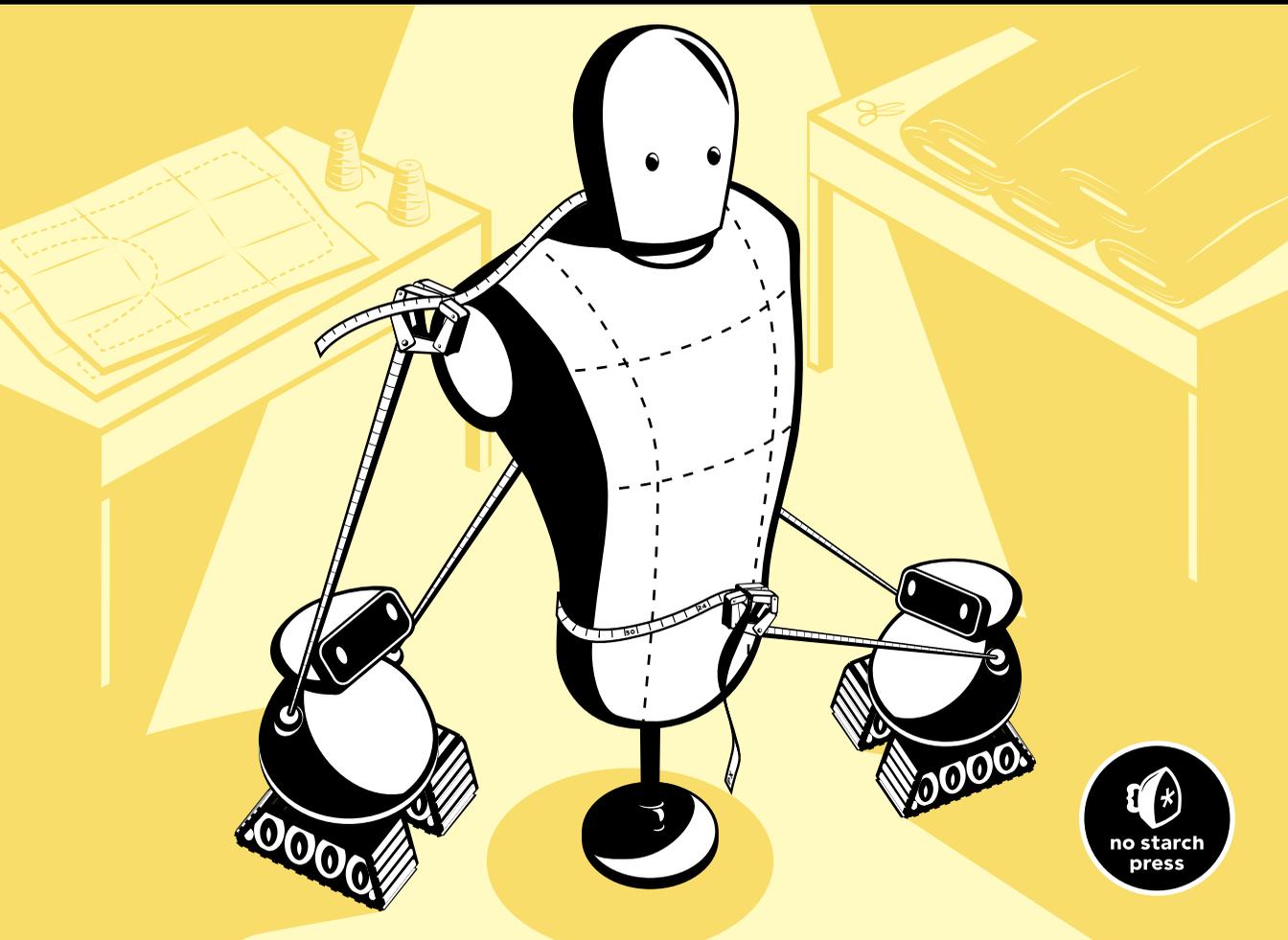


THE BOOK OF CSS3

A DEVELOPER'S GUIDE TO THE
FUTURE OF WEB DESIGN

PETER GASSTON



THE BOOK OF CSS3

THE BOOK OF CSS3

**A Developer's Guide to the
Future of Web Design**

by Peter Gasston



**no starch
press**

San Francisco

THE BOOK OF CSS3. Copyright © 2011 by Peter Gasston

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

Printed in Canada

15 14 13 12 11 1 2 3 4 5 6 7 8 9

ISBN-10: 1-59327-286-3

ISBN-13: 978-1-59327-286-9

Publisher: William Pollock
Production Editor: Serena Yang
Developmental Editor: Keith Fancher
Technical Reviewer: Joost de Valk
Copyeditor: LeeAnn Pickrell
Compositor: Susan Glinert Stevens
Proofreader: Nancy Sixsmith
Indexer: Nancy Guenther

For information on book distributors or translations, please contact No Starch Press, Inc. directly:

No Starch Press, Inc.
38 Ringold Street, San Francisco, CA 94103
phone: 415.863.9900; fax: 415.863.9950; info@nostarch.com; www.nostarch.com

Library of Congress Cataloging-in-Publication Data
A catalog record of this book is available from the Library of Congress.

No Starch Press and the No Starch Press logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

To my wife, Ana, for her patience and support

BRIEF CONTENTS

Foreword by Joost de Valk	xvii
Preface	xix
Introduction	xxi
Chapter 1: Introducing CSS3.....	1
Chapter 2: Media Queries	9
Chapter 3: Selectors	23
Chapter 4: Pseudo-classes and Pseudo-elements.....	33
Chapter 5: Web Fonts	49
Chapter 6: Text Effects and Typographic Styles.....	65
Chapter 7: Multiple Columns	81
Chapter 8: Background Images and Other Decorative Properties	93
Chapter 9: Border and Box Effects	107
Chapter 10: Color and Opacity.....	119
Chapter 11: Gradients	131
Chapter 12: 2D Transformations	147
Chapter 13: Transitions and Animations	163

Chapter 14: 3D Transformations	179
Chapter 15: Flexible Box Layout	195
Chapter 16: Template Layout.....	215
Chapter 17: The Future of CSS	229
Appendix A: CSS3 Support in Current Major Browsers.....	251
Appendix B: Online Resources.....	257
Index	265

CONTENTS IN DETAIL

FOREWORD by Joost de Valk	xvii
----------------------------------	-------------

PREFACE	xix
----------------	------------

INTRODUCTION	xxi
---------------------	------------

The Scope of This Book	xxii
A Quick Note About Browsers and Platforms	xxii
The Appendices and Further Resources	xxiii

1	
INTRODUCING CSS3	1

What CSS3 Is and How It Came to Be	2
A Brief History of CSS3	2
CSS3 Is Modular	2
Module Status and the Recommendation Process	3
CSS3 Is Not HTML5	4
Let's Get Started: Introducing the Syntax	4
Browser-Specific Prefixes	7
Future-Proofing Experimental CSS	8
Getting Started	8

2	
MEDIA QUERIES	9

The Advantages of Media Queries	10
Syntax	11
Media Features	12
Width and Height	13
Device Width and Height	15
Using Media Queries in the Real World	16
Orientation	17
Aspect Ratio	18
Pixel Ratio	19
Multiple Media Features	20
Mozilla-Specific Media Features	21
Summary	21
Media Queries: Browser Support	21

3	
SELECTORS	23

Attribute Selectors	24
New Attribute Selectors in CSS3	25
Beginning Substring Attribute Value Selector	25
Ending Substring Attribute Value Selector	27

Arbitrary Substring Attribute Value Selector	28
Multiple Attribute Selectors	29
The General Sibling Combinator	30
Summary	31
Selectors: Browser Support	31

4 PSEUDO-CLASSES AND PSEUDO-ELEMENTS 33

Structural Pseudo-classes	34
The nth-* Pseudo-classes	35
first-of-type, last-child, and last-of-type	40
only-child and only-of-type	41
Other Pseudo-classes	42
target	42
empty	44
root	44
not	44
UI Element States	45
Pseudo-elements	46
The selection pseudo-element	47
Summary	48
DOM and Attribute Selectors: Browser Support	48

5 WEB FONTS 49

The @font-face Rule	50
Defining Different Faces	51
True vs. Artificial Font Faces	53
A “Bulletproof” @font-face Syntax	54
Using Local Fonts	54
Font Formats	55
The Final “Bulletproof” Syntax	56
The Fontspring Bulletproof Syntax	56
Licensing Fonts for Web Use	57
A Real-World Web Fonts Example	57
More Font Properties	59
font-size-adjust	59
font-stretch	60
OpenType Features	61
Summary	63
Web Fonts: Browser Support	64

6 TEXT EFFECTS AND TYPOGRAPHIC STYLES 65

Understanding Axes and Coordinates	66
Applying Dimensional Effects: text-shadow	67
Multiple Shadows	70
Letterpress Effect	71
Adding Definition to Text: text-outline and text-stroke	72

More Text Properties	73
Restricting Overflow	73
Resizing Elements	74
Aligning Text	75
Wrapping Text	76
Setting Text Rendering Options	77
Applying Punctuation Properties	79
Summary	80
Text Effects: Browser Support	80

7

MULTIPLE COLUMNS 81

Column Layout Methods	82
Prescriptive Columns: column-count	82
Dynamic Columns: column-width	83
A Note on Readability	84
Different Distribution Methods in Firefox and WebKit	86
Combining column-count and column-width	87
Column Gaps and Rules	88
Containing Elements within Columns	90
Elements Spanning Multiple Columns	91
Elements Breaking over Multiple Columns	91
Summary	92
Multiple Columns: Browser Support	92

8

BACKGROUND IMAGES AND OTHER DECORATIVE PROPERTIES 93

Background Images	94
Multiple Background Images	94
Background Size	96
Background Clip and Origin	98
background-repeat	102
Background Image Clipping	103
Image Masks	104
Summary	106
Background Images: Browser Support	106

9

BORDER AND BOX EFFECTS 107

Giving Your Borders Rounded Corners	108
border-radius Shorthand	109
Differences in Implementation Across Browsers	111
Using Images for Borders	111
Multicolored Borders	114
Adding Drop Shadows	115
Summary	117
Border and Box Effects: Browser Support	117

10	COLOR AND OPACITY	119
Setting Transparency with the opacity Property		120
New and Extended Color Values		122
The Alpha Channel		122
Hue, Saturation, Lightness		125
HSLA		127
The Color Variable: currentColor		127
Matching the Operating System's Appearance		129
Summary		130
Color and Opacity: Browser Support		130

