrate the APIs themselves, not the intricacies of writing cross-browser JavaScript code. Again, any of this code can just as easily be written in plain JavaScript, if that's your preference.

Geolocation

The first new API we'll cover is geolocation. Geolocation allows your visitors to share their current location.

Depending on how they're visiting your site, their location may be determined by any of the following:

- IP address
- wireless network connection
- cell tower
- GPS hardware on the device

Which of the above methods are used will depend on the browser, as well as the device's capabilities. The browser then determines the location and passes it back to the Geolocation API. One point to note, as the W3C Geolocation spec states: "No guarantee is given that the API returns the device's actual location."³

Geolocation is supported in:

¹ Melbourne: SitePoint, 2007

² <https://developer.mozilla.org/en/JavaScript/Guide>

³ <http://dev.w3.org/geo/api/spec-source.html#introduction>

- Safari 5+
- Chrome 5+
- Firefox 3.5+
- IE 9+
- Opera 10.6+
- iOS (Mobile Safari) 3.2+
- Android 2.1+

Privacy Concerns

Not everyone will want to share their location with you, as there are privacy concerns inherent to this information. Thus, your visitors must opt in to share their location. Nothing will be passed along to your site or web application unless the user agrees.

The decision is made via a prompt at the top of the browser. Figure 10.1 shows what this prompt looks like in Chrome.

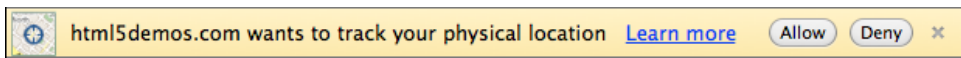


Figure 10.1. Geolocation user prompt



Blocking of the Geolocation Prompt in Chrome

Be aware that Chrome may block your site from showing this prompt entirely if you're viewing your page locally, rather than from an internet server. If this happens, you'll see an icon in the address bar alerting you to it.

There's no way around this at present, but you can either test your functionality in other browsers, or deploy your code to a testing server (this can be a local server on your machine, a virtual machine, or an actual internet server).

Geolocation Methods

With geolocation, you can determine the user's current position. You can also be notified of changes to their position, which could be used, for example, in a web application that provided real-time driving directions.

These different tasks are controlled through the three methods currently available in the Geolocation API:

- `getCurrentPosition`
- `watchPosition`
- `clearPosition`

We'll be focusing on the first method, `getCurrentPosition`.

Checking for Support with Modernizr

Before we attempt to use geolocation, we should ensure that our visitor's browser supports it. We can do that with Modernizr.

We'll start by creating a function called `determineLocation`. We've put it in its own JavaScript file, **`geolocation.js`**, and included that file in our page.

Inside the function, we'll first use Modernizr to check if geolocation is supported:

`geolocation.js` (*excerpt*)

```
function determineLocation() { ❶
  if (Modernizr.geolocation) { ❷
    navigator.geolocation.getCurrentPosition(displayOnMap);
  }
  else {
    // geolocation is not supported in this browser
  }
}
```

Let's examine this line by line:

- ❶ We declare a function called `determineLocation` to contain our location-checking code.
- ❷ We check the Modernizr object's `geolocation` property to see whether geolocation is supported in the current browser. For more information on how the Modernizr object works, consult Appendix A. If geolocation is supported, we continue on to line three, which is inside the `if` statement. If geolocation is unsupported, we move on to the code inside the `else` statement.

Let's assume that geolocation is supported.

Retrieving the Current Position

The `getCurrentPosition` method takes one, two, or three arguments. Here is a summary of the method's definition from the W3C's Geolocation API specification:⁴

```
void getCurrentPosition(successCallback, errorCallback, options);
```

Only the first argument, *successCallback*, is required. *successCallback* is the name of the function you want to call once the position is determined.

In our example, if the location is successfully found, the `displayOnMap` function will be called with a new `Position` object. This `Position` object will contain the current location of the device.



Callbacks

A callback is a function that is passed as an argument to another function. A callback is executed after the parent function is finished. In the case of `getCurrentPosition`, the *successCallback* will only run once `getCurrentPosition` is completed, and the location has been determined.

Geolocation's Position Object

Let's take a closer look at the `Position` object, as defined in the Geolocation API. The `Position` object has two attributes: one that contains the coordinates of the position (`coords`), and another that contains the timestamp of when the position was determined (`timestamp`):

```
interface Position {  
  readonly attribute Coordinates coords;  
  readonly attribute DOMTimeStamp timestamp;  
};
```

⁴ <http://dev.w3.org/geo/api/spec-source.html>



Interfaces

The HTML5, CSS3, and related specifications contain plenty of “interfaces” like the above. These can seem scary at first, but don’t worry. They’re just summarized descriptions of everything that can go into a certain property, method, or object. Most of the time the meaning will be clear—and if not, they’re always accompanied by textual descriptions of the attributes.

But where are the latitude and longitude stored? They’re inside the `Coordinates` object. The `Coordinates` object is also defined in the W3C Geolocation spec, and here are its attributes :

```
interface Coordinates {  
  readonly attribute double latitude;  
  readonly attribute double longitude;  
  readonly attribute double? altitude;  
  readonly attribute double accuracy;  
  readonly attribute double? altitudeAccuracy;  
  readonly attribute double? heading;  
  readonly attribute double? speed;  
};
```

The question mark after `double` in some of those attributes simply means that there’s no guarantee that the attribute will be there. If the browser can’t obtain these attributes, their value will be `null`. For example, very few computers or smartphones contain an altimeter—so most of the time you won’t receive an `altitude` value from a geolocation call. The only three attributes that are guaranteed to be there are `latitude`, `longitude`, and `accuracy`.

`latitude` and `longitude` are self-explanatory, and give you exactly what you would expect: the user’s latitude and longitude. The `accuracy` attribute tells you, in meters, how accurate is the latitude and longitude information.

The `altitude` attribute is the altitude in meters, and the `altitudeAccuracy` attribute is the altitude’s accuracy, also in meters.

The `heading` and `speed` attributes are only relevant if we’re tracking the user across multiple positions. These attributes would be important if we were providing real-time biking or driving directions, for example. If present, `heading` tells us, in degrees,

the direction the user is moving in relation to true north. And `speed`, if present, tells us how quickly the user is moving in meters per second.

Grabbing the Latitude and Longitude

Our `successCallback` is set to the function `displayOnMap`. Here's what this function looks like:

geolocation.js (excerpt)

```
function displayOnMap(position) {
  var latitude = position.coords.latitude;
  var longitude = position.coords.longitude;
  // Let's use Google Maps to display the location
}
```

The first line of our function grabs the `Coordinates` object from the `Position` object that was passed to our callback by the API. Inside the `Coordinates` object is the property `latitude`, which we store inside a variable called `latitude`. We do the same for `longitude`, storing it in the variable `longitude`.

In order to display the user's location on a map, we'll leverage the Google Maps JavaScript API. But before we can use this, we need to add a reference to it in our HTML page. Instead of downloading the Google Maps JavaScript library and storing it on our server, we can point to Google's publicly available version of the API:

geolocation.js (excerpt)

```
:
<!-- google maps API -->
<script type="text/javascript" src="http://maps.google.com/maps/
↪api/js?sensor=true">
</script>
</body>
</html>
```

Google Maps has a *sensor* parameter to indicate whether this application uses a sensor (GPS device) to determine the user's location. That's the `sensor=true` you can see in the sample above. You must set this value explicitly to either `true` or `false`. Because the W3C Geolocation API provides no way of knowing if the information you're obtaining comes from a sensor, you can safely specify `false` for most

web applications (unless they're intended specifically for devices that you know have GPS capabilities, like iPhones).

Loading a Map

Now that we've included the Google Maps JavaScript, we need to, first, add an element to the page to hold the map, and, second, provide a way for the user to call our `determineLocation` method by clicking a button.

To take care of the first step, let's create a fourth box in the sidebar of *The HTML5 Herald*, below the three advertisement boxes. We'll wrap it inside an `article` element, as we did for all the other ads. Inside it, we'll create a `div` called `mapDiv` to serve as a placeholder for the map. Let's also add a heading to tell the user what we're trying to find out:

`index.html` (excerpt)

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

We'll also add a bit of styling to this new HTML:

`css/styles.css` (excerpt)

```
#ad4 h1 {
  font-size: 30px;
  font-family: AcknowledgementMedium;
  text-align: center;
}

#ad4 {
  height: 140px;
}

#mapDiv {
```

```
height: 140px;  
width: 236px;  
}
```

Figure 10.2 reveals what our new sidebar box looks like.

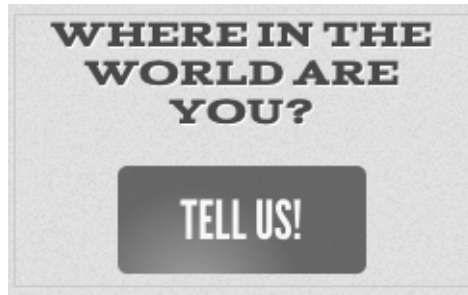


Figure 10.2. The new widget that lets users tell us their location

The second step is to call `determineLocation` when we hit the button. Using jQuery, it's a cinch to attach our function to the button's click event:

[js/geolocation.js \(excerpt\)](#)

```
$( 'document' ).ready(function(){  
    $( '#geobutton' ).click(determineLocation);  
});
```



Document Ready

In the above code snippet, the second line is the one that's doing all the heavy lifting. The `$('document').ready(function(){ ... });` bit is just telling jQuery not to run our code until the page has fully loaded. It's necessary because, otherwise, our code might go looking for the `#geobutton` element before that element even exists, resulting in an error.

This is a very common pattern in JavaScript and jQuery. If you're just getting started with front-end programming, trust us, you'll be seeing a lot of it.

With this code in place, `determineLocation` will be called whenever the button is clicked.

Now, let's return to our `displayOnMap` function and deal with the nitty-gritty of actually displaying the map. First, we'll create a `myOptions` variable to store some of the options that we'll pass to the Google Map:

js/geolocation.js (excerpt)

```
function displayOnMap(position) {  
  var latitude = position.coords.latitude;  
  var longitude = position.coords.longitude;  
  
  // Let's use Google Maps to display the location  
  var myOptions = {  
    zoom: 14,  
    mapTypeId: google.maps.MapTypeId.ROADMAP  
  };  
};
```

The first option we'll set is the zoom level. For a complete map of the Earth, use zoom level 0. The higher the zoom level, the closer you'll be to the location, and the smaller your frame (or viewport) will be. We'll use zoom level 14 to zoom in to street level.

The second option we'll set is the kind of map we want to display. We can choose from the following:

- `google.maps.MapTypeId.ROADMAP`
- `google.maps.MapTypeId.SATELLITE`
- `google.maps.MapTypeId.HYBRID`
- `google.maps.MapTypeId.TERRAIN`

If you've used the Google Maps website before, you'll be familiar with these map types. `ROADMAP` is the default, while `SATELLITE` shows you photographic tiles. `HYBRID` is a combination of `ROADMAP` and `SATELLITE`, and `TERRAIN` will display elements like elevation and water. We'll use the default, `ROADMAP`.



Options in Google Maps

To learn more about Google Maps' options, see the Map Options section of the Google Maps tutorial.⁵

⁵ <http://code.google.com/apis/maps/documentation/javascript/tutorial.html#MapOptions>

Now that we've set our options, it's time to create our map! We do this by creating a new Google Maps object with `new google.maps.Map()`.

The first parameter we pass is the result of the DOM method `getElementById`, which we use to grab the placeholder `div` we put in our `index.html` page. Passing the results of this method into the new Google Map means that the map created will be placed inside that element.

The second parameter we pass is the collection of options we just set. We store the resulting Google Maps object in a variable called `map`:

js/geolocation.js (excerpt)

```
function displayOnMap(position) {  
    var latitude = position.coords.latitude;  
    var longitude = position.coords.longitude;  
  
    // Let's use Google Maps to display the location  
    var myOptions = {  
        zoom: 16,  
        mapTypeId: google.maps.MapTypeId.ROADMAP  
    };  
  
    var map = new google.maps.Map(document.getElementById("mapDiv"),  
    ↪myOptions);
```

Now that we have a map, let's add a marker with the location we found for the user. A marker is the little red drop we see on Google Maps that marks our location.

In order to create a new Google Maps marker object, we need to pass it another kind of object: a `google.maps.LatLng` object—which is just a container for a latitude and longitude. The first new line creates this by calling `new google.maps.LatLng` and passing it the `latitude` and `longitude` variables as parameters.

Now that we have a `google.maps.LatLng` object, we can create a marker. We call `new google.maps.Marker`, and then between two curly braces `{}` we set `position` to the `LatLng` object, `map` to the map object, and `title` to "Hello World!". The title is what will display when we hover our mouse over the marker:

js/geolocation.js (excerpt)

```
function displayOnMap(position) {
    var latitude = position.coords.latitude;
    var longitude = position.coords.longitude;

    // Let's use Google Maps to display the location
    var myOptions = {
        zoom: 16,
        mapTypeId: google.maps.MapTypeId.ROADMAP
    };

    var map = new google.maps.Map(document.getElementById("mapDiv"),
    ↪myOptions);

    var initialLocation = new google.maps.LatLng(latitude, longitude);

    var marker = new google.maps.Marker({
        position: initialLocation,
        map: map,
        title: "Hello World!"
    });
}
```

The final step is to center our map at the initial point, and we do this by calling `map.setCenter` with the `LatLng` object:

```
map.setCenter(initialLocation);
```

You can find a plethora of documentation about Google Maps' JavaScript API, version 3 in the online documentation.⁶

A Final Word on Older Mobile Devices

While the W3C Geolocation API is well-supported in current mobile device browsers, you may need to account for older mobile devices, and support all the geolocation APIs available. If this is the case, you should take a look at the open source library `geo-location-javascript`.⁷

⁶ <http://code.google.com/apis/maps/documentation/javascript/>

⁷ <http://code.google.com/p/geo-location-javascript/>

Offline Web Applications

The visitors to our websites are increasingly on the go. With many using mobile devices all the time, it's unwise to assume that our visitors will always have a live internet connection. Wouldn't it be nice for our visitors to browse our site or use our web application even if they're offline? Thankfully, we can, with Offline Web Applications.