11	GRADIENTS	131
Linear Gradients		132
Linear Gradients in Firefox		132
Linear Gradients in WebKit		133
Using Linear Gradients		134
Adding Extra color-stop Values		135
Radial Gradients		136
Radial Gradients in Firefox		137
Radial Gradients in WebKit		137
Using Radial Gradients		138
Multiple color-stop Values		140
The WebKit Advantage		141
Multiple Gradients		141
Repeating Gradients in Firefox		142
Repeating Linear Gradients		143
Repeating Radial Gradients		144
Summary		145
Gradients: Browser Support		146

12	2D TRANSFORMATIONS	147
The transform Property		148
rotate		149
Position in Document Flow		149
transform-origin		150
translate		152
skew		153
scale		154
Multiple Transformations		156
Transforming Elements with Matrices		156
Reflections with WebKit		159
Summary		161
2D Transformations: Browser Support		161

13	TRANSITIONS AND ANIMATIONS	163
Transitions	164	
Property	165	
Duration	165	
Timing Function	166	
Delay	168	
Shorthand	169	
The Complete Transition Example	169	
Multiple Transitions	170	
Triggers	171	
More Complex Animations	172	
Key Frames	172	
Animation Properties	173	
The Complete Animations Example	177	
Multiple Animations	177	
Summary	178	
Transitions and Animations: Browser Support	178	

14	3D TRANSFORMATIONS	179
3D Elements in CSS	180	
Transform Style	182	
The Transformation Functions	182	
Rotation Around an Axis	183	
Translation Along the Axis	185	
Scaling	186	
The Transformation Matrix	187	
Perspective	188	
The perspective and perspective-origin Properties	190	
The Transformation Origin	191	
Showing or Hiding the Backface	193	
Summary	194	
3D Transformations: Browser Support	194	

15	FLEXIBLE BOX LAYOUT	195
Triggering the Flexible Box Layout	196	
The box Value in Firefox	197	
Inline Boxes	198	
Making the Boxes Flexible	199	
Unequal Ratios	201	
Zero Values and Firefox Layouts	202	
Grouping Flexible Boxes	203	
Changing Orientation	204	
Changing the Order of Flexible Boxes	205	
Reversing the Order	205	
Further Control over Ordering	206	

Alignment	208
Same-Axis Alignment	209
Multiple Rows or Columns	211
Cross-Browser Flex Box with JavaScript	211
Stop the Presses: New Syntax	212
Summary	212
Flexible Box Layout: Browser Support	213

16

TEMPLATE LAYOUT 215

Setting Up the JavaScript	216
Using position and display to Create Rows	216
Multiple Rows	219
Slots and the ::slot() Pseudo-element	220
Creating Empty Slots	223
Setting Height and Width on Rows and Columns	223
Width Keyword Values	225
Setting Both Row Height and Column Width	225
Default Content: The @ Sign	226
Summary	228
Template Layout: Browser Support	228

17

THE FUTURE OF CSS 229

Mathematical Operations	230
Calculation Functions	230
Cycle	233
The Grid Positioning Module	233
Implicit and Explicit Grids	234
The Grid Unit (gr)	236
Extended Floats	237
Extending the Possibilities of Images	237
Image Fallback	238
Image Slices	238
Image Sprites	239
Grouping Selectors	241
Constants and Variables	242
WebKit CSS Extensions	245
CSS Variables	245
Extending Variables Using Mixins	245
CSS Modules	246
Nested Rules	247
Haptic Feedback	248
Summary	248
Future CSS: Browser Support	249

A		
CSS3 SUPPORT IN CURRENT MAJOR BROWSERS		251
Media Queries (Chapter 2)	252
Selectors (Chapter 3)	252
Pseudo-classes and Pseudo-elements (Chapter 4)	252
Web Fonts (Chapter 5)	252
Text Effects and Typographic Styles (Chapter 6)	253
Multiple Columns (Chapter 7)	253
Background Images and Other Decorative Properties (Chapter 8)	253
Border and Box Effects (Chapter 9)	254
Color and Opacity (Chapter 10)	254
Gradients (Chapter 11)	254
2D Transformations (Chapter 12)	254
Transitions and Animations (Chapter 13)	255
3D Transformations (Chapter 14)	255
Flexible Box Layout (Chapter 15)	255
Template Layout (Chapter 16)	255
The Future of CSS (Chapter 17)	255

B		
ONLINE RESOURCES		257
CSS Modules	257
Browsers	258
WebKit	258
Firefox	258
Opera	259
Internet Explorer	259
Browser Support	259
When Can I Use	259
Quirks Mode	259
Find Me By IP	260
Feature Detection and Simulation	260
Perfection Kills	260
Modernizr	260
CSS3 Pie	261
Code-Generation Tools	261
CSS3, Please!	261
CSS3 Generator	261
CSS3 Gradient Generator	261
Type Folly	262
Web Fonts	262
Typekit	262
Fontdeck	262
Fonts.com Web Fonts	263
Google Font API	263
Web FontFonts	263
Font Squirrel	263
Fontspring	264

Other Resources	264
CSS3.info	264
CSS3 Watch	264
CSS3 Cheat Sheet	264

INDEX

265

FOREWORD

CSS3 used to be a topic for people who were in it for the long haul. Back in 2006, I started CSS3.info, and Peter joined me in writing posts about the development of the standard and real-life examples of what it looked like in browsers. Although I started the site, Peter was always the most prolific writer, and it's only fitting that while I wrote this foreword, he wrote the book.

CSS3 has finally gone mainstream. With the new age of browsers (such as Firefox 4, Google Chrome, and Internet Explorer 9), we as a web design community are finally getting the power and flexibility we've been waiting for. We can now manage media queries for different browsers, have smarter background images, and handle fonts in a way that doesn't drive us nuts.

If you plan on using CSS3, this book is the most hands-on guide you'll find. It shows you what works and what doesn't, and no caveat is forgotten. Peter even provides a clear explanation for how transitions and transformations work. This is no small feat; as you'll see for yourself when reading those chapters, the matrix functions are not for every user. Luckily you won't have to use those when you're taking advantage of the other—far more accessible—functions in CSS3.

More is to come: CSS3 is an ever-expanding standard that promises to help web designers do great things. I, for one, am very curious about where it will lead us. For now, though, this book is all you need to start uncovering the treasures within CSS3.

Joost de Valk
CEO and Founder, Yoast.com

PREFACE

This book is the culmination of five years' writing about CSS3, both on the Web and in print. The browser and CSS landscape has changed a lot in that short time and continues to change today, bringing new features and implementations at a rate that's difficult to keep up with. The CSS3 specification is written in (often dense) technical language that's intended for implementers rather than end users, and my intent in writing this book was to bridge the gap between specification and web developer.

I wrote about the CSS properties in the earlier chapters of this book with certainty, because they're well implemented and used on a daily basis. As I progressed through the book, I was able to learn more from experimentation and the work of pioneers and early adopters. By the final few chapters I had to rely on interpretation of the CSS3 specification to explain how future properties will behave. I would hope that there are few mistakes, but I accept that any that exist are based on my own misunderstanding.

In addition to the CSS3 specification itself, an invaluable resource was the Mozilla Developer Network (<https://developer.mozilla.org/>), a peerless collection of articles about anything web related—not least CSS—which is all the more amazing for being written by volunteers. The text used in many of

the code examples is taken from books in the public domain which are available at <http://www.gutenberg.org/>. All images in the book that are not my own creations are credited in the relevant chapters.

This book would not have been possible without the guidance of the team at No Starch Press, especially Serena Yang and my editor, Keith Fancher, who made me write more clearly and helped me transition from blogger to author. I'd also like to thank Joost de Valk, who not only acted as my technical editor but also gave me my first opportunity to write about CSS3 when he created the website <http://www.css3.info/> five years ago.

I'd also like to thank my colleagues at Preloaded and Poke for their support and encouragement, everyone at the many London web community meet-ups, my mum for teaching me the value of hard work, and my dad for buying me my first computer some almost thirty years ago—I promised I'd pay him back one day, and hopefully this book will go some way toward that debt.

INTRODUCTION



Let me tell you a little about who I think you are: You're a web professional who's been hand-coding HTML and CSS for a few years; you're pretty comfortable with creating complex layouts, and you know not only your `div` from your `span` but also your `bold` from your `strong`; you've read a little about CSS3 and may even have started experimenting with some of its more decorative features like rounded corners, but you want to gain a deeper understanding of the fundamentals.

The *Book of CSS3* helps you leverage the excellent knowledge you have of CSS2.1 in order to make learning CSS3 easier. I won't explain the fundamentals of CSS (except for the occasional reminder) as I assume you know them already. I won't talk you through step-by-step demonstrations of using CSS to make a listed navigation or an image gallery because I assume you can apply the examples in this book to anything you want to build on your own.

What I aim to do with this book is introduce you to what you can do with CSS3 now and what you'll be able to do with it in the future. I want to take the dense technical language of the CSS3 specification and translate it into language that's plain and practical.

In short, I want to give you some new tools for your toolkit and let you make cool stuff with them.

The Scope of This Book

CSS can be used across many types of media; almost any device that's capable of displaying HTML or XML can also display CSS rules, albeit in a limited form sometimes. CSS3 has two modules devoted exclusively to paged media, such as PDF or printed materials, and also supports braille, handheld mobile devices (i.e., cellphones rather than smartphones), teletypes, and televisions. The range and breadth of possibilities is so vast that I can't cover them all.

What this book focuses on is CSS for the computer screen. All of the demonstrations were written for (and tested in) the most common desktop browsers, and they're optimized for users of desktop and laptop computers. Although many of the new features in this book will still work if you're developing for other devices—especially smartphones and tablets—I make no guarantees or assurances that everything will display exactly as shown in the examples contained herein.

A Quick Note About Browsers and Platforms

I wrote the majority of this book—and, therefore, the majority of the demonstrations and examples—on a computer running Ubuntu 10.04 with Firefox, Chrome, and Opera installed. Other portions were written on a MacBook Pro with Safari installed. Tests for Internet Explorer were performed using Windows 7. (The exact versions of all of the browsers used can be found in the introduction to Appendix A.)

Throughout this book, I mostly make reference to Firefox and WebKit. The perspicacious among you will notice that Firefox is a type of browser, whereas WebKit is a type of layout engine, and wonder why I don't refer to the Gecko layout engine used by Firefox or to any WebKit-based browser by name.

The reason is quite simple: Firefox is clearly the preeminent Gecko-based browser, whereas Chrome and Safari dispute the eminence of WebKit between them. As a simple space-saving exercise, I will say "WebKit" rather than "Chrome and Safari." The exception to this rule is when a specific feature or syntax only appears in one type of WebKit browser—such as hardware-accelerated 3D transformations in Safari—in which case, I refer to the name of the browser in question.

The Appendices and Further Resources

At the end of this book are two appendices containing further information beyond what's discussed in the various chapters. The first provides a quick reference guide to the implementation of the features included in this book across the different versions of browsers, and the second is a list of online resources, useful tools, and interesting demonstrations.

A website accompanies this book at <http://www.thebookofcss3.com/>; here I'll keep updated versions of both appendices and all of the examples and demonstrations used in this book. God forbid I should make any mistakes, but on the super-rare possibility that I do, I'll also keep a full list of errata.

In addition to the accompanying website, I write more about CSS3 (and other emerging web technologies) at my blog, Broken Links (<http://www.broken-links.com/>). Feel free to comment or get in touch with me through either of these websites.

1

INTRODUCING CSS3



In this first chapter, I'll introduce a new CSS3 property in order to demonstrate the code conventions used in this book, but before getting to that, I'll explain a little about the history of CSS3. Obviously, you don't need to know its history in order to use CSS3, but I think having some context about the current state of CSS3 is important.

CSS3 is a specification in flux. Some parts of the spec are considered stable and have been well implemented in modern browsers; other parts should be considered experimental and have been partially implemented to varying degrees; yet others are still theoretical proposals and have not been implemented at all. Some browsers have created their own CSS properties that don't belong in any CSS3 specification and perhaps never will.

All of this means that knowing how the standardization process works and the levels of implementation for each new property is vital to understanding how you can use CSS3 in your code both now and in the future.

What CSS3 Is and How It Came to Be

First, I want to discuss what CSS3 is—and isn't—and the form it takes. The W3C's approach to CSS3 is quite different from its approach to CSS2, so this overview should help you understand how and when you can use CSS3 and why it has such varied implementation across different browsers.

A Brief History of CSS3

The version of CSS in current use is CSS2.1, a revision of the CSS2 specification that was originally published in 1997. Despite ongoing development and review since that time, many people are surprised to learn that CSS2 hasn't actually become an "official" recommendation of the W3C yet. (I'll talk more about the recommendation process shortly.) More surprising still is the fact that Internet Explorer 8 (IE8)—released in 2009—lays claim to being the first browser to support the entire CSS2.1 specification fully.

In the last few years, the talk has been about the new revision—CSS3. I say "new," but in fact work on CSS3 began back in 1998, the year after CSS2 was published. Browser implementation of CSS2 continued to be so frustratingly inconsistent, however, that the W3C decided to halt work on any new revision and work on CSS2.1 instead, standardizing the way CSS had been implemented in the real world. In 2005, all of the CSS3 modules were moved back to draft status, and the editing and review process began all over again.

For many years, Internet Explorer dominated the ever-expanding market of Internet users and showed no sign of wanting to implement CSS3. But in the last few years, a whole new range of browsers has appeared to compete for users, and this plethora of choice has led to a features arms race. One beneficiary of that arms race has been CSS3. Each of the browsers wants to offer developers and users the latest in web technologies, and with the CSS3 spec already mostly written, implementing and even adding new features has been a no-brainer.

So here we are today, with the CSS3 specification under active development, a broad range of browsers working on implementing it, and a community of interested developers building with it, studying it, and writing about it. A healthy situation, and one we couldn't have foreseen just a few years ago.

CSS3 Is Modular

Being the default styling language for every markup-based document in the world is an enormous undertaking, and the W3C was aware that it would take many years to come to fruition. W3C members, conscious that they didn't want to hold up some of the more obvious, in-demand features while they were considering and debating some of the more esoteric ones, made the decision to split CSS3 into various modules. Each of the modules could then be worked on by different authors at different paces, and the implementation and recommendation process—which I'll discuss shortly—could be staggered.

This is why, instead of a single, monolithic CSS3 specification document, you have CSS3 Basic User Interface Module, Selectors Level 3, Media Queries,

and so on. Some of these modules are revisions of CSS2.1, and some are newly created, but all fall under the banner of CSS3.

One of the few things I find irritating (I'm an easy-going guy) is that on many blogs you'll hear people complaining, "I want to use CSS3, but it won't be ready for years." This is nonsense; some modules of CSS3 already have very stable implementation in all modern browsers, and many more are just months away from prime time. If you want to wait until all of the modules are 100 percent implemented across every browser in existence, you'll be waiting a long time.

But CSS3 is here, and some of it is ready to use right now—you just have to be mindful about how you use it.

Module Status and the Recommendation Process

As I move through this book and discuss each of the different modules, I'll also refer to that module's status. Status is set by the W3C, and it indicates the module's progress through the recommendation process; note, however, that status is *not* necessarily an indication of a module's degree of implementation in any browser.

When a proposed document is first accepted as part of CSS3, its status is designated *Working Draft*. This status means that the document has been published and is now ready for review by the community—in this case, the community being browser makers, working groups, and other interested parties. A document may stay as a Working Draft for a long period, undergoing many revisions. Not all documents make it past this status level, and a document may return to this status on many occasions.

Before a document can progress from a Working Draft, its status changes to *Last Call*. This means the review period is about to close and usually indicates the document is ready to progress to the next level.

That next level is *Candidate Recommendation*, which means the W3C is satisfied that the document makes sense, that the latest reviews have found no significant problems, and that all technical requirements have been satisfied. At this point, browser makers may begin to implement the properties in the document to gather real-world feedback.

When two or more browsers have implemented the properties in the same way and if no serious technical issues have come to light, the document may progress to being a *Proposed Recommendation*. This status means that the proposal is now mature and implemented and ready to be endorsed by the W3C Advisory Committee. When this endorsement has been granted, the proposal becomes a *Recommendation*.

To reiterate what I briefly touched on before, the recommendation process and the implementation process do not always work in the same way. For example, later on in this book, I'll introduce a set of modules proposed by the WebKit team a few years ago, which includes 2D Transformations (Chapter 12). Despite the proposal still having Working Draft status, the properties are already well implemented in Firefox, Opera, and WebKit.

As I mentioned earlier in this chapter, not even CSS2.1—which we've all been using for many years now—has reached full Recommendation status.

Although CSS2.1 is as good as finished, a few matters of syntax and phrasing still need to be resolved. Obviously, that hasn't stopped the browsers from implementing it fully and moving on to CSS3.

As a result, I've written this book in a loose order of implementation, rather than recommendation status. Earlier chapters discuss features that have full implementation across all browsers (or should by the time this book is released), later chapters cover features that are implemented in some browsers only—often with browser-specific prefixes—and chapters toward the end of the book deal with potential, speculative, or partial implementations of properties.

CSS3 Is Not HTML5

One of the current buzzwords around the Internet is HTML5. HTML5 is, of course, a genuine (and exciting) new technology that has somehow broken out of the technical press and through to the mainstream media. Just about everywhere you turn people are discussing it. On that journey, however, its correct meaning seems to have been lost.

Before I discuss what that real meaning is, I should point out that the media are not solely responsible for obfuscating the true meaning of HTML5. Many developers are falling over themselves to make flashy “HTML5 demos”; but if you look more closely at these demos (or view their source code), more often than not you'll find they involve few-to-no actual new HTML5 features and an awful lot of CSS3 (yes, Apple, I'm looking at you: see <http://www.apple.com/html5/>).

Although HTML5 will bring a lot of cool new features to the Web, CSS3 is bringing the really fancy visual stuff: rotating, scaling, and animating in two and three dimensions; dynamic and decorative text effects; drop shadows; rounded corners; and gradient fill effects. All this is possible with CSS3 (and I'll show you how to do it all in this book).

What the media refers to as HTML5 is really that new revision of the markup language combined with CSS3, SVG, and JavaScript—what many people (myself included) prefer to call part of the *Web Stack* (or, *Open Web Stack*).

Let's Get Started: Introducing the Syntax

With the introductions and explanations out of the way, let's get to the meat of CSS3. Throughout this book, I use a certain syntactical convention to demonstrate each of the new rules and properties. Rather than simply describe that convention, I thought you'd find it more interesting if I explain it at the same time as I introduce the first new CSS3 property.

The property is *box-sizing*, which allows you to set how an element's dimensions are calculated. As you know, an element's total width—without including its margin—is usually calculated from its stated (or inherited) width value, plus its left and right padding, plus its border-width. Let's take as an example the following code, which should be very familiar to you.

```
div {  
  border: 10px solid black;  
  padding: 10px;  
  width: 150px;  
}
```

In this example, the `div` has a total width of 190px—150px width plus 20px padding plus 20px border. This code is fine and uncontentious when you’re using absolute values for `width`, but it becomes problematic when you start mixing in percentages:

```
div {  
  border: 10px solid black;  
  padding: 10px;  
  width: 15%;  
}
```

Now the width has become a lot harder to calculate because you first need to know the width value of the parent element before you can know how much 15 percent of that is and add the pixel values. Calculations can become very complicated very quickly, making percentage-based layouts tricky to manage properly.

At this point, the `box-sizing` property comes in handy. With this, you can set from which part of the box model—the content itself or the border—the width of the element is calculated. So now that I’ve covered the reason for the new property’s existence, I’ll begin the demonstration with a look at its syntax, using the convention that will be standard in this book:

```
E { box-sizing: keyword; }
```

In this code example, the selector is represented with *E*. Of course, in HTML, this selector doesn’t exist; I’m merely using it as an indicator that any selector can be used here. In the case of the examples used at the beginning of this section, the *E* represents the `div` element.

Next, you have the property itself: `box-sizing`. This property is implemented in all major browsers but with browser-specific prefixes in some: In Firefox, the property uses the `-moz-` prefix—that is, the property is `-moz-box-sizing`; and in WebKit, it has the `-webkit-` prefix, so it would be `-webkit-box-sizing`. Rather than obfuscate the code with all of the different prefixes, I use only the correct property name according to the CSS3 specification and note in the text when a prefix is required, as I’ve just done here. (I’ll explain more about these browser prefixes in the following section.)

The value of the declaration is the word *keyword*, but again the value is only an indicator; it represents a range of possible values. For `box-sizing` the permitted values are `border-box` and `content-box`.