HTML5's Offline Web Applications allows us to interact with websites offline. This initially might sound like a contradiction: a web application exists online by definition. But there are an increasing number of web-based applications that could benefit from being usable offline. You probably use a web-based email client, such as Gmail; wouldn't it be useful to be able to compose drafts in the app while you were on the subway traveling to work? What about online to-do lists, contact managers, or office applications? These are all examples of applications that benefit from being online, but which we'd like to continue using if our internet connection cuts out in a tunnel.

The Offline Web Applications spec is supported in:

- Safari 4+
- Chrome 5+
- Firefox 3.5+
- Opera 10.6+
- iOS (Mobile Safari) 2.1+
- Android 2.0+

It is currently unsupported in all versions of IE.

How It Works: the HTML5 Application Cache

Offline Web Applications work by leveraging what is known as the **application cache**. The application cache can store your entire website offline: all the JavaScript, HTML, and CSS, as well as all your images and resources.

This sounds great, but you may be wondering, what happens when there's a change? That's the beauty of the application cache: your application is automatically updated every time the user visits your page while online. If even one byte of data has changed in one of your files, the application cache will reload that file.



Application Cache versus Browser Cache

Browsers maintain their own caches in order to speed up the loading of websites; however, these caches are only used to avoid having to reload a given file—and not in the absence of an internet connection. Even all the files for a page are cached by the browser. If you try to click on a link while your internet connection is down, you'll receive an error message.

With Offline Web Applications, we have the power to tell the browser which files should be cached or fetched from the network, and what we should fall back to in the event that caching fails. It gives us far more control about how our websites are cached.

Setting Up Your Site to Work Offline

There are three steps to making an Offline Web Application:

1. Create a **cache.manifest** file.
2. Ensure that the manifest file is served with the correct content type.
3. Point all your HTML files to the **cache manifest**.

The HTML5 Herald isn't really an application at all, so it's not the sort of site for which you'd want to provide offline functionality. Yet it's simple enough to do, and there's no real downside, so we'll go through the steps of making it available offline to illustrate how it's done.

The **cache.manifest** File

Despite its fancy name, the **cache.manifest** file is really nothing more than a text file that adheres to a certain format.

Here's an example of a simple **cache.manifest** file:

```
CACHE MANIFEST
CACHE:
index.html
photo.jpg
main.js
```

```
NETWORK:
*
```

The first line of the **cache.manifest** file must read **CACHE MANIFEST**. After this line, we enter **CACHE:**, and then list all the files we'd like to store on our visitor's hard drive. This **CACHE:** section is also known as the explicit section (since we're explicitly telling the browser to cache these files).

Upon first visiting a page with a **cache.manifest** file, the visitor's browser makes a local copy of all files defined in the section. On subsequent visits, the browser will load the local copies of the files.

After listing all the files we'd like to be stored offline, we can specify an **online whitelist**. Here, we define any files that should never be stored offline—usually because they require internet access for their content to be meaningful. For example, you may have a PHP script, **lastTenTweets.php**, that grabs your last ten updates from Twitter and displays them on an HTML page. The script would only be able to pull your last ten tweets while online, so it makes no sense to store the page offline.

The first line of this section is the word **NETWORK**. Any files specified in the **NETWORK** section will always be reloaded when the user is online, and will never be available offline.

Here's what that example online whitelist section would look like:

```
NETWORK
lastTenTweets.php
```

Unlike the explicit section, where we had to painstakingly list every file we wanted to store offline, in the online whitelist section we can use a shortcut: the wildcard *****. This asterisk tells the browser that any files or URLs not mentioned in the explicit section (and therefore not stored in the application cache) should be fetched from the server.

Here's an example of an online whitelist section that uses the wildcard:

```
NETWORK
*
```



All Accounted For

Every URL in your website must be accounted for in the **cache.manifest** file, even URLs that you simply link to. If it's unaccounted for in the manifest file, that resource or URL will fail to load, even if you're online. To avoid this problem, you should use the ***** in the **NETWORK** section.

You can also add comments to your **cache.manifest** file by beginning a line with **#**. Everything after the **#** will be ignored. Be careful to avoid having a comment as the first line of your **cache.manifest** file—as we mentioned earlier, the first line must be **CACHE MANIFEST**. You can, however, add comments to any other line.

It's good practice to have a comment with the version number of your **cache.manifest** file (we'll see why a bit later on):

```
CACHE MANIFEST
# version 0.1
CACHE:
index.html
photo.jpg
main.js

NETWORK:
*
```

Setting the Content Type on Your Server

The next step in making your site available offline is to ensure that your server is configured to serve the manifest files correctly. This is done by setting the content type provided by your server along with the **cache.manifest** file—we discussed content type in the section called “MIME Types” in Chapter 5, so you can skip back there now if you need a refresher.

Assuming you're using the Apache web server, add the following to your **.htaccess** file:

```
AddType text/cache-manifest .manifest
```

Pointing Your HTML to the Manifest File

The final step to making your website available offline is to point your HTML pages to the manifest file. We do that by setting the `manifest` attribute on the `html` element in each of our pages:

```
<!doctype html>
<html manifest="/cache.manifest">
```

Once we've done that, we're finished! Our web page will now be available offline. Better still, since any content that hasn't changed since the page has been viewed will be stored locally, our page will now load much faster—even when our visitors are online.



Do This for Every Page

Each HTML page on your website must set the `manifest` attribute on the `html` element. Ensure you do this, or your application might not be stored in the application cache! While it's true that you should only have one `cache.manifest` file for the entire application, every HTML page of your web application needs `<html manifest="/cache.manifest">`.

Getting Permission to Store the Site Offline

As with geolocation, browsers provide a permission prompt when a website is using a `cache.manifest` file. Unlike geolocation, however, not all browsers are required to do this. When present, the prompt asks the user to confirm that they'd like the website to be available offline. Figure 10.3 shows the prompt's appearance in Firefox.

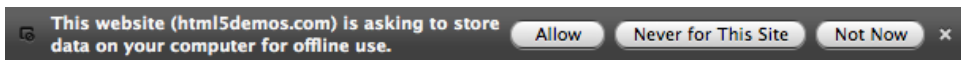


Figure 10.3. Prompt to allow offline web application storage in the app cache

Going Offline to Test

Once we have completed all three steps to make an offline website, we can test out our page by going offline. Firefox and Opera provide a menu option that lets you work offline, so there's no need to cut your internet connection. To do that in Firefox, go to **File > Work Offline**, as shown in Figure 10.4.

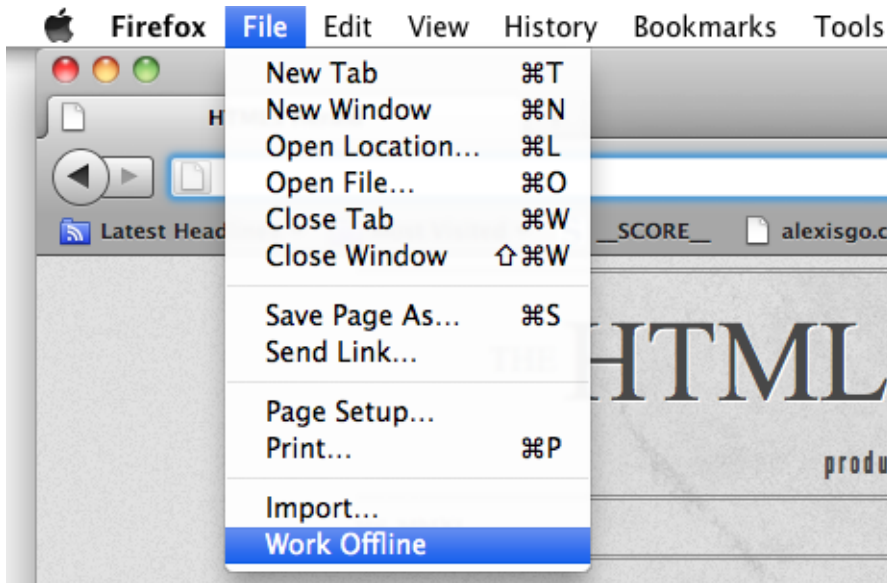


Figure 10.4. Testing offline web applications with Firefox's **Work Offline** mode

While it's convenient to go offline from the browser menu, it's most ideal to turn off your network connection altogether when testing Offline Web Applications.

Testing If the Application Cache Is Storing Your Site

Going offline is a good way to spot-check if our application cache is working, but for more in-depth debugging, we'll need a finer instrument. Fortunately, Chrome's Web Inspector tool has some great features for examining the application cache.

To check if our **cache.manifest** file has the correct content type, here are the steps to follow in Chrome (<http://html5laboratory.com/s/offline-application-cache.html> has a sample you can use to follow along):

1. Navigate to the URL of your home page in Chrome.
2. Open up the Web Inspector (click the wrench icon, then choose **Tools > Developer Tools**).
3. Click on the **Console** tab, and look for any errors that may be relevant to the **cache.manifest** file. If everything is working well, you should see a line that starts with "Document loaded from Application Cache with manifest" and ends

with the path to your **cache.manifest** file. If you have any errors, they will show up in the **Console**, so be on the lookout for errors or warnings here.

4. Click on the **Resources** tab.
5. Expand the **Application Cache** section. Your domain (`www.html5laboratory.com` in our example) should be listed.
6. Click on your domain. Listed on the right should be all the resources stored in Chrome's application cache, as shown in Figure 10.5.

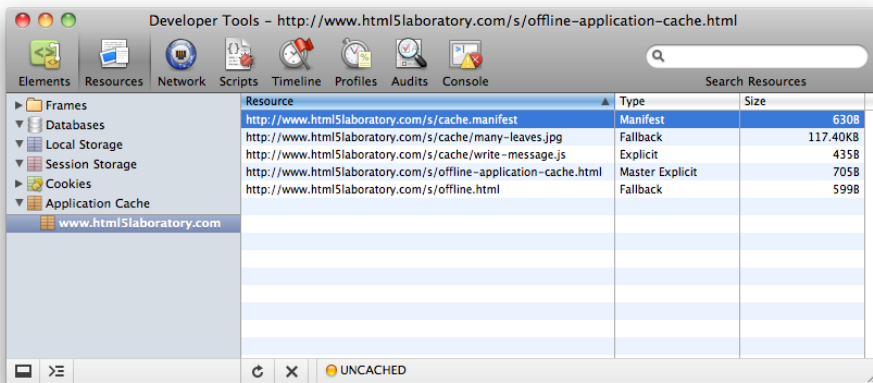


Figure 10.5. Viewing what is stored in Chrome's Application Cache

Making *The HTML5 Herald* Available Offline

Now that we understand the ingredients required to make a website available offline, let's practice what we've learned on *The HTML5 Herald*. The first step is to create our **cache.manifest** file. You can use a program like TextEdit on the Mac or Notepad on Windows to create it, but you have to make sure the file is formatted as plain text. If you're using Windows, you're in luck! As long as you use Notepad to create this file, it will already be formatted as plain text. To format a file as plain text in TextEdit on the Mac, choose **Format > Make Plain Text**. Start off your file by including the line `CACHE MANIFEST` at the top.

Next, we need to add all the resources we'd like to be available offline in the explicit section, which starts with the word `CACHE:`. We must list all our files in this section. Since there's nothing on the site that requires network access (well, there's *one*

thing, but we'll get to that in a bit), we'll just add an asterisk to the **NETWORK** section to catch any files we may have missed in the explicit section.

Here's an excerpt from our **cache.manifest** file:

```
cache.manifest (excerpt)

CACHE MANIFEST
#v1
index.html
register.html

js/hyphenator.js
js/modernizr-1.7.min.js
css/screen.css
css/styles.css
images/bg-bike.png
images/bg-form.png
:
fonts/League_Gothic-webfont.eot
fonts/League_Gothic-webfont.svg
:

NETWORK:
*
```

Once you've added all your resources to the file, save it as **cache.manifest**. Be sure the extension is set to **.manifest** rather than **.txt** or something else.

Then, if you're yet to do so already, configure your server to deliver your manifest file with the appropriate content type.

The final step is to add the **manifest** attribute to the **html** element in our two HTML pages.

We add the **manifest** attribute to both **index.html** and **register.html**, like this:

```
<!doctype html>
<html lang="en" manifest="cache.manifest">
```

And we're set! We can now browse *The HTML5 Herald* at our leisure, whether we have an internet connection or not.

Limits to Offline Web Application Storage

While the Offline Web Applications spec doesn't define a specific storage limit for the application cache, it does state that browsers should create and enforce a storage limit. As a general rule, it's a good idea to assume that you've no more than 5MB of space to work with.

Several of the files we specified to be stored offline are video files. Depending on how large your video files are, it mightn't make any sense to have them available offline, as they could exceed the browser's storage limit.

What can we do in that case? We could place large video files in the **NETWORK** section, but then our users will simply see an unpleasant error when the browser tries to pull the video while offline.

A better alternative is to use an optional section of the **cache.manifest** file: the fallback section.

The Fallback Section

This section allows us to define what the user will see should a resource fail to load. In the case of *The HTML5 Herald*, rather than storing our video file offline and placing it in the explicit section, it makes more sense to leverage the fallback section.

Each line in the fallback section requires two entries. The first is the file for which you want to provide fallback content. You can specify either a specific file, or a partial path like **media/**, which would refer to any file located in the **media** folder. The second entry is what you would like to display in case the file specified fails to load.

If the files are unable to be loaded, we can load a still image of the film's first frame instead. We'll use the partial path **media/** to define the fallback for both video files at once:

cache.manifest (excerpt)

```
FALLBACK:  
media/ images/ford-plane-still.png
```

Of course, this is a bit redundant since, as you know from Chapter 5, the HTML5 `video` element already includes a fallback image to be displayed in case the video fails to load.

So, for some more practice with this concept, let's add another fallback. In the event that any of our pages don't load, it would be nice to define a fallback file that tells you the site is offline. We can create a simple **offline.html** file:

offline.html

```
<!doctype html>
<html lang="en" manifest="/cache.manifest">
  <head>
    <meta charset="utf-8">
    <title>We are offline!</title>
    <link rel="stylesheet" href="css/styles.css?v=1.0"/>
  </head>
  <body>
    <header>
      <h1>Sorry, we are now offline!</h1>
    </header>
  </body>
</html>
```

Now, in the fallback section of our cache manifest, we can specify `/`, which will match any page on the site. If any page fails to load or is absent from the application cache, we'll fall back to the **offline.html** page:

cache.manifest (excerpt)

```
FALLBACK:
media/ images/video-fallback.jpg
/ /offline.html
```



Safari Offline Application Cache Fails to Load Media Files

There is currently a bug in Safari 5 where media files such as **.mp3** and **.mp4** won't load from the offline application cache.

Refreshing the Cache

When using a cache manifest, the files you've specified in your explicit section will be cached until further notice. This can cause headaches while developing: you might change a file and be left scratching your head when you're unable to see your changes reflected on the page.

Even more importantly, once your files are sitting on a live website, you'll want a way to tell browsers that they need to update their application caches. This can be done by modifying the **cache.manifest** file. When a browser loads a site for which it already has a **cache.manifest** file, it will check to see if the manifest file has changed. If it hasn't, it will assume that its existing application cache is all it needs to run the application, so it won't download anything else. If the **cache.manifest** file has changed, the browser will rebuild the application cache by re-downloading all the specified files.

This is why we specified a version number in a comment in our **cache.manifest**. This way, even if the list of files remains exactly the same, we have a way of indicating to browsers that they should update their application cache; all we need to do is increment the version number.

Caching the Cache

This might sound absurd, but your **cache.manifest** file may itself be cached by the browser. Why, you may ask? Because of the way HTTP handles caching.

In order to speed up performance of web pages overall, caching is done by browsers, according to rules set out via the HTTP specification.⁸ What do you need to know about these rules? That the browser receives certain HTTP headers, including **Expire** headers. These **Expire** headers tell the browser when a file should be expired from the cache, and when it needs updating from the server.

If your server is providing the manifest file with instructions to cache it (as is often the default for static files), the browser will happily use its cached version of the file instead for fetching your updated version from the server. As a result, it won't re-download any of your application files because it thinks the manifest has not changed!

⁸ <http://www.w3.org/Protocols/rfc2616/rfc2616-sec13.html>

If you're finding that you're unable to force the browser to refresh its application cache, try clearing the regular browser cache. You could also change your server settings to send explicit instructions not to cache the **cache.manifest** file.

If your site's web server is running Apache, you can tell Apache not to cache the **cache.manifest** file by adding the following to your **.htaccess** file:

```
.htaccess (excerpt)  
  
<Files cache.manifest>  
  ExpiresActive On  
  ExpiresDefault "access"  
</Files>
```

The `<Files cache.manifest>` tells Apache to only apply the rules that follow to the **cache.manifest** file. The combination of `ExpiresActive On` and `ExpiresDefault "access"` forces the web server to always expire the **cache.manifest** file from the cache. The effect is, the **cache.manifest** file will never be cached by the browser.

Are we online?

Sometimes, you'll need to know if your user is viewing the page offline or online. For example, in a web mail app, saving a draft while online involves sending it to the server to be saved in a database; but while offline, you would want to save that information locally instead, and wait until the user is back online to send it to your server.

The offline web apps API provides a few handy methods and events for managing this. For *The HTML5 Herald*, you may have noticed that the page works well enough while offline: you can navigate from the home page to the sign-up form, play the video, and generally mess around without any difficulty. However, when you try to use the geolocation widget we built earlier in this chapter, things don't go so well. This makes sense: without an internet connection, there's no way for our page to figure out your location (unless your device has a GPS), much less communicate with Google Maps to retrieve the map.

Let's look at how we can fix this. We would like to simply provide a message to users indicating that this functionality is unavailable while offline. It's actually very easy; browsers that support Offline Web Applications give you access to the

`navigator.onLine` property, which will be true if the browser is online, and false if it's not. Here's how we'd use it in our `determineLocation` method:

js/geolocation.js (excerpt)

```
function determineLocation(){
  if (navigator.onLine) {
    // find location and call displayOnMap
  } else {
    alert("You must be online to use this feature.");
  }
}
```

Give it a spin. Using Firefox or Opera, first navigate to the page and click the button to load the map. Once you're satisfied that it works, choose Work Offline, reload the page, and try clicking the button again. This time you'll receive a helpful message telling you that you'll need to be online to access the map.

Some other features that might be of use to you include events that fire when the browser goes online or offline. These events fire on the window element, and are simply called `window.online` and `window.offline`. These can, for example, allow your scripts to respond to a change in state by either synchronizing information up to the server when you go online, or saving data locally when you drop offline.

There are a few other events and methods available to you for dealing with the application cache, but the ones we've covered here are the most important. They'll suffice to have most websites and applications working offline without a hitch.

Further Reading

If you would like to learn more about Offline Web Applications, here are a few good resources:

- The WHATWG Offline Web Applications spec⁹
- HTML5 Laboratory's "Using the cache manifest to work offline"¹⁰
- Opera's Offline Application Developer's Guide¹¹

⁹ <http://www.whatwg.org/specs/web-apps/current-work/multipage/offline.html#offline>

¹⁰ <http://www.html5laboratory.com/working-offline.php>

¹¹ <http://dev.opera.com/articles/view/offline-applications-html5-appcache/>

- Peter Lubbers' Slide Share presentation on Offline Web Applications¹²
- Mark Pilgrim's walk-through of Offline Web Applications¹³
- Safari's Offline Applications Programming Guide¹⁴

Web Storage

The Web Storage API defines a standard for how we can save simple data locally on a user's computer or device. Before the emergence of the Web Storage standard, web developers often stored user information in cookies, or by using plugins. With Web Storage, we now have a standardized definition for how to store up to 5MB of simple data created by our websites or web applications. Better still, Web Storage already works in Internet Explorer 8.0!

Web Storage is a great complement to Offline Web Applications, because you need somewhere to store all that user data while you're working offline, and Web Storage provides it.

Web Storage is supported in these browsers:

- Safari 4+
- Chrome 5+
- Firefox 3.6+
- Internet Explorer 8+
- Opera 10.5+
- iOS (Mobile Safari) 3.2+
- Android 2.1+

Two Kinds of Storage

There are two kinds of HTML5 Web Storage: session storage and local storage.

¹² <http://www.slideshare.net/robinzimmermann/html5-offline-web-applications-silicon-valley-user-group>

¹³ <http://diveintohtml5.org/offline.html>

¹⁴ <http://developer.apple.com/library/safari/#documentation/iPhone/Conceptual/SafariJSData-baseGuide/OfflineApplicationCache/OfflineApplicationCache.html>

Session Storage

Session storage lets us keep track of data specific to one window or tab. It allows us to isolate information in each window. Even if the user is visiting the same site in two windows, each window will have its own individual session storage object and thus have separate, distinct data.

Session storage is not persistent—it only lasts for the duration of a user’s session on a specific site (in other words, for the time that a browser window or tab is open and viewing that site).

Local Storage

Unlike session storage, local storage allows us to save persistent data to the user’s computer, via the browser. When a user revisits a site at a later date, any data saved to local storage can be retrieved.

Consider shopping online: it’s not unusual for users to have the same site open in multiple windows or tabs. For example, let’s say you’re shopping for shoes, and you want to compare the prices and reviews of two brands. You may have one window open for each brand, but regardless of what brand or style of shoe you’re looking for, you’re always going to be searching for the same shoe size. It’s cumbersome to have to repeat this part of your search in every new window.

Local storage can help. Rather than require the user to specify again the shoe size they’re browsing for every time they launch a new window, we could store the information in local storage. That way, when the user opens a new window to browse for another brand or style, the results would just present items available in their shoe size. Furthermore, because we’re storing the information to the user’s computer, we’ll be able to still access this information when they visit the site at a later date.



Web Storage is Browser-specific

One important point to remember when working with web storage is that if the user visits your site in Safari, any data will be stored to Safari’s Web Storage store. If the user then revisits your site in Chrome, the data that was saved via Safari will be unavailable. Where the Web Storage data is stored depends on the browser, and each browser’s storage is separate and independent.



Local Storage versus Cookies

Local storage can at first glance seem to play a similar role to HTTP cookies, but there are a few key differences. First of all, cookies are intended to be read on the server side, whereas local storage is only available on the client side. If you need your server-side code to react differently based on some saved values, cookies are the way to go. Yet, cookies are sent along with each HTTP request to your server—and this can result in significant overhead in terms of bandwidth. Local storage, on the other hand, just sits on the user's hard drive waiting to be read, so it costs nothing to use.

In addition, we have significantly more size to store things using local storage. With cookies, we could only store 4KB of information in total. With local storage, the maximum is 5MB.

What Web Storage Data Looks Like

Data saved in Web Storage is stored as key/value pairs.

A few examples of simple key/value pairs:

- key: *name*, value: *Alexis*
- key: *painter*, value: *Picasso*
- key: *email*, value: *info@me.com*

Getting and Setting Our Data

The methods most relevant to Web Storage are defined in an object called `Storage`. Here is the complete definition of `Storage`:¹⁵

```
interface Storage {  
  readonly attribute unsigned long length;  
  DOMString key(in unsigned long index);  
  getter any getItem(in DOMString key);  
  setter creator void setItem(in DOMString key, in any value);  
  deleter void removeItem(in DOMString key);  
  void clear();  
};
```

¹⁵ <http://dev.w3.org/html5/webstorage/#the-storage-interface>

The first methods we'll discuss are `getItem` and `setItem`. We store a key/value pair in either local or session storage by calling `setItem`, and we retrieve the value from a key by calling `getItem`.

If we want to store the data in or retrieve it from session storage, we simply call `setItem` or `getItem` on the `sessionStorage` global object. If we want to use local storage instead, we'd call `setItem` or `getItem` on the `localStorage` global object. In the examples to follow, we'll be saving items to local storage.

When we use the `setItem` method, we must specify both the key we want to save the value under, and the value itself. For example, if we'd like to save the value "6" under the key "size", we'd call `setItem` like this:

```
localStorage.setItem("size", "6");
```

To retrieve the value we stored to the "size" key, we'd use the `getItem` method, specifying only the key:

```
var size = localStorage.getItem("size");
```

Converting Stored Data

Web Storage stores all values as strings, so if you need to use them as anything else, such as a number or even an object, you'll need to convert them. To convert from a string to a numeric value, we can use JavaScript's `parseInt` method.

For our shoe size example, the value returned and stored in the `size` variable will actually be the string "6", rather than the number 6. To convert it to a number, we'll use `parseInt`:

```
var size = parseInt(localStorage.getItem("size"));
```

The Shortcut Way

We can quite happily continue to use `getItem(key)` and `setItem(key, value)`; however, there's a shortcut we can use to save and retrieve data.

Instead of `localStorage.getItem(key)`, we can simply say `localStorage[key]`. For example, we could rewrite our retrieval of the shoe size like this:

```
var size = localStorage["size"];
```

And instead of `localStorage.setItem(key, value)`, we can say `localStorage[key] = value`:

```
localStorage["size"] = 6;
```



There Are No Keys for That!

What happens if you request `getItem` on a key that was never saved? In this case, `getItem` will return `null`.

Removing Items and Clearing Data

To remove a specific item from Web Storage, we can use the `removeItem` method. We pass it the key we want to remove, and it will remove both the key and its value.

To remove *all* data stored by our site on a user's computer, we can use the `clear` method. This will delete all keys and all values stored for our domain.

Storage Limits

Internet Explorer “allows web applications to store nearly 10MB of user data.”¹⁶ Chrome, Safari, Firefox, and Opera all allow for up to 5MB of user data, which is the amount suggested in the W3C spec. This number may evolve over time, as the spec itself states: “A mostly arbitrary limit of five megabytes per origin is recommended. Implementation feedback is welcome and will be used to update this suggestion in the future.” In addition, Opera allows users to configure how much disk space is allocated to Web Storage.