With all of that in mind, if you used actual values, one possible real-world application of this new property might look like this:

```
div { box-sizing: content-box; }
```

So what does this property do? As mentioned, `box-sizing` sets the point of the element from which the width is calculated. The default is `content-box`, which means the stated width of the element applies to the content, and the padding and border values are added on as usual.

A value of `border-box` means the stated width value is the total width of the content, the padding, and the border—in other words, the entire box (without the margin). To illustrate this, let's return to the previous example code and add the property:

```
div {  
  border: 10px solid black;  
  box-sizing: border-box;  
  padding: 10px;  
  width: 150px;  
}
```

Now the width of 150px *includes* the padding and border. As each of those is 10px on both sides, the content of the `div` element has a calculated width of 110px—150px minus the 20px padding and 20px border.

Before illustrating the difference between the two values visually, I should also mention that Firefox (with its `-moz-box-sizing` property) has an extra permitted value that other browsers don't: `padding-box`. Using this value, the width of the element is calculated from its padding and its content and doesn't include the element's border. Here's an example:

```
div {  
  border: 10px solid black;  
  -moz-box-sizing: padding-box;  
  padding: 10px;  
  width: 150px;  
}
```

In this code, the 150px width includes the padding, so the width of the content itself would be 130px. You can see the difference among the values when rendered in Firefox in Figure 1-1.

Moving through the examples from left to right: The first `div` has the default `content-box` value for `box-sizing`, so the stated width—150px—is the width of the content, and the border and padding are added to the content. In the second example, the value is `border-box`, meaning the 150px width includes the border and padding, making the content 110px wide. Finally, the `div` on the right uses Firefox's own `-moz-box-sizing` property with the value of `padding-box`, so the 150px includes the content and padding but not the border, putting the content's width at 130px.

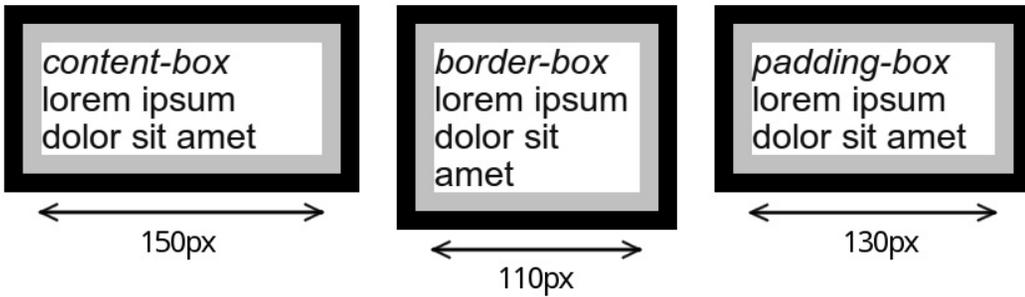


Figure 1-1: The effects of different values on the box-sizing property

Please note that although I only discuss the width property in all of these examples, the exact same rules apply to an element’s height property. (You may also note that this works in exactly the same way as a browser that is put into “quirks” mode.)

NOTE *If you’re a younger developer you may not remember “quirks” mode. It’s a system that emulates the incorrect way that Internet Explorer 5.5 used to lay out web pages; you can read more about it on Wikipedia (http://en.wikipedia.org/wiki/Quirks_mode).*

I use the box-sizing property in a few examples throughout this book, so if the effects (and benefits) aren’t immediately apparent right now, they should become clearer as you work through the rest of the chapters.

Browser-Specific Prefixes

In the previous section, I briefly discussed using browser-specific prefixes on the box-sizing property. As CSS3 is still in a state of change and revision, you’ll see these mentioned a lot throughout the rest of this book, so I’ll take some time to talk about these in more detail.

When a module is still under active review, as much of CSS3 is, a lot is subject to change; the syntax of a property may be revised, or properties may be dropped entirely. On occasion, even the wording of the draft itself is perhaps a little nebulous and open to interpretation.

At the same time, browsers need to implement these features so we can see how they work in practice. But consider the difficulties that would occur if two separate browsers implemented the same property but interpreted it slightly differently: Your code would appear differently—perhaps radically so—in each of the browsers. To prevent this from happening, each of the different browser engines prefixes a short code to the beginning of experimental properties. Let’s imagine a property called `monkeys` (I’ve always wanted a `monkeys` property), which is brand new in the specification, and that all of the browsers have decided to implement it to see how it works. In this case, you would use the following code:

```
E {
  -moz-monkeys: value; /* Firefox */
  -ms-monkeys: value; /* Internet Explorer */
```

```
-o-monkeys: value; /* Opera */  
-webkit-monkeys: value; /* WebKit */  
}
```

The amount of repetition may seem somewhat unnecessary, but the repetition is for our own good; the last thing you want is for all the browsers to implement the `monkeys` property differently, leading to total chaos. (You'll see a great example of the benefits of prefixes when I discuss gradients in Chapter 11.)

Future-Proofing Experimental CSS

Quite often people will suggest that when using prefixed, experimental CSS properties, you also add the unprefixed property at the end:

```
E {  
  -moz-monkeys: value; /* Firefox */  
  -ms-monkeys: value; /* Internet Explorer */  
  -o-monkeys: value; /* Opera */  
  -webkit-monkeys: value; /* WebKit */  
  monkeys: value;  
}
```

The theory is that this future-proofs the code; when the property is fully implemented in the browsers, you don't need to go back and add the property to your stylesheets. I used to agree with this technique, but now I'm not so sure. I think future-proofing is okay if more than one browser has already fully implemented the property in a compatible way, as that usually means the specification is stable. However, if the spec is still under review, then the syntax is subject to change. Adding the unprefixed property could cause problems when browsers actually implement the updated syntax—or, indeed, it may not work at all.

Some properties in this book—such as `background-size` in Chapter 8—are semi-implemented, which is to say they are prefixed in some browsers, but unprefixed in others. In this case, you will obviously have to use a mixture of the two states if you want to use those properties across browsers. Others—like the aforementioned gradient properties in Chapter 11—are immature, still open to review, and far from final, so you should probably not use the future-proofing method with them just yet.

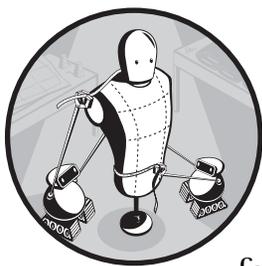
Getting Started

That should be everything you need to get started with this book—except, of course, an inquisitive nature. I have a lot of ground to cover in CSS3, so I'll move fairly quickly, but each chapter should give you the knowledge you need to build your own tests, demonstrations, and sites that take advantage of the flexibility and rich features that CSS3 provides.

We'll begin with a look at one of the simplest—and yet potentially the most disruptive (and I mean that in a good way)—new features: Media Queries.

2

MEDIA QUERIES



Back when the World Wide Web was something you only accessed via a browser on your desktop or laptop, writing CSS was fairly straightforward. Although you had to consider cross-browser and cross-platform issues, at least you knew with reasonable certainty that everyone was using fundamentally similar devices to view your website. Over the last few years, however, we've seen an explosion of new devices for accessing the Web—from game consoles to mobile devices such as the iPhone or iPad. Presenting your content to everybody in the same way no longer makes sense when they could be viewing your website on a widescreen desktop monitor or a narrow handheld screen.

CSS has had a way to serve different styles to different media types for quite some time, using the `media` attribute of the `link` element:

```
<link href="style.css" rel="stylesheet" media="screen">
```

But using this is like wielding a pretty blunt instrument when the screen in question can be between 3.5 inches and 32 inches in size. The

CSS3 solution to this problem is the Media Queries Module (<http://www.w3.org/TR/css3-mediaqueries/>). Media Queries extend the media types by providing a query syntax that lets you serve styles far more specifically to your user's device, allowing a tailored experience. The description may sound quite dry, but this feature is actually one of the most revolutionary in the entire CSS3 specification. Media Queries give you the freedom to make websites that are truly device-independent and give your users the best possible experience no matter how they choose to visit your site.

The Media Queries Module has Candidate Recommendation status so is considered ready for implementation. The module is already well implemented in Firefox, WebKit, and Opera, and will be in Internet Explorer from version 9.

The Advantages of Media Queries

As a quick demonstration of the power and flexibility of Media Queries, I want to show an example of how websites can be optimized for mobile browsers without requiring a great deal of extra development.

People visiting your site on a mobile device may well struggle to use it: The text may appear too small, and zooming in means a lot of scrolling to find navigational elements; those navigational elements may involve drop-down functionality that is triggered by hovering over them, an action that often doesn't exist on mobile devices; large images may take a long time to download over a weak data connection and use a substantial portion of your monthly bandwidth allowance. Some sites plan for this by providing mobile-friendly versions, but these generally involve a lot of development work. A subdomain has to be set up with stylesheets and HTML templates that differ from the parent site, images have to be resized to better fit small screens, and a script has to be created to detect whether a mobile browser is being used and to redirect to the mobile site accordingly. This approach can cause problems: Your script has to be kept up to date with all mobile browser versions, and maintenance often involves duplication to keep both mobile and desktop versions in sync.

Media Queries address many of these issues. For a start, they detect devices based on their attributes, so no browser-sniffing scripts are required. They allow you to target stylesheets directly for a device's capabilities, so if a device with a small screen is detected, CSS rules will be tailored to that screen size, removing extraneous elements from the screen, serving smaller images, and making text clearer.

For example, take a look at the website of the dConstruct conference from 2010 (<http://2010.dconstruct.org/>), as shown in Figure 2-1.

When viewed in a desktop browser, the site features large images of the speakers, and text is displayed in columns laid out horizontally. Through the power of Media Queries, when you see the same site viewed in a narrower browser—as smartphones such as the iPhone would use—the speaker images are removed, the links to the speakers' pages are more prominent, and all of the text on the page is moved into a single column, which is ideal for scrolling down.



Figure 2-1: The dConstruct website viewed in a desktop browser (left) and a mobile browser (right)

Of course, the Web is appearing on more than just desktop and smart-phone devices, and we really need to be working toward an era of websites optimized for any device. I urge you to read Ethan Marcotte’s article, “Responsive Web Design” (<http://www.alistapart.com/articles/responsive-web-design/>), which provides a great introduction to this new paradigm of web design.

And if you want to see what other people have been doing with Media Queries there’s a great gallery online at <http://www.mediaqueri.es/>, which showcases some of the better examples of what’s possible.