Rather than worrying about how much storage each browser has, a better approach is to test to see if the quota is exceeded before saving important data. The way you test for this is by catching the `QUOTA_EXCEEDED_ERR` exception. Here's one example of how we can do this:

¹⁶ <http://msdn.microsoft.com/en-us/library/cc197062%28VS.85%29.aspx>

```

try
{
    sessionStorage["name"] = "Tabatha";
}
catch (exception)
{
    if (exception == QUOTA_EXCEEDED_ERR)
    {
        // we should tell the user their quota has been exceeded.
    }
}

```



Try/Catch and Exceptions

Sometimes, problems happen in our code. Designers of APIs know this, and in order to mitigate the effects of these problems, they rely on **exceptions**. An exception occurs when something unexpected happens. The authors of APIs can define specific exceptions to be thrown when particular kinds of problems occur. Then, developers using those APIs can decide how they'd like to respond to a given type of exception.

In order to respond to exceptions, we can wrap any code we think may throw an exception in a **try/catch** block. This works the way you might expect: first, you *try* to do something. If it fails with an exception, you can *catch* that exception and attempt to recover gracefully.

To read more about **try/catch** blocks, see the “try...catch” article at the Mozilla Developer Networks’ JavaScript Reference.¹⁷

Security Considerations

Web Storage has what’s known as **origin-based** security. What this means is that data stored via Web Storage from a given domain is only accessible to pages from that domain. It’s impossible to access any Web Storage data stored by a different domain. For example, assume we control the domain `html5isgreat.com`, and we store data created on that site using local storage. Another domain, say, `google.com`, does not have access to any of the data stored by `html5isgreat.com`. Likewise, `html5isgreat.com` has no access to any of the local storage data saved by `google.com`.

¹⁷ <https://developer.mozilla.org/en/JavaScript/Reference/Statements/try...catch>

Adding Web Storage to *The HTML5 Herald*

We can use Web Storage to add a “Remember me on this computer” checkbox to our registration page. This way, once the user has registered, any other forms they may need to fill out on the site in the future would already have this information.

Let’s define a function that grabs the value of the form’s input elements for name and email address, again using jQuery:

js/rememberMe.js (excerpt)

```
function saveData() {  
    var email = $("#email").val();  
    var name = $("#name").val();  
}
```

Here we’re simply storing the value of the email and name form fields, in variables called `email` and `name`, respectively.

Once we have retrieved the values in the two input elements, our next step is to actually save these values to `localStorage`:

js/rememberMe.js (excerpt)

```
function saveData() {  
    var email = $("#email").val();  
    var name = $("#name").val();  
  
    localStorage["name"] = name;  
    localStorage["email"] = email;  
}
```

Let’s also store the fact that the “Remember me” checkbox was checked by saving this information to local storage as well:

js/rememberMe.js (excerpt)

```
function saveData() {  
    var email = $("#email").val();  
    var name = $("#name").val();  
  
    localStorage["name"] = name;
```

```

localStorage["email"] = email;
localStorage["remember"] = "true";
}

```

Now that we have a function to save the visitor's name and email address, let's call it if they check the "Remember me on this computer" checkbox. We'll do this by watching for the `change` event on the checkbox—this event will fire whenever the checkbox's state changes, whether due to a click on it, a click on its label, or a keyboard press:

js/rememberMe.js (excerpt)

```

$('document').ready(function() {
  $('#rememberme').change(saveData);
});

```

Next, let's make sure the checkbox is actually checked, since the `change` event will fire when the checkbox is unchecked as well:

js/rememberMe.js (excerpt)

```

function saveData() {
  if ($("#rememberme").attr("checked"))
  {
    var email = $("#address").val();
    var name = $("#register-name").val();

    localStorage["name"] = name;
    localStorage["email"] = email;
    localStorage["remember"] = "true";
  }
}

```

This new line of code calls the jQuery method `attr("checked")`, which will return `true` if the checkbox is checked, and `false` if not.

Finally, let's ensure that Web Storage is present in our visitor's browser:

js/rememberMe.js (excerpt)

```
function saveData() {
  if (Modernizr.localstorage) {
    if ($("#rememberme").attr("checked"))
    {
      var email = $("#address").val();
      var name = $("#register-name").val();

      localStorage["name"] = name;
      localStorage["email"] = email;
      localStorage["remember"] = "true";
    }
  }
  else
  {
    // no support for Web Storage
  }
}
```

Now we're saving our visitor's name and email whenever the checkbox is checked, so long as local storage is supported. The problem is, we have yet to actually do anything with that data!

Let's add another function to check and see if the name and email have been saved and, if so, fill in the appropriate input elements with that information. Let's also precheck the "Remember me" checkbox if we've set the key "remember" to "true" in local storage:

js/rememberMe.js (excerpt)

```
function loadStoredDetails() {
  var name = localStorage["name"];
  var email = localStorage["email"];
  var remember = localStorage["remember"];

  if (name) {
    $("#name").val(name);
  }
  if (email) {
    $("#email").val(name);
  }
  if (remember == "true") {
```

```

    $("#rememberme").attr("checked", "checked");
  }
}

```

Again, we want to check and make sure Web Storage is supported by the browser before taking these actions:

js/rememberMe.js (excerpt)

```

function loadStoredDetails() {
  if (Modernizr.localstorage) {
    var name = localStorage["name"];
    var email = localStorage["email"];
    var remember = localStorage["remember"];

    if (name) {
      $("#name").val(name);
    }
    if (email) {
      $("#email").val(name);
    }
    if (remember == "true") s{
      $("#rememberme").attr("checked", "checked");
    }
  } else {
    // no support for Web Storage
  }
}

```

As a final step, we call the `loadStoredDetails` function as soon as the page loads:

js/rememberMe.js (excerpt)

```

$('document').ready(function(){
  loadStoredDetails();
  $('#rememberme').change(saveData);
});

```

Now, if the user has previously visited the page and checked “Remember me on this computer,” their name and email will already be populated on subsequent visits to the page.

Viewing Our Web Storage Values with the Web Inspector

We can use the Safari or Chrome Web Inspector to look at, or even change, the values of our local storage. In Safari, we can view the stored data under the **Storage** tab, as shown in Figure 10.6.

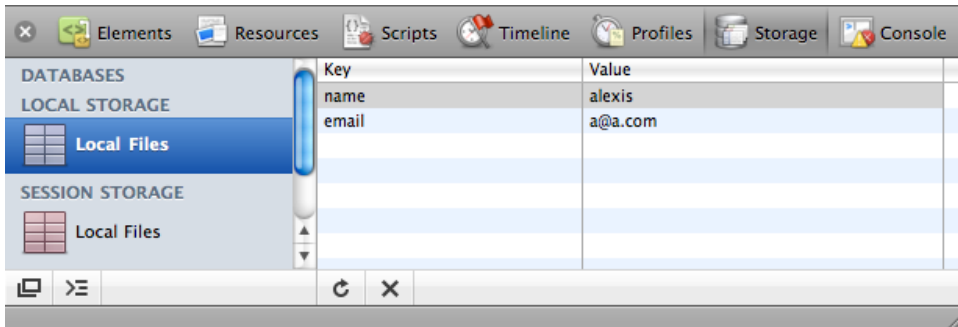


Figure 10.6. Viewing the values stored in local and session storage

In Chrome, the data can be viewed through the **Resources** tab.

Since the user owns any data saved to their hard drive, they can actually modify the data in Web Storage, should they choose to do so.

Let's try this ourselves. If you double-click on the “email” value in Web Inspector's **Storage** tab while viewing the **register.html** page, you can actually modify the value stored there, as Figure 10.7 shows.

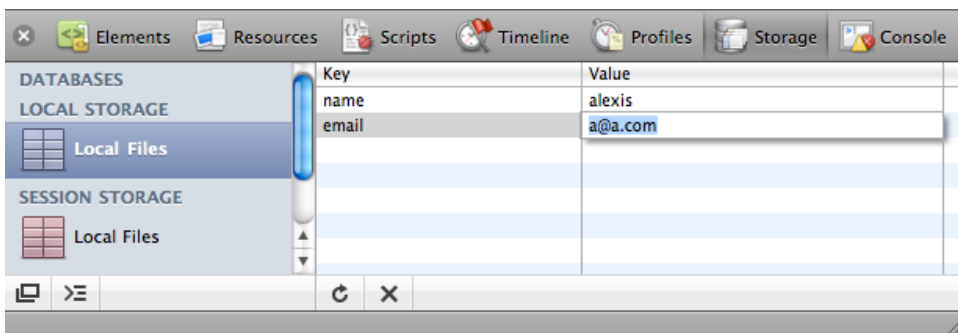


Figure 10.7. Modifying the values stored in Web Storage

There's nothing we as developers can do to prevent this, since our users own the data on their computers. We can and should, however, bear in mind that savvy users have the ability to change their local storage data. In addition, the Web Storage spec states that any dialogue in browsers asking users to clear their cookies should now also allow them to clear their local storage. The message to retain is that we can't be 100% sure that the data we store is accurate, nor that it will always be there. Thus, sensitive data should never be kept in local storage.

If you'd like to learn more about Web Storage, here are a few resources you can consult:

- The W3C's Web Storage specification¹⁸
- The Mozilla Developer Network's Web Storage documentation¹⁹
- Web Storage tutorial from IBM's developerWorks²⁰

Additional HTML5 APIs

There are a number of other APIs that are outside the scope of this book. However, we'd like to mention them briefly here, to give you an overview of what they are, as well as give you some resources should you want to learn more.

Web Workers

The new Web Workers API allows us to run large scripts in the background without interrupting our main page or web app. Prior to Web Workers, it was impossible to run multiple JavaScript scripts concurrently. Have you ever come across a dialogue like the one shown in Figure 10.8?

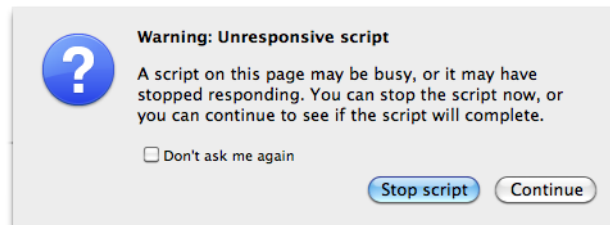


Figure 10.8. A script that runs for too long freezes the whole page

¹⁸ <http://dev.w3.org/html5/webstorage/#the-storage-interface>

¹⁹ <https://developer.mozilla.org/en/DOM/Storage>

²⁰ <http://www.ibm.com/developerworks/xml/library/x-html5mobile2/>

With Web Workers, we should see less of these types of warnings. The new API allows us to take scripts that take a long time to run, and require no user interaction, and run them behind the scenes concurrently with any other scripts that *do* handle user interaction. This concept is known as **threading** in programming, and Web Workers brings us thread-like features. Each “worker” handles its own chunk of script, without interfering with other workers or the rest of the page. To ensure the workers stay in sync with each other, the API defines ways to pass messages from one worker to another.

Web Workers are supported in:

- Safari 4+
- Chrome 5+
- Firefox 3.5+
- Opera 10.6+

Web Workers are currently unsupported in all versions of IE, iOS, and Android.

To learn more about Web Workers, see:

- HTML5 Rocks, “The Basics of Web Workers”²¹
- Mozilla Developer Network, “Using Web Workers”²²
- The W3C Web Workers’ specification²³

Web Sockets

Web Sockets defines a “protocol for two-way communication with a remote host.”²⁴ We’ll skip covering this topic for a couple of reasons. First, this API is of great use to server-side developers, but is less relevant to front-end developers and designers. Second, Web Sockets are still in development and have actually run into some security issues. Firefox 4 and Opera 11 have disabled Web Sockets by default due to these issues.²⁵

²¹ <http://www.html5rocks.com/tutorials/workers/basics/>

²² https://developer.mozilla.org/En/Using_web_workers

²³ <http://dev.w3.org/html5/workers/>

²⁴ <http://www.w3.org/TR/websockets/>

²⁵ See <http://hacks.mozilla.org/2010/12/websockets-disabled-in-firefox-4/> and <http://dev.opera.com/articles/view/introducing-web-sockets/>.

Web Sockets are supported in:

- Safari 5+
- Chrome 4+
- Firefox 4+ (but disabled by default)
- Opera 11+ (but disabled by default)
- iOS (Mobile Safari) 4.2+

Web Sockets are currently unsupported in all versions of IE and on Android.

To learn more about Web Sockets, see the specification at the W3C:

<http://dev.w3.org/html5/websockets/>.

Web SQL and IndexedDB

There are times when the 5MB of storage and simplistic key/value pairs offered by the Web Storage API just aren't enough. If you need to store substantial amounts of data, and more complex relationships between your data, you likely need a full-fledged database to take care of your storage requirements.

Usually databases have been unique to the server side, but there are currently two database solutions proposed to fill this need on the client side: Web SQL and the Indexed Database API (called IndexedDB for short). The Web SQL specification is no longer being updated, and though it currently looks like IndexedDB is gaining steam, it remains to be seen which of these will become the future standard for serious data storage in the browser.

Web SQL is supported in:

- Safari 3.2+
- Chrome
- Opera 10.5+
- iOS (Mobile Safari) 3.2+
- Android 2.1+

Web SQL is currently unsupported in all versions of IE and Firefox. IndexedDB, meanwhile, is currently only supported in Firefox 4.