Syntax

A Media Query sets a parameter (or series of parameters) that displays associated style rules if the device used to view the page has properties that match that parameter. You can use Media Queries in three ways, all of which match the different ways that CSS can be applied to a document. The first is to call an external stylesheet using the link element:

```
<link href="file" rel="stylesheet" media="logic media and (expression)">
```

The second is to call an external stylesheet using the @import directive:

```
@import url('file') logic media and (expression);
```

The third is to use Media Queries in an embedded style element or in the stylesheet itself with the extended `@media` rule:

```
@media logic media and (expression) { rules }
```

This method is the one I'll use throughout the rest of this chapter, as it's clearer for demonstration purposes. Which method you use will largely depend on your own preference and the demands of your existing stylesheet structure.

Now that I've introduced the declaration methods, let's explore the syntax. You should already be familiar with the `media` attribute—it declares the media types that the styles are applied to, just as in the HTML link tag:

```
<link href="style.css" rel="stylesheet" media="screen, projection">
```

As with the current syntax, you can use a comma-separated list to choose multiple media types.

The first new attribute for the `@media` rule is *logic*. This optional keyword can have the value of either *only* or *not*:

```
@media only media and (expression) { rules }  
@media not media and (expression) { rules }
```

The *only* value is used if you want to hide the rule from older browsers that don't support the syntax; for browsers that do support it, *only* is effectively ignored. The *not* value is used to negate the Media Query; you use *not* to apply the styles if the parameters you set are *not* met.

The next attribute is *expression*, which is where the main selection takes place. You declare *expression* by using the *and* operator and use it to set parameters beyond the media type. These parameters are known as *Media Features*, and they're critical to the power of Media Queries. That being the case, let's explore them in detail.

Media Features

Media Features are information about the device that's being used to display the web page: its dimensions, resolution, and so on. This information is used to evaluate an *expression*, the result of which determines which style rules are applied. That *expression* could be, for example, "apply these styles only on devices that have a screen wider than 480 pixels" or "only on devices that are orientated horizontally."

In Media Queries, most Media Feature expressions require that a value be supplied:

```
@media media and (feature:value) { rules }
```

This value is required to construct the example expressions I just mentioned. In a few cases, however, the value can be left out and just the existence of the Media Feature itself tested against:

```
@media media and (feature) { rules }
```

Expressions will become clearer as I talk through the Media Features and explain when values are required or optional.

With the syntax covered, let's meet some of the more prominent Media Features. The ones I introduce next are the most applicable to color display screens used for accessing the Web and are the ones you're most likely to use on a day-to-day basis. Other Media Features are available, but they're more likely to be used for alternative devices such as TVs or fixed-grid terminals.

Width and Height

The width Media Feature describes the width of the rendering viewport of the specified media type, which, in practice, usually means the current width of the browser (including the scroll bar) for desktop operating systems. The basic syntax requires a length value:

```
@media media and (width:600px) { rules }
```

In this case, the rules are applied only to browsers that are set to be exactly 600px wide, which is probably far too specific. width also accepts one of two prefixes, however, max- and min-, which allows you to test for a minimum or maximum width:

```
@media media and (max-width:480px) { rules }  
@media media and (min-width:640px) { rules }
```

The first query applies the rules in browsers that are no wider than 480px, and the second in browsers that are at least 640px wide.

Let's look at a practical example. Here, I'll take advantage of browser window sizes by providing a decorative header for wider windows:

```
@media screen and (min-width: 400px) {  
  h1 {  
    background: black url('landscape.jpg') no-repeat 50% 50%;  
    color: white;  
    height: 189px;  
    margin-bottom: 0;  
    padding: 20px;  
  }  
}
```

This Media Query is testing for browser windows that are at least 400px wide and applying a background image to the h1 element when that is the case. If my browser window is at least 400px wide, I see the image; if I resize it to be narrower, only a text header is shown. You can see this example illustrated in Figure 2-2.

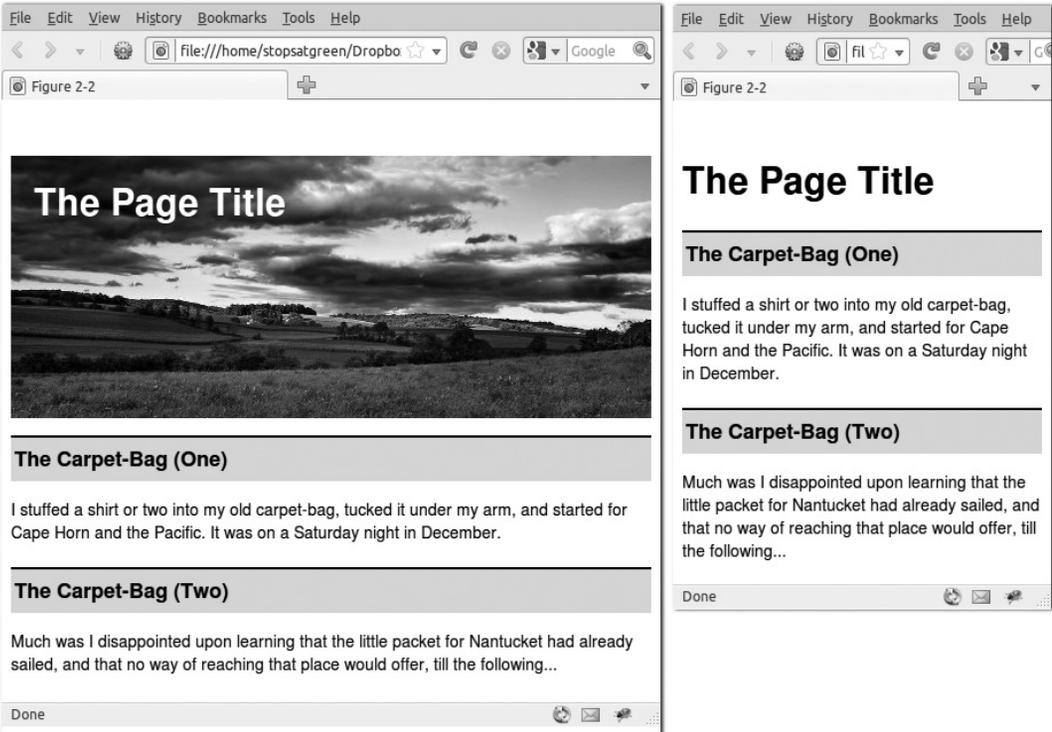


Figure 2-2: Different style rules applied with the width Media Query

The height Media Feature works in the same way, except that it targets browsers based on their height instead of width. The syntax is the same as width and also permits using max- and min- prefixes:

```
@media media and (height:value) { rules }  
@media media and (max-height:value) { rules }  
@media media and (min-height:value) { rules }
```

Because of the prevalence of vertical scrolling, however, height is used much less frequently than width.

Device Width and Height

The device-width Media Feature functions in a similar way, but it describes the width of the *device* that is rendering the page. When dealing with web pages, the device width is the width of the screen that's displaying the page rather than the width of the browser window. As with width and height, the basic syntax requires a length value and can be prefixed in the same way:

```
@media media and (device-width:1024px) { rules }
@media media and (max-device-width:320px) { rules }
@media media and (min-device-width:800px) { rules }
```

device-width becomes most useful when designing for mobile devices, which have smaller display areas. Using this feature, you can cater your designs to those smaller devices without having to create a whole new version of the site designed for the mobile web.

For example, I'll optimize some content for display on two different devices. I'll use two boxes of content that, by default, will be floated to sit next to each other on the horizontal plane. For smaller devices (I'll use an iPhone in this example) the boxes will not be floated and, instead, will sit one on top of the other, making better use of the narrower device. Here's the code:

```
.container { width: 500px; }
.container div {
  float: left;
  margin: 0 15px 0 0;
  width: 235px;
}
@media only screen and (max-device-width: 320px) {
  .container { width: auto; }
  .container div {
    float: none;
    margin: 0;
    width: auto;
  }
}
```

By default, I have a 500px container element, with two floated div elements inside it that are 235px wide and have horizontal margins of 15px. Then I create a Media Query (using the only operator to hide it from older browsers) that applies only to devices with a maximum width of 320px (the default iPhone width). This query sets rules that remove the explicit values from the width and margin properties and prevents the boxes from floating. You can see the results in Figure 2-3. The content is more appropriately formatted for its target device, without losing any readability or requiring a special “mobile version.”

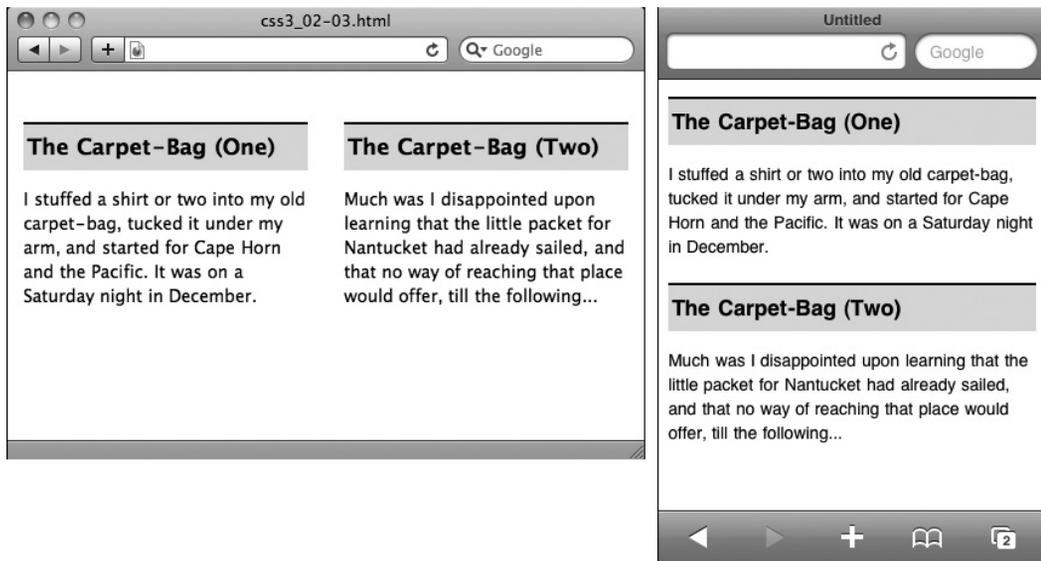


Figure 2-3: The device-width Media Query displays content differently on a desktop (left) and iPhone (right).