If you would like to learn more, here are a few good resources:

- Mark Pilgrim’s summary of local storage in HTML5²⁶
- The W3C’s IndexedDB specification²⁷
- The W3C’s Web SQL specification²⁸

Back to the Drawing Board

In this chapter, we’ve had a glimpse into the new JavaScript APIs available in the latest generation of browsers. While these might for some time lack full browser support, tools like Modernizr can help us gradually incorporate them into our real-world projects, bringing an extra dimension to our sites and applications.

In the next—and final—chapter, we’ll look at one more API, as well as two techniques for creating complex graphics in the browser. These open up a lot of potential avenues for creating web apps that leap off the page.

²⁶ <http://diveintohtml5.org/storage.html#future>

²⁷ <http://dev.w3.org/2006/webapi/IndexedDB/>

²⁸ <http://dev.w3.org/html5/webdatabase/>

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.

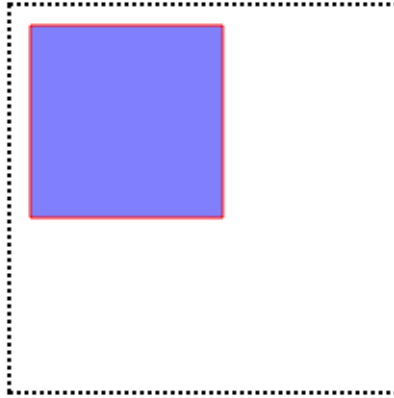


Figure 11.2. A simple rectangle—not bad for our first canvas drawing!

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.

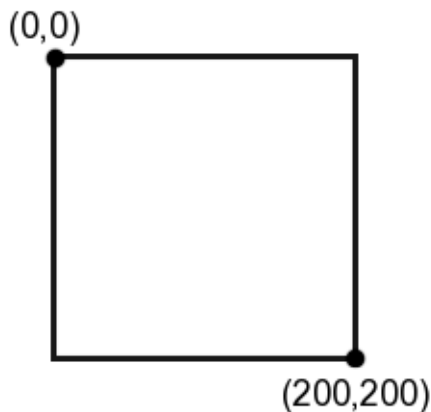


Figure 11.3. The canvas coordinate system goes top-to-bottom and left-to-right

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.

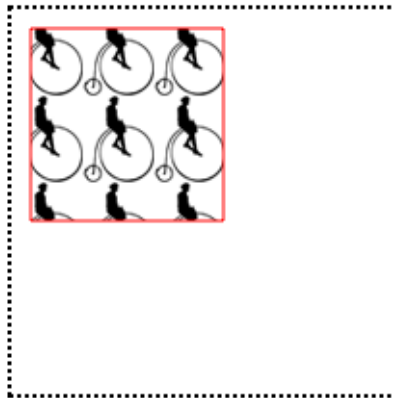


Figure 11.4. A pattern fill on a canvas

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.

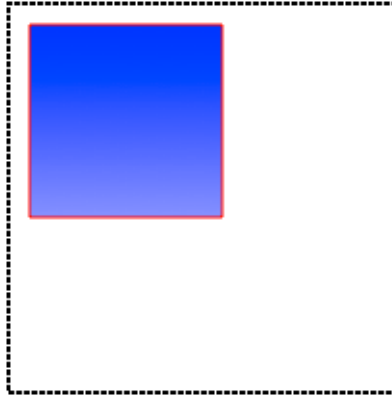


Figure 11.5. Creating a linear gradient with Canvas

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 counter-clockwise 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.closePath();  
}
```

```

    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.

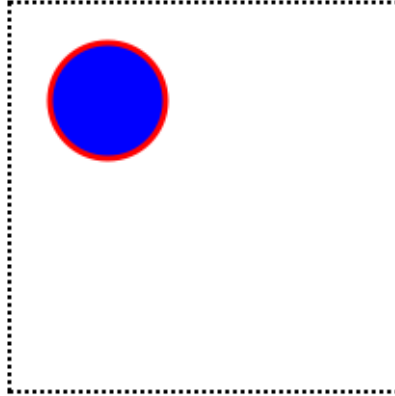


Figure 11.6. Our shiny new circle

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.

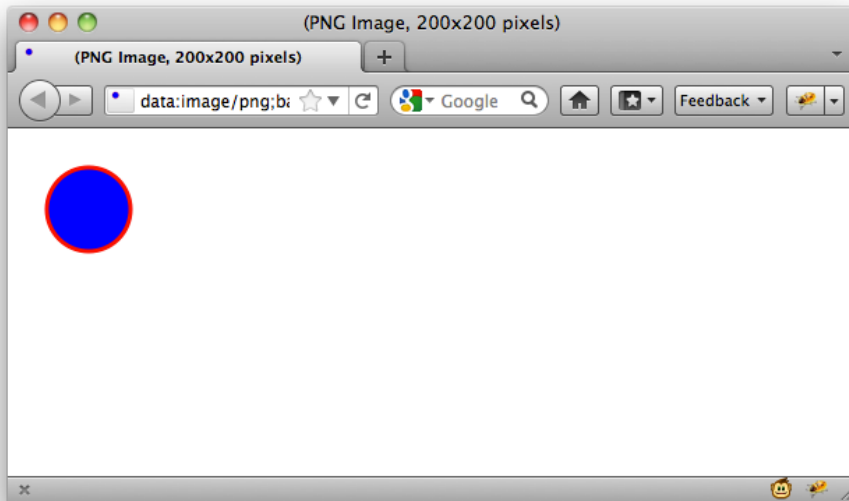


Figure 11.7. Our image loads in a new window

To learn more about saving our canvas drawings as files, see the W3C Canvas spec² 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 logo⁴ 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>

```

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.

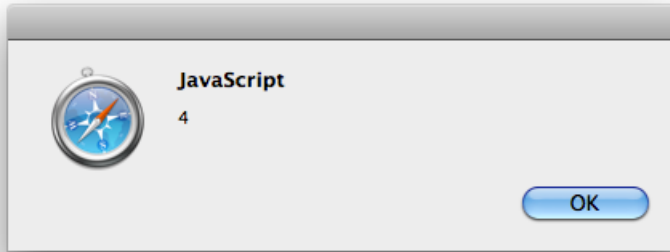


Figure 11.11. The data array for a single pixel contains four values

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 < imageData.length; i += 4) {
    var red = imageData[i];
    var green = imageData[i + 1];
    var blue = imageData[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 < imageData.length; i += 4) {
    var red = imageData[i];
    var green = imageData[i + 1];
    var blue = imageData[i + 2];

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

    imageData[i] = grayscale;
    imageData[i + 1] = grayscale;
    imageData[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 spec⁶ 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.Opera⁸
- Safari's HTML5 Canvas Guide⁹

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* geo-location 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.

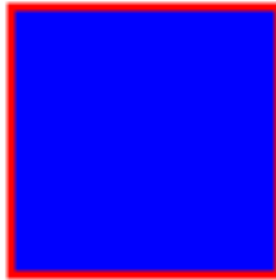


Figure 11.13. A rectangle drawn with SVG

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 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" />
    :
  </g>
</g>
```

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.

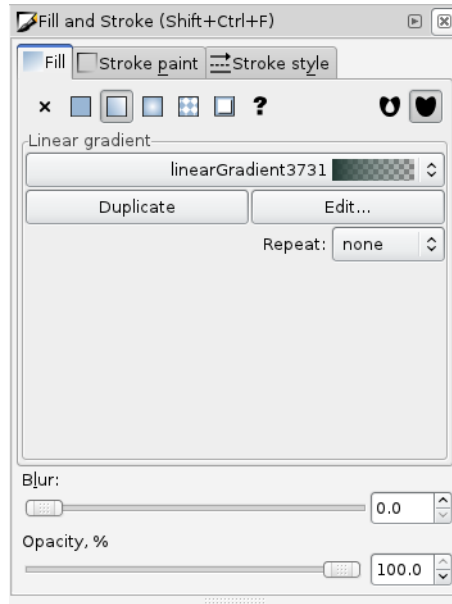


Figure 11.16. Modifying color using **Fill and Stroke**

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

Raphaël¹¹ 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://dmitrybaranovskiy.github.io/raphael/>

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 API¹² 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>

  <div class="content">
    <p id="mouseContainer">
      
      
      
    </p>
  </div>
  :

```

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)

```





```



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>

```

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 documentation¹⁵
- The W3C's Drag and Drop specification¹⁶

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!

Appendix A: Modernizr

Modernizr is an open source JavaScript library that allows us to test for individual features of HTML5 in our users' browsers. Instead of testing just for a particular browser and trying to make decisions based on that, Modernizr allows us to ask specific questions like: "Does this browser support geolocation?" and receive a clear "yes" or "no" answer.

The first step to using Modernizr is to download it from the Modernizr site, at <http://modernizr.com>.

Once you have a copy of the script, you'll need to include the script file in your pages. We'll add it to the head in this example:

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>My Beautiful Sample Page</title>
  <script src="modernizr-1.7.min.js"></script>
</head>
```

You can use Modernizr in two ways: with CSS, and with JavaScript.

Using Modernizr with CSS

When Modernizr runs, it will add an entry in the `class` attribute of the HTML `<html>` tag for every feature it detects, prefixing the feature with `no-` if the browser doesn't support it.

For example, if you're using Safari 5, which supports almost everything in HTML5 and CSS3, your opening `<html>` tag will look a little like this after Modernizr runs:

```
<html class=" js flexbox canvas canvastext no-webgl no-touch
geolocation postmessage websqldatabase no-indexeddb hashchange
history draganddrop websockets rgba hsla multiplebgs backgroundsize
borderimage borderradius boxshadow textshadow opacity cssanimations
csscolumns cssgradients cssreflections csstransforms csstransforms3d
csstransitions fontface video audio localStorage sessionStorage
webworkers applicationcache svg no-inlinesvg smil svgclippaths">
```

Here's what Modernizr adds to the `<html>` tag in Firefox 4:

```
<html class=" js flexbox canvas canvastext webgl no-touch
geolocation postmessage no-websqldatabase indexeddb hashchange
history draganddrop no-websockets rgba hsla multiplebgs
backgroundsize borderimage borderradius boxshadow textshadow opacity
no-cssanimations csscolumns cssgradients no-cssreflections
csstransforms no-csstransforms3d csstransitions fontface video audio
localstorage no-sessionstorage webworkers applicationcache svg
inlinesvg smil svgclippaths">
```

To make better use of this feature of Modernizr, we should first add the class `no-js` to our `html` element in our HTML source:

```
<html class="no-js">
```

Why do we do this? If JavaScript is disabled, Modernizr won't run at all—but if JavaScript *is* enabled, the first thing Modernizr will do is change `no-js` to `js`, as we saw in the Safari 5 and Firefox 4 samples above. This way, you'll have hooks to base your styles on the presence or absence of JavaScript.

You might be thinking, “That sounds pretty cool, but what am I actually supposed to do with this information?” What we can do is use these classes to provide two flavors of CSS: styles for browsers that support certain features, and different styles for browsers that don't.

Because the classes are set on the `html` element, we can use descendant selectors to target any element on the page based on support for a given feature.

Here's an example. Any element with an `id` of `#ad2` that lives inside an element with a class of `cssgradients` (in other words, the `html` element when Modernizr has detected support for gradients) will receive whatever style we specify here:

```
.cssgradients #ad2 {
  /* gradients are supported! Let's use some! */
  background-image:
    -moz-linear-gradient(0% 0% 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%);
    :
}

```

But what if CSS gradients *aren't* supported? We could change the styling to use a simple PNG background image that recreates the same gradient look. Here's how we might do that:

```

.no-cssgradients #ad2 {
    background-image:
        url(../images/put_a_replacement_img_here.png)
}

```

Another way we could use the classes Modernizr adds to the `html` element is with Drag and Drop. We discussed Drag and Drop in Chapter 11, where we added several images of computer mice that can be dragged onto our cat picture to be eaten. These images are all stored in a `div` with an `id` of `mouseContainer`.

But in Opera, where Drag and Drop will fail to work, why even show the mouse images at all? We can use Modernizr to hide the `div` if Drag and Drop is unsupported:

css/styles.css (excerpt)

```

.no-draganddrop #mouseContainer {
    visibility:hidden;
    height:0px;
}

```

If Drag and Drop *is* supported, we simply align all the content in the `div` horizontally:

css/styles.css (excerpt)

```

.draganddrop #mouseContainer {
    text-align:center;
}

```

Using Modernizr with JavaScript

We can also use Modernizr in our JavaScript to provide some fallback if the visitor's browser lacks support for any of the HTML5 elements you use.

When Modernizr runs, as well as adding all those classes to your `<html>` element, it will also create a global JavaScript object that you can use to test for feature support. The object is called, appropriately enough, `Modernizr`. This object contains a property for every HTML5 feature.

Here are a few examples:

```
Modernizr.draganddrop;  
Modernizr.geolocation;  
Modernizr.textshadow;
```

Each property will be either `true` or `false`, depending on whether or not the feature is available in the visitor's browser. This is useful, because we can ask questions like "Is geolocation supported in my visitor's browser?" and then take actions depending on the answer.

Here's an example of using an `if/else` block to test for geolocation support using `Modernizr`:

```
if (Modernizr.geolocation) {  
    // go ahead and use the HTML5 geolocation API,  
    // it's supported!  
}  
else {  
    // There is no support for HTML5 geolocation.  
    // We may try another library, like Google Gears  
    // (http://gears.google.com/), to locate the user.  
}
```

Support for Styling HTML5 Elements in IE8 and Earlier

As mentioned in Chapter 2, IE8 and earlier versions will forbid unrecognized elements to be styled. We then introduced a solution by Remy Sharp, the “HTML5 shiv,” that solves this problem. However, as we alluded to then, Modernizr also solves this problem for us!

As stated on the Modernizr documentation page: “Modernizr runs through a little loop in JavaScript to enable the various elements from HTML5 (as well as abbr) for styling in Internet Explorer. Note that this does not mean it suddenly makes IE support the audio or video element, it just means that you can use `section` instead of `div` and style them in CSS.”

In other words, we now no longer need the HTML5 shiv for styling the new semantic elements in IE8 and below—if we’re using Modernizr anyway, it will take care of that for us.



Location, Location, Location

If you’re using Modernizr instead of the HTML5 shiv, it will need to be placed at the very top of the page. Otherwise, your elements will appear unstyled in IE until the browser has reached the location of the Modernizr script and executed it.

Further Reading

To learn more about Modernizr, see:

- Modernizr documentation: <http://www.modernizr.com/docs/>
- A fairly comprehensive and up-to-date list of polyfills for HTML5 and CSS3 properties that can be used in conjunction with Modernizr is maintained at <https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-browser-Polyfills>
- *A List Apart* article, “Taking Advantage of HTML5 and CSS3 with Modernizr”: <http://www.alistapart.com/articles/taking-advantage-of-html5-and-css3-with-modernizr/>

Appendix B: WAI-ARIA

In Chapter 2 and Chapter 3, we covered considerable ground explaining the potential benefits of using HTML5's new semantic elements for greater accessibility and portability of our pages. Yet, improved semantics alone is sometimes insufficient to make a sophisticated web application fully accessible.

In order to have the content and functionality of our pages as accessible as possible for our users, we need the boost that WAI-ARIA provides, extending what HTML5 (or any markup language) already does.

We'll avoid going into an extensive discussion on WAI-ARIA here—that's a topic that could fill many chapters—but we felt it was important to mention it here so that you're aware of your options.

WAI-ARIA stands for Web Accessibility Initiative-Accessible Rich Internet Applications. The overview of WAI-ARIA on the W3C site explains it as:¹

[...] a way to make Web content and Web applications more accessible to people with disabilities. It especially helps with dynamic content and advanced user interface controls developed with Ajax, HTML, JavaScript, and related technologies.

Users who rely on screen reader technology, or who are unable to use a mouse, are often excluded from using certain website and web application functionality—for example, sliders, progress bars, and tabbed interfaces. With WAI-ARIA, you're able to deal with these shortcomings in your pages—even if the content and functionality is trapped in complex application architecture. Thus, parts of a website that would normally be inaccessible can be made available to users who are reliant on assistive technology.

How WAI-ARIA Complements Semantics

WAI-ARIA assigns **roles** to elements, and gives those roles properties and states. Here's a simple example:

¹ <http://www.w3.org/WAI/intro/aria.php>

```
<li role="menuitemcheckbox" aria-checked="true">Sort by Date</li>
```

The application might be using the list item as a linked element to sort content; yet without the `role` and `aria-checked` attributes, a screen reader would have no way to determine what this element is for. Semantics alone (in this case, a list item) tells it nothing. By adding these attributes, the assistive device is better able to understand what this function is for.

For semantic elements—for example `header`, `h1`, and `nav`—WAI-ARIA attributes are unnecessary, as those elements already express what they are. Instead, they should be used for elements whose functionality and purpose cannot be immediately discerned from the elements themselves.

The Current State of WAI-ARIA

The WAI-ARIA specification is new, as is HTML5, so these technologies are yet to provide all the benefits we would like. Although we’ve described the way that WAI-ARIA can extend the semantics of our page elements, it may be necessary to include WAI-ARIA roles on elements that *already* express their meaning in their names, because assistive technology doesn’t support all the new HTML5 semantics yet. In other words, WAI-ARIA can serve as a sort of stopgap, to provide accessibility for HTML5 pages while the screen readers are catching up.

Let’s look at a site navigation, for example:

```
<nav>
  <ul role="navigation">
    :
  </ul>
</nav>
```

It would seem that we’re doubling up here: the `nav` element implies that the list of links contained within it make up a navigation control, but we’ve still added the WAI-ARIA role `navigation` to the list. Because WAI-ARIA and HTML5 are new technologies, this sort of doubling up will often be necessary: some browsers and screen readers that may lack support for HTML5 *will* have support for WAI-ARIA—and the inverse is possible too.

Does this mean that WAI-ARIA will become redundant once HTML5 is fully supported? No. There are roles in WAI-ARIA without corresponding HTML5 elements; for example, the `timer`² role. While you might represent a timer using the HTML5 `time` element and then update it with JavaScript, you'd have no way of indicating to a screen reader that it was a timer, rather than just an indication of a static time.

For a screen reader to access WAI-ARIA roles, the browser must expose them through an accessibility API. This allows the screen reader to interact with the elements similarly to how it would access native desktop controls.

An article on *A List Apart*, published in late 2010, summarized WAI-ARIA support in browsers and assistive devices by saying:³

Support for some parts of WAI-ARIA [...] is already quite good in later versions of browser and screen readers. However, many problems remain.

Finally, it's worth noting that not all users who could benefit from WAI-ARIA roles are utilizing them. In December, 2010, an organization called WebAIM (Web Accessibility In Mind) conducted their third screen reader user survey,⁴ which revealed that more than 50% of participants didn't use WAI-ARIA features, nor knew they existed.

In short, there is some support for WAI-ARIA—and you won't hurt your HTML5 documents by including these attributes, as they validate in HTML5. Even though the full benefits are yet to be seen, they'll only increase over time.

Further Reading

As mentioned, a full primer on all of the WAI-ARIA roles is beyond the scope of this book, but if you're interested in learning more, we recommend the official specification⁵ first and foremost. The W3C has also put together a shorter Primer⁶ and an Authoring Practices guide.⁷

² <http://www.w3.org/TR/wai-aria/roles#timer>

³ <http://www.alistapart.com/articles/the-accessibility-of-wai-aria/>

⁴ <http://webaim.org/projects/screenreadersurvey3/>

⁵ <http://www.w3.org/TR/wai-aria/>

⁶ <http://www.w3.org/TR/wai-aria-primer/>

⁷ <http://www.w3.org/TR/wai-aria-practices/>

Appendix C: Microdata

Microdata is another technology that's rapidly gaining adoption and support, but unlike WAI-ARIA, it's actually part of HTML5. The Microdata Specification¹ is still early in development, but it's worth mentioning this technology here, because it provides a peek into what may be the future of document readability and semantics.

In the spec, Microdata is defined as a mechanism that “allows machine-readable data to be embedded in HTML documents in an easy-to-write manner, with an unambiguous parsing model.”

With Microdata, page authors can add specific labels to HTML elements to annotate them so that they are able to be read by machines or bots. This is done by means of a customized vocabulary. For example, you might want a script or other third-party service to be able to access your pages and interact with specific elements on the page in a certain manner. With Microdata, you can extend existing semantics (like `article` and `figure`) to allow those services to have specialized access to the annotated content.

This can appear confusing, so let's think about a real-world example. Let's say your site includes reviews of movies. You might have each review in an `article` element, with a number of stars or a percentage score for your review. But when a machine comes along, like Google's search spider, it has no way of knowing which part of your content is the actual review—all it sees is a bunch of text on the page.

Why would a machine want to know what you thought of a movie? It's worth considering that Google has recently started displaying richer information in its search results pages, in order to provide searchers with more than just textual matches for their queries. It does this by reading the review information encoded into those sites' pages using Microdata or other similar technologies. An example of movie review information is shown in Figure C.1.

¹ <http://www.w3.org/TR/microdata/>

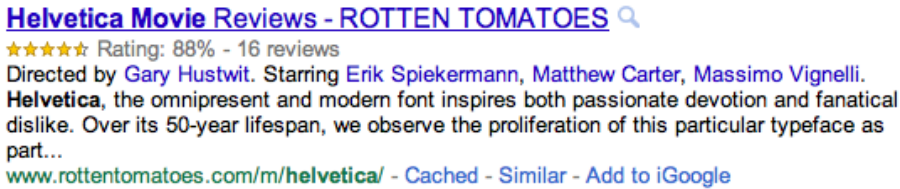


Figure C.1. Google leverages Microdata to show additional information in search results

By using Microdata, you can specify exactly which parts of your page correspond to reviews, people, events, and more—in a consistent vocabulary that software applications can understand and make use of.

Aren't HTML5's semantics enough?

You might be thinking that if a specific element is unavailable using existing HTML, then how useful could it possibly be? After all, the HTML5 spec now includes a number of new elements to allow for more expressive markup. But the creators of HTML5 have been careful to ensure that the elements that are part of the HTML5 spec are ones that will most likely be used.

It would be counterproductive to add elements to HTML that would only be used by a handful of people. This would unnecessarily bloat the language, making it unmaintainable from the perspective of a specification author or standards body.

Microdata, on the other hand, allows you to create your own custom vocabularies for very specific situations—situations that aren't possible using HTML5's semantic elements. Thus, existing HTML elements and new elements added in HTML5 are kept as a sort of semantic baseline, while specific annotations can be created by developers to target their own particular needs.

The Microdata Syntax

Microdata works with existing, well-formed HTML content, and is added to a document by means of name-value pairs (also called **properties**). Microdata does not allow you to create new elements; instead it gives you the option to add customized attributes that expand on the semantics of existing elements.

Here's a simple example:

```

<aside itemscope>
  <h1 itemprop="name">John Doe</h1>
  <p></p>
  <p><a href="http://www.sitepoint.com" itemprop="url">Author's
  ➡website</a></p>
</aside>

```

In the example above, we have a run-of-the-mill author bio, placed inside an `aside` element. The first oddity you'll notice is the `itemscope` attribute. This identifies the `aside` element as the container that defines the **scope** of our Microdata vocabulary. The presence of the `itemscope` attribute defines what the spec refers to as an **item**. Each item is characterized by a group of name-value pairs.

The ability to define the scope of our vocabularies allows us to define multiple vocabularies on a single page. In the example above, all name-value pairs inside the `aside` element are part of a single Microdata vocabulary.

After the `itemscope` attribute, the next item of interest is the `itemprop` attribute, which has a value of `name`. At this point, it's probably a good idea to explain how a script would obtain information from these attributes, as well as what we mean by “name-value pairs.”

Understanding Name-Value Pairs

A name is a property defined with the help of the `itemprop` attribute. In our example, the first property name happens to be one called `name`. There are two additional property names in this scope: `photo` and `url`.

The values for a given property are defined differently, depending on the element the property is declared on. For most elements, the value is taken from its text content; for instance, the `name` property in our example would get its value from the text content between the opening and closing `<h1>` tags. Other elements are treated differently.

The `photo` property takes its value from the `src` attribute of the image, so the value consists of a URL pointing to the author's photo. The `url` property, although defined on an element that has text content (namely, the phrase “Author's website”), doesn't

use this text content to determine its value; instead, it gets its value from the href attribute.

Other elements that don't use their associated text content to define Microdata values include `meta`, `iframe`, `object`, `audio`, `link`, and `time`. For a comprehensive list of elements that obtain their values from somewhere other than the text content, see the Values section of the Microdata specification.²

Microdata Namespaces

What we've described so far is acceptable for Microdata that's not intended to be reused, but that's a little impractical. The real power of Microdata is unleashed when, as we discussed, third-party scripts and page authors can access our name-value pairs and find beneficial uses for them.

In order for this to happen, each item must define a **type** by means of the `itemtype` attribute. Remember that an item in the context of Microdata is the element that has the `itemscope` attribute set. Every element and name-value pair inside that element is part of that item. The value of the `itemtype` attribute, therefore, defines the namespace for that item's vocabulary. Let's add an `itemtype` to our example:

```
<aside itemscope itemtype="http://www.data-vocabulary.org/Person">
  <h1 itemprop="name">John Doe</h1>
  <p></p>
  <p><a href="http://www.sitepoint.com" itemprop="url">Author's
  ➔website</a></p>
</aside>
```

In our item, we're using the URL `http://www.data-vocabulary.org/`, a domain owned by Google. It houses a number of Microdata vocabularies, including Organization, Person, Review, Breadcrumb, and more.

² <http://www.w3.org/TR/microdata/#values>

Further Reading

This brief introduction to Microdata barely does the topic justice, but we hope it will provide you with a taste of what's possible when extending the semantics of your documents with this technology.

It's a very broad topic that requires reading and research outside of this source. With that in mind, here are a few links to check out if you want to delve deeper into the possibilities offered by Microdata:

- “Extending HTML5—Microdata” on HTML5 Doctor³
- The W3C Microdata specification⁴
- Mark Pilgrim’s excellent overview of Microdata⁵
- Google’s Rich Snippets Help⁶

³ <http://html5doctor.com/microdata/>

⁴ <http://www.w3.org/TR/microdata/>

⁵ <http://diveintohtml5.org/extensibility.html>

⁶ <http://www.google.com/support/webmasters/bin/answer.py?hl=en&answer=99170>

Index

Symbols

3D graphics, 269
? (question mark), in attributes, 230
@-rules, 199

A

a element, 47, 54
accessibility
 audio/video, 116
 canvas API and, 294
 hovering and, 124
 nav element and, 28
 required attribute and, 62
 WAI-ARIA, 319–321
Accessible Rich Internet Applications
 (ARIA) (*see* WAI-ARIA)
action attribute, 84
:active pseudo-class, 124
addColorStop method, 274
addEventListener method, 104, 106
adjacent sibling selector, 121
::after pseudo-element, 129
alpha value, 131
animate method, 302
animation property, 195
animation-delay property, 194
animation-direction property, 193–194
animation-duration property, 193
animation-fill-mode property, 194
animation-iteration-count property, 193
animation-name property, 192–193
animation-play-state property, 194–195

animations
 (*see also* Drag and Drop API; trans-
 itions)
 about, 190–191
 browser support, 191
 keyframes, 191–192
 multiple, 195
 properties for, 192–195
 properties shorthand, 195
animation-timing-function property, 193
anonymous functions, 273
APIs (Application Programming Inter-
 face), 5
 about, 2
 for database interaction, 263–264
 for dragging/dropping (*see* Drag and
 Drop API)
 for drawing (*see* Canvas API)
 for geolocation (*see* geolocation API)
 for graphics (*see* SVG)
 for offline site access (*see* offline site
 access)
 for running scripts in background,
 261–262
 for two-way communication, 262–263
 for video (*see* video API)
 for web storage (*see* web storage)
application cache, 237–238
arc method, 276
article element, 27, 31
aside element, 29, 31
async attribute, 52–53
at-rules, 199
attribute selectors, 122–123
attribute values, 22, 23, 54

audio

- browser support, 88, 89–90, 115
- captions, 116
- controls, 115
- muting, 94–95

audio attribute, 94–95

audio codecs, 89

audio element, 115

autocomplete attribute, 70–71, 84

autofocus attribute, 72

autoplay attribute, 92–93, 115

B

b element, 48

background images

(*see also* gradients)

multiple, 169–172

sizing, 172–174

background property, 169, 170, 172

background-color property, 170

background-image property, 169, 171

background-size property, 172–174

backwards selectors, 122

::before pseudo-element, 129

big element, 49

block elements, 35, 47–48, 54, 120, 177

blur distance, 140

bold text, 48

Boolean attributes, 23

border-radius property, 136–139

boxes

basic, 134–136

rounding corners, 136–139

shadows, 140–142

box-shadow property, 140–143

breaks, column, 216–217

browser cache, 238

browser support

about, 8–9

for 3D graphics, 269

for animations, 191

for audio, 88, 89–90, 115

for background sizing, 172–173

for canvas API, 266

character encoding and, 16

for columns, 211, 215, 217, 218, 219–220

for debugging, 209

for Drag and Drop API, 304–305

for File API, 305

for fonts, 201–203, 208

for geolocation, 226–227

graceful degradation, 186

for gradients, 147, 148–150, 164, 165

for HTML5, 19–20, 119–120

for image manipulation, 286–287

for media queries, 222

for obsolete elements, 47

for offline API, 237

for pseudo-classes, 126

for required fields, 60

for RGBA, 132

for rounded corners, 136, 139

for shadows, 141–142, 143, 144

for SVG, 295

for transforms, 175, 177, 182–183

for unrecognized elements, 16–18

for video, 88, 89–90, 97–99

for WAI-ARIA, 321

for Web Sockets API, 263

for Web SQL, 263

for web storage, 250, 251

for Web Workers API, 262

buffered attribute, 115

button input, 60

C

cache, refreshing, 247–248

cache.manifest file, 238–240, 241, 247–248

calculations, displaying, 83

calendar control, 80, 81

callbacks, 229

canplay event, 113

canplaythrough event, 105

canvas API

2D vs 3D, 268–269

about, 265–266

accessibility issues, 294

browser support for, 266

clearing canvas, 291

color, 269–270

color to grayscale, 284–286, 287–290

coordinate system, 271

creating elements, 266–268

displaying text, 290–294

drawing an image, 280–282

drawing circles, 275–278

drawing complex shapes, 278

drawing on canvas, 268

drawing rectangles, 270–271

gradients, 274–275

manipulating an image, 282–284

pattern fills, 271–273

resources, 294

saving drawings, 278–280

security issues, 286–287

setting context, 268–269

SVG vs, 303–304

canvas element, 266–268

CanvasGradient object, 269, 274

CanvasPattern object, 269, 271

CanvasRenderingContext2D object, 268–269

captions, 116

case, upper vs lower, 22, 23, 54

character encoding, 15–16, 20

:checked pseudo-class, 125

child elements, selecting, 126–128

child selector, 121

Chrome

(*see also* browser support)

geolocation prompt, 227

local storage viewing/changing, 260–261

circle element, 296

cite element, 50

clear input, 73

clear method, 254

clearPosition, 228

closePath method, 276

closing tags, 21–22, 23, 54

codecs, 89

collapsible text, 51

color

converting to grayscale, 284–286

fills, 269–270

grayscale conversion, 299–300

HSL notation, 132–133

opacity property, 133–134

pre-CSS3, 131

RGBA notation, 131–132

for shadows, 141

transparency, 270, 291

color input, 60, 79

color picker, 79

color stops, 151–152, 154–155, 159, 274

color-stop function, 154

- column-count property, 211–212
- column-fill property, 215
- column-gap property, 212–213
- Columnizer, 220
- column-rule property, 216
- columns
 - about, 211
 - borders on, 216
 - breaking, 216–217
 - browser support, 211, 215, 217, 218, 219–220
 - gaps between, 212–213
 - height of, 215–216
 - hyphenation in, 218–219
 - shorthand, 215
 - spanning, 217–218
 - specifying number, 211–212
 - width of, 213–214
- columns property, 215
- column-span property, 217–218
- column-width property, 213–214
- compatibility mode, 202–203
- content, semantic types, 35–37
- content-type, setting, 99–100, 240
- context menu, 107–108
- context object, 269
- context, 2D or 3D, 268–269
- controls attribute, 91–92, 115
- cookies, 252
- coordinates
 - geolocation API, 229–231
 - in canvas, 271
 - latitude/longitude, 231–232
- corners
 - asymmetrical, 138
 - rounding, 136–139
- createLinearGradient method, 274

- createPattern method, 272–273
- createRadialGradient method, 274
- CSS3
 - about, 5–7
 - Modernizr and, 313–315
 - older browsers and, 19, 119–120
- CSS3 Sandbox, 161
- CSS3, Please, 161
- cubic-bezier function, 187
- currentSrc attribute, 114
- currentTime property, 110, 111

D

- database APIs, 263–264
- datalist element, 71
- DataTransfer Object, 307–308
- date and time inputs, 79–82
- dates, dynamic, 82
- dates, encoding (*see* time element)
- datetime attribute, 45–46
- datetime input, 80
- datetime-local input, 80
- debugging tools, 242–243
- :default pseudo-class, 125
- defer attribute, 52–53
- definition lists, 50
- deprecated, vs. obsolete, 47
- descendant selector, 121
- description lists, 50
- details element, 51
- determineLocation method, 249
- dialogue, encoding, 50
- disabled attribute, 69
- :disabled pseudo-class, 69, 124
- :disabled pseudo-class, 69, 124
- displayOnMap function, 229, 231
- div element, 24, 25, 26, 31–32

- dl element, 50
- doctype, 14–15, 21
- document outline, 37–39, 40–42
- document ready check, 233
- double colon, 129
- DPI, 174
- Drag and Drop API
 - about, 304–305
 - browser support, 304–305
 - dropping objects, 308–311
 - getting/setting data, 307–308
 - making draggable, 306–307
 - resources, 311
- draggable attribute, 306–307
- dragover event, 308
- draw method, 268
- drawImage method, 281
- drawOneFrame functio, 288
- drop event, 308
- drop shadows, 140–142
- duration attribute, 114
- dynamic dates, 82

E

- elements
 - moving, 176–178
 - resizing, 178–180
 - role attribute, 319–321
 - rotating, 180–181
 - selecting (*see* selectors)
 - (*see also* selectors)
 - skewing, 181
- em element, 49
- email input, 74–75
- embedded content, 36
- :empty pseudo-class, 127
- :enabled pseudo-class, 124

- ended event, 110
- error event, 113
- error handling
 - getImageData, 291
 - QUOTA_EXCEEDED_ERR, 254
 - try/catch, 255, 290
- error messages, customizing, 75
- exceptions, handling, 255
- explicit section, 239
- eXtensible Markup Language (XML), 295

F

- fallback section, 245–246
- figcaption element, 42–43
- figure element, 42–43
- File API, 305
- fillRect method, 270
- fillStyle property, 269
- fillText method, 293
- filters
 - for gradients, 160
 - for shadows, 142
 - in SVG, 299–300
 - for transforms, 183
- Firebug, 209
- Firefox (*see* browser support)
- :first-child pseudo-class, 127
- ::first-letter pseudo-element, 129
- ::first-line pseudo-element, 129
- :first-of-type pseudo-class, 127
- flash of unstyled text (FOUT), 210
- Flash Player plugin, 88, 97
- float property, 211
- flow content, 36
- focus, 60, 72
- focus event, 60
- :focus pseudo-class, 124

- font property descriptors, 203–204
- Font Squirrel, 206–209
- font stack, 205
- fonts
 - using @font-face, 199–200
- font-face rule, 198
- font-family declaration, 200
- fonts
 - browser support, 201–203, 208
 - converting formats, 206–209
 - declaring sources, 200–201
 - file size, 207, 208
 - in canvas, 292
 - licensing issues, 205–206
 - performance issues, 210
 - property descriptors, 203–204
 - setting font stack, 205
 - troubleshooting, 209
 - unicode support, 204
 - web font services, 206
- footer element, 30
- for attribute, 83
- form attribute, 70
- form attributes
 - about, 59–60
 - accessibility, 62
 - autocomplete, 70–71, 84
 - autofocus, 72
 - disabled, 69
 - form, 70
 - list, 71
 - multiple, 69–70
 - pattern, 67–68, 77
 - placeholder, 64–67, 81
 - readonly, 69
 - required, 60–62, 126
- form element, 84

- form elements
 - basic structure, 58–59
 - datalist, 71
 - form, 84
 - HTML5 vs XHTML, 54
 - keygen, 83–84
 - optgroup, 84
 - output, 83
 - textarea, 84–85
- form inputs (*see* input types)
- future-proofing, 10

G

- general sibling selector, 121
- generated content, 129–130
- geolocation API
 - about, 226–227
 - browser support, 226–227
 - checking browser support, 228
 - coordinates available, 229–231
 - getting current position, 229
 - getting latitude/longitude, 231–232
 - getting timestamp, 229
 - loading a map, 232–236
 - methods, 227–228
 - mobile devices, 236
 - privacy issues, 227
- geo-location-javascript library, 236
- getContext method, 269
- getCurrentPosition method, 228, 229
- getImageData method, 283–284, 286–287
- getItem method, 253, 254
- Google Maps, 231, 234–236
- graceful degradation, 186
- Gradient Generator, 161
- gradients
 - about, 147

- images vs., 148
- in canvas, 274–275
- linear (*see* linear gradients)
- radial (*see* radial gradients)
- repeating, 168–169
- grayscale, converting to, 284–286, 287–290, 299–300

H

- h5o, 39
- head element, 15–16
- header element, 24–25, 31
- heading content, 36, 38
- height attribute, 91
- height property, 215–216
- hgroup element, 40–42
- Hickson, Ian, 29
- hidden input, 60
- high attribute, 45
- hints, on forms, 64
- :hover pseudo-class, 124
- HSL notation, 132–133
- .htaccess file, 99–100, 240, 248
- html element, 15
- HTML5
 - about, 1–5
 - basic page structure, 13–19
 - changes to existing features, 47–50
 - choice of new elements, 30
 - content types, 35–37
 - older browsers and, 19–20, 119–120
 - specifications, 3–4
 - validation changes, 53–55
 - XHTML vs, 21–23, 53–55
- HTML5 shim (*see* HTML5 shiv)
- HTML5 shiv, 17–18, 119, 317
- hyphenation, 218–219

I

- i element, 48–49
- image method, 302
- images
 - converting color to grayscale, 284–286
 - drawing, 280–282
 - as fills, 271–273
 - gradients vs., 148
 - manipulating, 282–284, 286–287
 - rotating, 302–303
- implied sections, 38, 40
- :indeterminate pseudo-class, 125
- IndexedDB, 263–264
- Inkscape, 299–300
- inline elements, 35, 36, 177, 179, 180
- input types
 - color, 60, 79
 - date and time, 79–82
 - email address, 74–75