NOTE *iOS (iPhone, iPad, etc.) and Android devices measure device-width by the shorter pair of the four screen sides; that is, given a device with dimensions of 320×480, the device-width will be 320px, regardless of whether you're viewing it in portrait or landscape mode.*

The device-width feature also has a counterpart device-height, which operates as height does to width. Once more, device-height uses a similar syntax, also with max- and min- prefixes:

```
@media media and (device-height:value) { rules }
@media media and (max-device-height:value) { rules }
@media media and (min-device-height:value) { rules }
```

Much like height, device-height is used less frequently than device-width because vertical scrolling is easier than horizontal scrolling.

Using Media Queries in the Real World

In the examples thus far, I've tended to create a site that is optimized for larger browsers or devices first, with Media Queries used to provide different styles for smaller (mobile) devices. This has been useful for demonstration purposes, but in your own sites you'll probably want to do this the other way round.

The reason for this is because of the way that some browsers load page assets, such as images, that are included in stylesheets. Some early adopters of Media Queries would write their code the way we've seen in my examples, using large background images and then setting a value of `display: none` to hide them from mobile devices. However, those background images can still

be downloaded and held in the cache even though they aren't displayed. This increases the page's load time and can consume bandwidth allowances—neither of which is good for mobile device users without wireless connections.

A better way to create your pages is to make a basic stylesheet for your mobile audience first and then one with larger assets for desktop or tablet users that is loaded using a Media Query such as `device-width`:

```
<link href="basic.css" rel="stylesheet" media="screen">
<link href="desktop.css" rel="stylesheet" media="screen and (min-device-width: 480px)">
```

When the stylesheets are separated in this way, the file *desktop.css* won't be loaded for devices with a screen width of less than 480px, so none of those large assets will be downloaded in the background.

This will work for the great majority of browsers from the past few years; any really old browsers will get the basic stylesheet instead, which is probably better for them as they won't be able to cope with the advanced features I'll be teaching throughout the rest of this book.

The big exception to this is Internet Explorer (get used to that sentence; you'll be reading it a lot). While IE9 does have Media Query support, previous versions don't. To get around that you just need to load the desktop file using a conditional comment that only Internet Explorer recognizes:

```
<link href="basic.css" rel="stylesheet" media="screen">
<link href="desktop.css" rel="stylesheet" media="screen and (min-device-width: 480px)">
<!--[if lt IE 9]>
  <link href="desktop.css" rel="stylesheet" media="screen">
<![endif]-->
```

This simply means: "If you're using Internet Explorer below version 9, load the file *desktop.css*." It's a small bit of repetition, but it will solve this problem and let you build your websites in a progressive way.

Orientation

If you're less concerned with the actual dimensions of the viewing device but want to optimize your pages for either horizontal (like a typical web browser) or vertical (like an ebook reader) viewing, the Media Feature you need is orientation. Here is its syntax:

```
@media media and (orientation:value) { rules }
```

value can be one of two options: *landscape* or *portrait*. The *landscape* value applies when the width of your browser is greater than its height, and the *portrait* value applies when the opposite is true. Although orientation can certainly be applied to desktop browsers, you'll find it most useful when dealing with handheld devices that the user can easily rotate, such as the new generation of smartphones and tablets.

For example, you can use orientation to display a navigation menu horizontally or vertically, depending on the visitor's browser orientation. The code looks like this:

```
ul { overflow: hidden; }
li { float: left; }
@media only screen and (orientation: portrait) {
  li { float: none; }
}
```

By default, the `li` elements have a `float` value of `left`, making them stack horizontally across the page. If the same page is viewed in a portrait orientation—either by resizing the browser to be taller than it is wide or by viewing the page in a device with a portrait orientation—the `float` is removed and the `li` elements stack vertically instead. You can see the result in Figure 2-4.

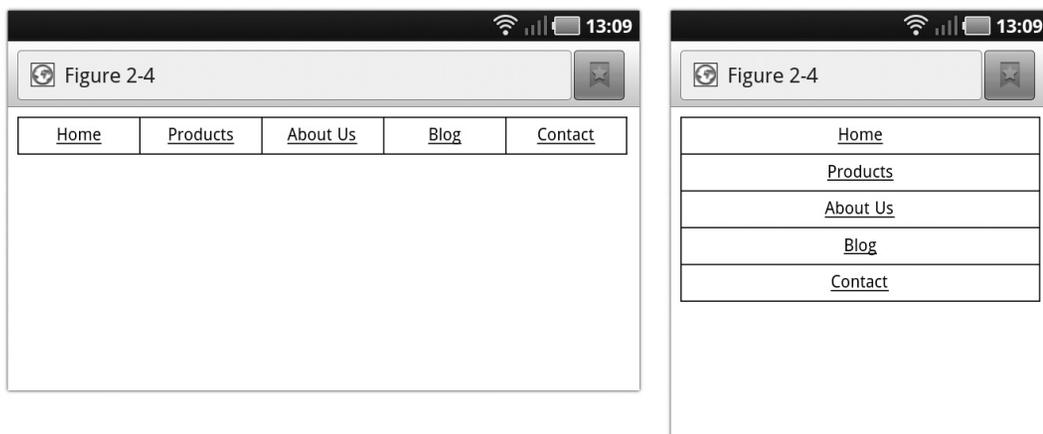


Figure 2-4: The orientation Media Query on an Android browser: landscape (left) and portrait (right)

As only two values are possible for the orientation feature, if you apply differentiating rules using one value, then the other tacitly becomes the opposite. In this example, I only used the portrait value, so, by default, all of the rules outside of that function apply to the landscape orientation.

Aspect Ratio

You can also create queries that apply when a certain width-to-height ratio is met. Use `aspect-ratio` to test the browser's aspect ratio or `device-aspect-ratio` to test the device's aspect ratio. Here is the syntax for these two features:

```
@media media and (aspect-ratio:horizontal/vertical) { rules }
@media media and (device-aspect-ratio:horizontal/vertical) { rules }
```

The *horizontal* and *vertical* values are positive integers that represent the ratio of the width and height (respectively) of the viewing device's screen, so a square display would be 1/1 and a cinematic widescreen display would be 16/9.

Selecting by aspect ratio is potentially fraught with caveats. For example, some device manufacturers define widescreen as 16:9, some as 16:10, and some as 15:10. This variation means you have to include all of these as parameters if you want to apply a "widescreen" set of rules.

Pixel Ratio

In general, the CSS Pixel unit (px) is a measurement of a single pixel on the computer screen—if your screen is 1024×768 resolution and you give an element a width of 1024px, you expect it to fill the screen horizontally. This is not always the case with smartphone and mobile devices, however. Reading websites on their small screens often involves zooming in, and this magnification causes a screen pixel to be larger than a CSS pixel. For example, magnifying a page by 100 percent means 1 CSS pixel is displayed on the screen by 4 device pixels (2×2).

Magnification is fine for scalable content such as text and vector graphics, but bitmap images can suffer badly from a loss of quality when magnified. To get around this problem, many devices now have screens with higher pixel density ratios, which allow for the display of high-resolution content without any loss of quality. The iPhone 4, for example, has a pixel density of 2, meaning every CSS pixel is displayed on screen by 4 device pixels (as in the example in the previous paragraph), allowing for 100 percent zoom without any loss of detail.

A Media Feature is available that lets you target devices based on their pixel density. The feature is `device-pixel-ratio`, and it's implemented in Mobile WebKit with the `-webkit-` prefix:

```
@media media and (-webkit-device-pixel-ratio: number) { rules }
```

The *number* is a decimal that represents the device's pixel density. For example, the Samsung Galaxy S has a pixel density of 1.5; to target screens similar to that, you would use:

```
@media screen and (-webkit-device-pixel-ratio: 1.5) { rules }
```

As with the other Media Features, you can also detect maximum and minimum pixel ratios:

```
@media media and (-webkit-max-device-pixel-ratio: number) { rules }  
@media media and (-webkit-min-device-pixel-ratio: number) { rules }
```

This flexibility makes serving higher-resolution images to browsers with higher pixel density easier, as you can see in this code:

```
❶ E { background-image: url('image-lores.png'); }
❷ @media screen and (-webkit-min-device-pixel-ratio: 1.5) {
    background-image: url('image-hires.png');
❸    background-size: 100% 100%;
}
```

The first rule (❶) means browsers on devices with a “standard” (or low-resolution) pixel ratio will use the standard image (*image-lores.png*), whereas devices with a pixel ratio of at least 1.5 will use the high-resolution image (*image-hires.png*) instead (❷). Note the use of the `background-size` property here (❸); this property should be used with high-resolution images to ensure they aren’t displayed larger than the element they are applied to (I introduce `background-size` fully in Chapter 8).

Pixel ratio detection should also be available in the Firefox Mobile browser (which is still in beta at the time of this writing), albeit with a different syntax for the maximum and minimum media features:

```
@media media and (-moz-device-pixel-ratio: number) { rules }
@media media and (max--moz-device-pixel-ratio: number) { rules }
@media media and (min--moz-device-pixel-ratio: number) { rules }
```

Multiple Media Features

You can chain multiple queries together on the same media type by adding expressions with the `and` operator:

```
@media logic media and (expression) and (expression) { rules }
```

This syntax tests that both expressions are matched before applying the selected rules. For example, to make sure all permutations of widescreen are covered, as mentioned in the previous section, you would create this query:

```
@media only screen and (aspect-ratio: 15/10) and (aspect-ratio: 16/9) and
(aspect-ratio: 16/10) { rules }
```

You can also set different expressions on multiple media types:

```
@media logic media and (expression), media and (expression) { rules }
```

Here a different expression would be used for each media type, for example, setting some rules to all landscape devices and portrait projection devices:

```
@media all and (orientation: landscape), projection and (orientation: portrait) { rules }
```

You can also, of course, create any combination of the above syntaxes.

Mozilla-Specific Media Features

As part of the work on Firefox Mobile, the Mozilla team has introduced a number of proprietary new media features, many of which are very specific to Gecko (the Firefox rendering engine), but some of which may be proposed to the W3C as official features. You can see all of these at https://developer.mozilla.org/En/CSS/Media_queries#Mozilla-specific_media_features.

Perhaps the most interesting is `-moz-touch-enabled`, which allows you to apply rules to elements specifically on touchscreen devices, for example, for making buttons bigger to suit fingers rather than a stylus or mouse. Here's the syntax:

```
@media media and (-moz-touch-enabled) { rules }
```

A device is either touch enabled, in which case it has a value of 1, or isn't, in which case the value is 0. As such, you don't need to state a value parameter and can use the *feature* keyword only.

Summary

Their syntax may be simple, but Media Queries have the capacity to be extraordinarily powerful. With the mobile web explosion of recent years, designers and developers are beginning to realize they have the power to tailor their content to the user without employing the old techniques of browser sniffing or separate (and completely different) mobile versions of their sites.

Media Queries are already being recognized and hailed by some of the leading lights of web development, and talks are being given about their use at all of the big web conferences. This enthusiasm is largely due to the desire to overcome the constraints of browsing the Web on handheld devices with small screens but is now also being driven by the excitement of early iPad adopters.

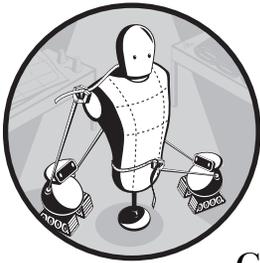
With careful consideration and clever use of Media Queries, you can create websites that scale perfectly for users, however they access the Web.

Media Queries: Browser Support

	WebKit	Firefox	Opera	IE
Media Queries	Yes	Yes	Yes	No (expected in IE9)

3

SELECTORS



Selectors are the heart of CSS, and although the original CSS1 specification had only 5 or 6, CSS2 expanded the range with 12 more. CSS3 goes further still, roughly doubling the number of available selectors.

Selectors can be broadly separated into two categories. The first are those that act directly on elements defined in the document tree (p elements and href attributes, for example); this category contains *class*, *type*, and *attribute* selectors. For the sake of expediency, I'll group these together under the banner of *DOM* selectors. The second category contains *pseudo*-selectors that act on elements or information that sits outside of the document tree (such as the first letter of a paragraph or the last child of a parent element). I'll cover pseudo-selectors in the next chapter—here I'll discuss DOM selectors.

CSS3 provides three new attribute selectors and one new *combinator*—that is, a selector that joins other selectors together, such as the child combinator (>) from CSS2. These are defined in the Selectors Level 3 Module (<http://www.w3.org/TR/css3-selectors/>), which currently has the status of *Proposed Recommendation*. This status means the module already has widespread and stable implementation across most browsers (with the current exception of

Internet Explorer, though full support is planned for IE9). The Selectors Level 3 Module is also expected to be one of the first to obtain *Candidate Recommendation* status. As long as you make provisions for Internet Explorer's lack of support, you can start using CSS3 selectors right away—many sites already do.

Attribute Selectors

Attribute selectors were introduced in CSS2, and, as you may expect from the name, they allow you to specify rules that match elements based on their attributes—such as href or title—and the values of those attributes. The four selectors defined in CSS2 are:

```
E[attr] {} /* Simple Attribute Selector */
E[attr='value'] {} /* Exact Attribute Value Selector */
E[attr~='value'] {} /* Partial Attribute Value Selector */
E[attr|='value'] {} /* Language Attribute Selector */
```

Before moving on to the new selectors in CSS3, a quick recap of how each selector is utilized is worthwhile. For this, I'll use the following markup, which is a (very short) contact list:

```
<ul>
❶ <li><a href="" lang="en-GB" rel="friend met">Peter</a></li>
❷ <li><a href="" lang="es-ES" rel="friend">Pedro</a></li>
❸ <li><a href="" lang="es-MX" rel="contact">Pancho</a></li>
</ul>
```

The *Simple Attribute Selector* applies rules to elements that have the specified attribute defined, regardless of that attribute's value. So given the following code:

```
a[rel] { color: red; }
```

all of the a elements in my markup have a rel attribute, despite their having different values. In this case, therefore, all elements have the rule applied. If you want to be more specific, you can use the *Exact Attribute Value Selector* to define a value:

```
a[rel='friend'] { color: red; }
```

This code applies the rule only to the second a element in our markup (❷) because it selects only elements that have the exact value of friend. If you want to select both of the elements that contain the value of friend, you would use the *Partial Attribute Value Selector*:

```
a[rel~='friend'] { color: red; }
```

This code looks for the value of `friend` as part of a space-separated list (in most cases, a word) in any `rel` attribute and so applies the rule to elements ❶ and ❷.

The final selector, the *Language Attribute Selector*, applies rules to elements based on their `lang` attribute. The example markup has two Spanish names, although one is from Spain and the other is from Mexico. To select both of these, you use this code:

```
a[lang|= 'es'] { color: red; }
```

This code selects all `lang` attributes whose value begins with `es`, regardless of their country values—that is, elements ❷ and ❸.

New Attribute Selectors in CSS3

You've seen how useful attribute selectors can be for finding exact or partial values in selectors, but what if you want even more flexibility? CSS3's new selectors provide it with the power to match substrings within an attribute value. This feature makes them especially useful for applying rules to XML documents, which have more arbitrary attributes than HTML—though they are still quite useful for HTML developers as well.

Beginning Substring Attribute Value Selector

The first new attribute selector—which, to avoid having to repeat that mouthful of a title, I'll refer to as the *Beginning Selector*—finds elements whose chosen attribute begins with the string supplied to it as an argument. It uses the caret (^) symbol to modify the equals sign in the property. Here's the full syntax:

```
E[attr^='value'] {}
```

This code looks for the supplied value at the beginning of the specified attribute. For example, if you use the following rule:

```
a[title^='image'] {}
```

and apply it to this markup:

```
<p>Lorem <a href="http://example.com/" title="Image Library">ipsum</a></p>
<p>Lorem <a href="http://example.com/" title="Free Image Library">ipsum</a></p>
```

the rule will be applied to the `a` element in the first paragraph since the title attribute string begins with the word *image*. However, the rule will *not* be applied to the `a` element in the second paragraph because its title attribute string contains those characters but does not begin with them.

NOTE In HTML documents, the attribute selector value is case insensitive; for XML documents, however, the value is case sensitive.

The Beginning Selector is especially useful when you want to add visual information to hyperlinks. Here's an example of a typical hyperlink to an external website:

```
<p>This is a <a href="http://example.com/">hyperlink</a>.</p>
```

When you see this link in your browser, you can't immediately tell whether it's a link to a page on the same website or to an external URI. With this new attribute, however, you can pass the protocol (http) as the argument and add an icon to signify external links clearly:

```
a[href^='http'] {  
    background: url('link-go.png') no-repeat left center;  
    display: inline-block;  
    padding-left: 20px;  
}
```

The result is shown in Figure 3-1.

This is a hyperlink.

Figure 3-1: An icon applied with the Beginning Selector

Of course, you can extend this to cover many other web protocols, some of which are used in this example:

```
a[href^='mailto'] { background-image: url('email_go.png'); }  
a[href^='ftp'] { background-image: url('folder_go.png'); }  
a[href^='https'] { background-image: url('lock_go.png'); }
```

With the different protocols provided as values for the selector, you can apply it to this example markup:

```
<p>Lorem ipsum dolor <a href="mailto:email@example.com">email</a> sit amet.</p>  
<p>Nulla lacus metus <a href="ftp://example.com">FTP server</a> luctus eget.</p>  
<p>Etiam luctus tortor <a href="https://example.com">secure server</a> quis.</p>
```

The results of the output are shown in Figure 3-2.

- Lorem ipsum dolor  email sit amet.
- Nulla lacus metus  FTP server luctus eget.
- Etiam luctus tortor  secure server quis.

Figure 3-2: More examples of link icons with the Beginning Selector

Of course, the Beginning Selector also has many applications with attributes that accept more verbose values, such as `alt`, `cite`, and `title`. And with the introduction of HTML5 and a whole range of new form elements and attributes, this selector and its soon-to-be-introduced siblings will become much more flexible.

Consider, for example, the proposed `datetime` attribute. This attribute accepts date-string values such as `2010-03-11`, so you could use the Beginning Selector to apply styles to all elements meeting a supplied year value, which is very handy for calendar or archiving applications.

Ending Substring Attribute Value Selector

The *Ending Selector*, as I'll call it, works exactly like the Beginning Selector—just the other way around! That is, you use it to select attributes that *end* with the supplied value. The syntax differs by just one character: This time you use the dollar character (\$) to modify the equal sign (=). Here's the full syntax:

```
E[attr$='value'] {}
```

Let's return to the first example in this chapter, which used this markup:

```
<p>Lorem <a href="http://example.com/" title="Image Library">ipsum</a></p>
<p>Lorem <a href="http://example.com/" title="Free Image Library">ipsum</a></p>
```

And apply the new rule to it with a new value:

```
a[title$='library'] {}
```

This time the rule applies to both a elements because they both end with the word *library*.

As with the Beginning Selector, you can use this selector to provide visual clarity to hyperlinks. But this time, instead of using the protocols at the beginning of the `href` attribute, you use the file types at the end. The code here shows rules for many popular file-type extensions:

```
a[href$='.pdf'] { background-image: url('pdf.png'); }
a[href$='.doc'] { background-image: url('word.png'); }
a[href$='.rss'] { background-image: url('feed.png'); }
```

And here's a markup snippet containing a list of links to files:

```
<ul>
<li>Lorem ipsum dolor <a href="/example.pdf">PDF</a> sit amet.</li>
<li>Lorem ipsum dolor <a href="/example.doc">MS Word</a> sit amet.</li>
<li>Nulla lacus metus <a href="/example.rss">RSS Feed</a> luctus eget.</li>
</ul>
```

When the stylesheet is applied to the markup, an appropriate icon is applied to each of the `a` elements, as shown in Figure 3-3.

- Lorem ipsum dolor  [PDF](#) sit amet.
- Lorem ipsum dolor  [MS Word](#) sit amet.
- Nulla lacus metus  [RSS Feed](#) luctus eget.

Figure 3-3: Link icons applied with the Ending Selector

To achieve this effect using CSS2, you would have to apply set class values to the markup (`class="pdf"`, for example). The advantage of using the Ending Selector is that links to files can be detected automatically, without an end user having to apply a particular class. The disadvantage is that sometimes the file-type suffix is not at the end of the URI. However, the next new selector helps get around that shortcoming.

Arbitrary Substring Attribute Value Selector

The final new attribute selector—which I’ll call the *Arbitrary Selector*—works in the same way as the previous two, but it searches for the provided substring value *anywhere* inside the specified attribute string. This selector uses the asterisk (*) character. Here’s the new syntax:

```
E[attr*='value'] {}
```

To demonstrate this selector, I’ll once again use the markup from my first example:

```
<p>Lorem <a href="http://example.com/" title="Image Library">ipsum</a></p>
<p>Lorem <a href="http://example.com/" title="Free Image Library">ipsum</a></p>
```

and provide a value to the new selector:

```
a[title*='image'] {}
```

As with the Ending Selector, the rule applies to both `a` elements. However, this time it’s applied because they both contain the word *image* in their title attributes, even though the word appears in a different position in each example.