 - full list, 72–73
 - numbers, 76–77
 - phone numbers, 76
 - range, 60, 78
 - search, 73–74
 - URLs, 75–76
- :in-range pseudo-class, 125
- inset shadows, 143
- integer/string conversion, 253
- interactive content, 37
- Internet Explorer
 - (*see also* browser support)
 - font support, 201–203
 - gradient filters, 160
 - HTML5 support, 119–120
 - Modernizr and, 317
 - offline API and, 237

- transforms support, 182–183
- unrecognized elements, 17–18
- :invalid pseudo-class, 63, 125
- :invalid pseudo-class, 63, 125
- italic text, 48–49
- itemtype attribute, 326

J

JavaScript

- (*see also* APIs (Application Programming Interface); Modernizr)
- caching, 103
- collapsible text, 51
- for columns, 220
- default HTML5 styling, 120
- disabled, 18
- embedding, 19
- focus event, 60
- form validation and, 64
- forms, 57
- GoogleMaps, 231–232
- h5o, 39
- html5 shiv, 17–18, 119, 317
- hyphenation, 219
- jQuery library, 65, 308
- placeholder attribute polyfill, 65–67
- Raphaël Library, 301–303
- resources, 226
- running in background, 261–262
- scoped styles, 52
- string/integer conversion, 253
- for transitions, 184
- jQuery library, 65, 308
- JW Player, 98

K

- key/value pairs, 252
- keyframes, 191–192
- keygen element, 83–84
- kind attribute, 116

L

- lang attribute, 15
- :lang pseudo-class, 128
- :last-child pseudo-class, 127
- :last-of-type pseudo-class, 127
- Lawson, Bruce, 26
- line breaks, 84
- linear gradients
 - about, 148
 - browser support, 148–150, 164, 165
 - color stops, 151–152, 154–155, 159
 - direction, 150, 154
 - in Internet Explorer, 160
 - from Photoshop, 156–158
 - stripes, 153
 - SVG files, 158–160
 - tools for creating, 161
 - W3C syntax, 150–154
 - WebKit syntax, 154–155
- linearGradient element, 159
- link element, 16, 20
- lint tools, 55
- list attribute, 71
- lists
 - datalists, 71
 - description/definition, 50
 - ordered, 52
- loadeddata event, 113
- loadedmetadata event, 113
- local storage, 251–252, 260–261

localStorage[key], 253–254

loop attribute, 93, 115

low attribute, 45

M

manifest attribute, 241

mark element, 43–44

max attribute, 44, 45, 77, 78

media queries, 220–223

 browser support, 222

 syntax, 221

meta element, 15

metadata content, 36

metadata, loading video, 113

meter element, 44–45

microdata

 about, 323–324

 adding, 324–326

 item types, 326

 resources, 327

 semantic elements vs, 324

MIME types, 99–100

min attribute, 44, 45, 77, 78

Miro Video Converter, 100

mobile devices, 9–10, 94, 174, 201, 230, 236

 (*see also* offline web applications)

Modernizr

 about, 313

 with CSS, 313–315

 geolocation API and, 228

 HTML5 shiv and, 119

 Internet Explorer and, 317

 with JavaScript, 315–316

 placeholder attribute, 66

 resources, 317

 video API and, 104

month input, 80

multiple attribute, 69–70

muted attribute, 109

muting/unmuting, 94–95, 108–109

N

names, in citations, 50

nav element, 27–29, 320

navigator.onLine property, 248–249

negative delays, 188

NEWT (New Exciting Web Technologies), 3

nightly builds (WebKit), 149

no-js class, 314

normalization (of date/time) (*see* time element)

:not pseudo-class, 128

novalidate attribute, 84

:nth-child pseudo-class, 127

:nth-last-child pseudo-class, 127

:nth-last-of-type pseudo-class, 127

:nth-of-type pseudo-class, 127

number input, 76–77

numeric pseudo-classes, 128

O

obsolete, vs. deprecated, 47

offline site access

 application cache, 237–238, 242–243

 application cache size, 245

 browser support, 237

 cache refresh, 247–248

 cache.manifest file, 238–240, 241, 247–248

 handling load failure, 245–246

 manifest attribute, 241

- prompting for, 241
- resources, 249–250
- setting content type, 240
- testing functionality, 241–243
- testing whether user is online, 248–249
- ol element, 52
- online/offline condition, testing, 248–249
- :only-child pseudo-class, 127
- :only-of-type pseudo-class, 127
- opacity property, 133–134
 - (*see also* transparency)
- Opera (*see* browser support)
- optgroup element, 84
- optimum attribute, 45
- :optional pseudo-class, 126
- ordered lists, 52
- originalEvent method, 308
- origin-based security, 255
- origin-based security issues, 286–287
- outline, document, 37–39, 40–42
- :out-of-range pseudo-class, 126
- output element, 83

P

- page structure, 24–33
- parseInt method, 253
- paths, 275–278
- pattern attribute, 67–68, 77
- pause method, 106
- paused attribute, 105
- persistent storage (*see* local storage)
- phone number input, 76
- phrasing content, 36
- placeholder attribute, 64–67, 81
- play method, 106

- playbackRate attribute, 114
- playing event, 114
- polyfills, 8
- Position object, 229–231
- poster attribute, 94
- prefixes, vendor, 7
- preload attribute, 93–94, 115
- preventDefault method, 309
- progress element, 44
- progress meter, 44, 299–300
- pseudo-classes
 - (*see also* specific pseudo-classes by name, e.g. :visited)
 - attribute or state, 124–126
 - numeric, 128
 - structural, 126–128
- pseudo-elements, 129–130
- pubdate attribute, 46
- putImageData method, 286

Q

- Quartz 2D, 266
- quirks mode, 21
- QUOTA_EXCEEDED_ERR, 254

R

- radial gradients
 - about, 161, 162, 163, 164
 - W3C syntax, 162–164
 - WebKit syntax, 164–166
- range input, 60, 78, 125–126
- Raphaël Library, 301–303
- Raphael method, 301
- readonly attribute, 69
- :read-only pseudo-class, 126
- :read-write pseudo-class, 126

- readyState attribute, 114
- redundant values, 22, 23
- regular expressions, matching, 67–68
- relational selectors, 121–122
- removeItem method, 254
- repeating gradients, 168–169
- required attribute, 60–62, 126
- required fields
 - encoding, 60–62
 - error messages, 62–63
 - styling, 63–64
- :required pseudo-class, 63–64, 126
- :required pseudo-class, 63–64, 126
- Resig, John, 17
- reversed attribute, 52
- reversed sliders, 78
- RGB to grayscale formula, 285
- RGBA notation, 131–132, 270
- rivers, in text, 218–219
- role attribute, 319–321
- :root pseudo-class, 127
- rotate function, 180–181

S

- Safari
 - (*see also* browser support)
 - local storage viewing/changing, 260–261
 - offline file loading, 246
- Scalable Vector Graphics (SVG) (*see* SVG (Scalable Vector Graphics))
- scale function, 178–180
- scoped element, 52
- scoped styles, 52
- screen pixel density, 174
- screen readers (*see* accessibility)
- script attribute, 52–53

- script element, 18–19, 20
- scripts, running in background, 261–262
- search input, 73–74
- section element, 25–27, 31, 32
- sectioning content, 25, 36, 38
- sectioning roots, 39
- sections, implied, 38, 40
- security issues
 - image manipulation, 286–287
 - :visited pseudo-class, 124
 - web storage, 255
- seeked event, 114
- seeking event, 114
- ::selection pseudo-element, 130
- selectors
 - about, 120
 - attribute, 122–123
 - backwards, 122
 - pseudo-classes, 124–126
 - pseudo-elements, 129–130
 - relational, 121–122
 - structural, 126–128
- semantic elements
 - about, 24, 26–27, 30
 - article, 27, 31
 - aside, 29, 31
 - footer, 30
 - header, 24–25, 31
 - microdata vs, 324
 - nav, 27–29, 320
 - section, 25–27, 31, 32
- semantics, 24, 319–321
- session storage, 251, 260–261
- setCustomValidity method, 75, 76
- setItem method, 253
- setTimeout method, 289

- shadows
 - drop, 140–142
 - inset, 143
 - text, 144–145
- sibling selectors, 121
- skew function, 181
- sliders, 78
- small element, 49
- source element, 95–97
- span tags, 44
- spread distance, 140
- src attribute, 90, 95, 114
- src property, 200
- standards mode, 21
- start attribute, 52
- step attribute, 77, 78, 82
- storage object, 252–253
- storage, web (*see* web storage)
- strict validation, 55
- string/integer conversion, 253
- strokeRect method, 270
- strokeStyle property, 269
- strong element, 48
- style element, 52
- styles
 - default HTML5, 120
 - for required fields, 63–64
 - scoped, 52
- stylesheets, linking to, 16
- summary element, 51
- SVG (Scalable Vector Graphics)
 - about, 295
 - browser support, 295
 - canvas vs, 303–304
 - drawing shapes, 296–298
 - filters, 299–300
 - font format, 201

- gradients, 158–160
- Inkscape, 299–300
- Raphaël Library, 301–303
- rotating objects, 302–303
- # (hash), in cache.manifest file, 240
- \$ (dollar sign), in attribute selectors, 123
- * (asterisk)
 - in attribute selectors, 123
 - in cache.manifest file, 240
- :: (double colon), 129
- ^ (caret), in attribute selectors, 123
- | (vertical bar), in attribute selectors, 123
- ~ (tilde), in attribute selectors, 123

T

- target attribute, 54
- :target pseudo-class, 125
- tel input, 76
- text
 - (*see also* fonts)
 - bolding, 48
 - collapsing, 51
 - displaying in canvas, 290–294
 - FOUT effect, 210
 - italicizing, 48–49
 - rotating, 180–181
- text shadows, 144–145
- textarea element, 54, 84–85
- text-shadow property, 144–145
- time element, 45–47
- time input, 80
- timeupdate event, 110–112
- toDataURL method, 278
- track element, 116
- transform-origin property, 182
- transforms
 - about, 175–176

- browser support, 175, 177, 182
- in Internet Explorer, 182–183
- order of, 180
- rotation, 180–181
- scaling, 178–180
- setting origins, 182
- skew, 181
- translations, 176–178
- transition property, 188–189
- transition-delay property, 187–188
- transition-duration property, 186
- transition-property, 184–186
- transitions
 - about, 183–184
 - delaying start of, 187–188
 - duration, 186
 - multiple, 189–190
 - older browsers and, 186
 - pace of, 187
 - properties available, 184–186
 - shorthand for, 188–189
- transition-timing-function, 187
- translate function, 176–178
- transparency
 - animations, 192
 - in canvas, 270, 291
 - colors, 131–132, 133–134
 - opacity property, 133–134
 - shadows and, 142
- try/catch blocks, 255, 290
- two-way communication, 262–263
- type attribute, 16, 19, 95

U

- unicode-range descriptor, 204
- unrecognized elements, 16–18, 317
- url input, 75–76

- user agents, 28
- utf-8, 15

V

- :valid pseudo-class, 63, 125
- :valid pseudo-class, 63, 125
- validation
 - client-side, 57
 - customizing error messages, 75
 - of email addresses, 75
 - of HTML5 files, 53–55
 - of phone numbers, 76
 - of required fields, 60, 62–63
 - of URLs, 75
- value attribute, 44, 45
- vendor prefixes, 7
- video
 - autoplaying, 92–93
 - browser support, 88, 89–90, 97–99
 - captions, 116
 - color to grayscale conversion, 287–290
 - custom controls (*see* video API)
 - disabling "save as", 108
 - encoding for the web, 100
 - licensing issues, 90
 - looping, 93
 - MIME types, 99–100
 - multiple sources, 95–97
 - muting, 94–95
 - native controls, 91–92, 104
 - preloading, 93–94
 - setting dimensions, 91
 - teaser image, 94
 - video element, 90
- video API
 - about, 101
 - addEventListener, 104, 106

- attribute list, 114–115
- canplaythrough event, 105
- custom controls, 101
- events, 113–114
- HTML code, 101–103
- muting/unmuting, 108–109
- playing and pausing, 105–107
- resetting to start, 110
- timer, 110–112
- videoEl variable, 103–104
- video codecs, 89
- video containers, 89
- video conversion, 100, 287–290
- video element, 90, 97–99, 101, 107–108
- video object, 288
- videoEl variable, 103–104
- videoHeight attribute, 115
- videoWidth attribute, 115
- :visited pseudo-class, 124
- volumechange event, 109

W

- W3C, 3–4
- WAI-ARIA, 62, 305, 319–321
- watchPosition, 228
- web font services, 206
- Web Inspector, 242–243, 260–261
- Web Sockets API, 262–263
- Web SQL, 263–264
- web storage
 - across domains, 255
 - browser dependence, 251
 - browser support, 250
 - clearing data, 254
 - converting data, 253
 - data format, 252
 - databases and, 263–264

- exception handling, 255
- getting/setting data, 252–254
- local storage, 260–261
- resources, 261
- size limits, 254–255
- types of, 250–252
- Web Workers API, 261–262
- WebGL, 269
- WebKit browsers, 76, 149
 - (*see also* browser support)
- webkit-gradient property, 154
- week input, 80
- WHATWG (Web Hypertext Application Technology Working Group), 3–4
- whitelist, 239
- width attribute, 91
- window.offline event, 249
- window.online event, 249
- "work offline" option, 241
- wrap attribute, 84

X

- XHTML, vs. HTML5, 21–23, 53–55
- XML (eXtensible Markup Language), 295
- xmlns attribute, 15, 20

Y

- YouTube, 108

What's Next?

Web designers: Prepare to master the ways of the jQuery ninja!



JQUERY: NOVICE TO NINJA

By Earle Castledine & Craig Sharkie

jQuery has quickly become the JavaScript library of choice, and it's easy to see why.

In this easy-to-follow guide, you'll master all the major tricks and techniques that jQuery offers—within hours.

Save 10% with this link:



www.sitepoint.com/launch/customers-only-jquery1

Use this link to save 10% off the cover price of *jQuery: Novice to Ninja*, compliments of the SitePoint publishing team.

“ This book has saved my life! I especially love the “excerpt” indications, to avoid getting lost. JQuery is easy to understand thanks to this book. It's a must-have for your development library, and you truly go from Novice to Ninja!

”

Amanda Rodriguez, USA