You may notice that this selector is somewhat similar to the Partial Attribute Value Selector from CSS2, and, indeed, in this example they are interchangeable. But the two selectors differ in a major way. In the example markup, with CSS3, I can also use just a small piece of the string:

```
a[title*='im'] {}
```

And the rules are still applied. The Partial Attribute Value Selector requires that the user enter a value that matches a full item in a space-separated list—in the example that would be either *free*, *image*, or *library*—so the `im` value would not be found anywhere in the markup in CSS2.

To continue with the examples provided for the first two attribute selectors, the Arbitrary Selector is also useful for adding file-type icons to URIs that have parameters at the end. Consider this fairly typical URI:

```
<a href="http://example.com/example.pdf?something=something">Lorem</a>
```

If you use the Ending Selector with a value of `pdf`, this element would not be recognized as a valid target, even though the file type is a PDF, because the value does not appear at the very end of the string. Providing the same value using the Arbitrary Selector does the trick, however:

```
a[href*='.pdf'] { background-image: url('pdf.png'); }
```

The `.pdf` substring value occurs within the specified attribute, so the icon is applied. You can see this illustrated in Figure 3-4.

Lorem ipsum dolor  PDF sit amet.

Figure 3-4: Link icon applied with the Arbitrary Selector

This selector is the most flexible of the three new attribute selectors as it takes a more wildcard approach to searching within strings. But the extra flexibility means you must take more care when defining the values provided to the selector; simple combinations of letters are far more likely to occur when you can match anywhere within a string.

Multiple Attribute Selectors

You can also chain multiple selectors together, which allows you to be very specific. Using multiple selectors, you can create rules to apply to attributes with values defined for the start, end, and anywhere in between. Imagine, for example, that you had links to two files with identical names but that were located in different folders:

```
<p><a href="http://example.com/folder1/file.pdf">Lorem ipsum</a></p>  
<p><a href="http://example.com/folder2/file.pdf">Lorem ipsum</a></p>
```

If you want to specify a rule to apply to only the second `p` element, you can chain some selectors together:

```
a[href^='http://'][href*='/folder2/'][href$='.pdf'] { }
```

This code looks for a elements that have an `href` attribute beginning with `http://`, ending with `.pdf`, and with `/folder2/` in the middle. Very specific!

The General Sibling Combinator

Our final new DOM selector in CSS3 is a combinator, which you'll recall means that it joins together more than one selector. The General Sibling Combinator is an extension of the Adjacent Sibling Combinator, which was introduced in CSS2. The syntaxes differ by just a single character:

```
E + F {} /* Adjacent Sibling Combinator */
E ~ F {} /* General Sibling Combinator */
```

The difference between the two is subtle but important: Adjacent Sibling selects any element (*F*) that is immediately preceded by element (*E*) on the same level of the document tree, but General Sibling selects any element (*F*) that is preceded by element (*E*) on the same level of the tree, regardless of whether it is immediately adjacent.

If that still sounds confusing, I'll try to explain with an example. Let's start with this CSS:

```
h2 + p { font-weight: bolder; } /* Adjacent Sibling */
h2 ~ p { font-style: italic; } /* General Sibling */
```

And apply it to the following markup (truncated for clarity):

```
❶ <p>Next we're going to discuss...</p>
<h2>Ren&eacute; Descartes</h2>
❷ <p>A highly influential French philosopher...</p>
❸ <p>He once famously declared:</p>
<blockquote>
❹ <p>I think, therefore I am.</p>
</blockquote>
❺ <p>However, this presumes the existence of the speaker.</p>
```

You can see the outcome in Figure 3-5. In the CSS, I'm using the Adjacent Sibling Combinator to bold the *p* element immediately adjacent to the *h2* element—that is, element ❷. I'm using the General Sibling Combinator to italicize all the *p* elements following the *h2* element, which applies to elements ❸ and ❺.

Elements ❶ and ❹ have no rules applied to them. Why not? Element ❶ precedes the *h2* element, and element ❹ is inside a *blockquote* element and, therefore, on a different level (the level below) in the document tree, so neither is affected by the rules.

To achieve the desired effect of only italicizing the paragraphs on the same level as the *h2* element in CSS2, without the General Sibling Combinator, you would need to set all *p* elements to display in italics and then add an extra rule for the *p* inside the *blockquote* to overrule the inheritance:

```
p { font-style: italic; }
blockquote p { font-style: normal; }
```

Next we're going to discuss a man who has been dubbed the "Father of Modern Philosophy" and the "Father of Modern Mathematics."

René Descartes

A highly influential French philosopher, mathematician, scientist, and writer.

He once famously declared:

|| I think, therefore I am.

However, this presumes the existence of the thinker.

Figure 3-5: Showing the difference between the Adjacent Sibling and General Sibling Combinators

You probably won't need to use the General Sibling Combinator often because much of its function overlaps with the basic DOM selectors. That said, you will still discover plenty of occasions where this can save you a little bit of code (and time).

Summary

Although attributes are a key feature of HTML4, most of them accept only a limited range of values, so many of them do not really require the use of the attribute selectors I've introduced in this chapter. Aside from the href attribute, only a handful of attributes accept more verbose values (alt, class, id, rel, and title are the ones that spring to mind). But, as I mentioned before, HTML5 is on the horizon and is set to introduce new attributes like datetime and pubdate that will allow you to be more creative with selectors.

The new selectors introduced in this chapter, along with those from previous versions of CSS, provide ways to traverse the document tree based on defined elements and attributes. Of course, on occasion the markup alone isn't sufficient for your purposes, and then you need to add classes or nonsemantic elements to act as hooks to hang your styles on. In the next chapter, you'll discover how CSS3 removes that need.

Selectors: Browser Support

	WebKit	Firefox	Opera	IE
Attribute Selectors	Yes	Yes	Yes	Yes
General Sibling Combinator	Yes	Yes	Yes	Yes

4

PSEUDO-CLASSES AND PSEUDO-ELEMENTS



The very first CSS specification, CSS1, introduced the concepts of *pseudo-classes* and *pseudo-elements*. These are selectors that act upon information about elements that extends (or sits outside of) the document tree, such as the state of a link or the first letter of a text node. A pseudo-class differentiates among an element's different states or types; these include—but are not limited to—those that provide information about link states: `:hover`, `:visited`, `:active`, and so on. A pseudo-element provides access to an element's subpart, which includes those pseudo-elements that select portions of text nodes, for instance, `:first-line` and `:first-letter`.

Though the selectors just mentioned have been around since CSS1, CSS2.1 added a handful more—although pseudo-element support has not been well implemented, if at all, until recently. CSS3 builds on these foundations with an expanded range of pseudo-classes, as well as a (slightly) tweaked syntax to differentiate pseudo-elements.

The advantage of having more methods for traversing documents should be clear: Fewer styling hooks are required. Markup like this is most likely familiar to you:

```
<ul>
  <li class="❶first ❷odd">❸<span>L</span>orem ipsum</li>
  <li>Lorem ipsum</li>
  <li class="odd">Lorem ipsum</li>
  <li class="❹last">Lorem ipsum</li>
</ul>
```

The markup contains class names to describe each element's position in the document tree: `first` (❶) and `last` (❹) show that the `li` elements are the first and last children of the `ul` element, and `odd` (❷) is used for the odd-numbered `li` elements. An extra `span` (❸) is included around the first letter of the first `li` element.

You mark up code like this when you want to add styles to alternating elements, set different values on the first and last elements, or add special formatting to the first letter of a text node. This markup detracts from the clarity and semantic meaning of your code, but in many cases you need it to provide the hooks to hang your styles on.

CSS3's new methods allow you to achieve the same visual results without muddying the markup with unnecessary classes and nonsemantic elements:

```
<ul>
  <li>Lorem ipsum</li>
  <li>Lorem ipsum</li>
  <li>Lorem ipsum</li>
  <li>Lorem ipsum</li>
</ul>
```

This change takes CSS a big step closer to achieving its stated goal: the separation of content and presentation.

Structural Pseudo-classes

As I stated in the introduction to this chapter, a pseudo-class provides a way to select an element based on information that is not specified in the document tree. Various subtypes are available, the most common of which is the *structural pseudo-class*. These subtypes are used to select elements that are not accessible using simple selectors. Take, for example, the following markup:

```
<div>
  <p>Lorem ipsum.</p>
  <p>Dolor sit amet.</p>
</div>
```

The first of the two `p` elements is the first child of the `div` element. That's obvious from the document tree, but the document tree doesn't provide any information that would allow you to apply a rule to that element only. CSS2 introduced the `:first-child` pseudo-class for exactly that reason:

```
E:first-child {}
```

This pseudo-class allows you to make a selection based on information that exists but isn't provided as an attribute of the element—the exact purpose of a pseudo-class. Since `:first-child` was introduced in CSS2, it has been the only pseudo-class of its type. But CSS3 extends the range greatly with the introduction of 11 new structural pseudo-classes.

The nth-* Pseudo-classes

Four of the new pseudo-classes are based on a count value used to find an element's position in the document tree; for this count, you use the syntax `nth-*`. Note that I've used the asterisk here in place of a number of different values, each of which I'll introduce throughout the rest of this chapter.

The basic syntax of the `nth-*` pseudo-classes is quite straightforward. By default, `n` represents a number that begins at 0 and increments by 1 (1, 2, 3, etc.). Another integer can be passed into it as a multiplier. For example, `2n` is every multiple of 2 (2, 4, 6, etc.), `3n` is every multiple of 3 (3, 6, 9, etc.), and so on:

```
E:nth-*(n) {}  
E:nth-*(2n) {}  
E:nth-*(3n) {}
```

The first example uses the default value `n`, so all elements of type `E` would be selected; in practice, this is the same as using a simple element selector. The next example selects every other `E` element, and the final example selects every third element of type `E`.

You may also use the mathematical operators for plus (+) and minus (-). So `2n+1` would be every multiple of two plus one (1, 3, 5, etc.), `3n-1` would be every multiple of three minus one (2, 5, 8, etc.):

```
E:nth-*(n+1) {}  
E:nth-*(2n+1) {}  
E:nth-*(3n-1) {}
```

The first example selects every element of type `E` except for the first instance; the count for this would be 2, 3, 4, 5, etc. The next example selects every odd-numbered `E` element (1, 3, 5, etc.). The final example, as mentioned above, selects elements in the sequence 2, 5, 8, etc.

Two special keyword values, `even` and `odd`, are also available; these can be used to replace $2n$ and $2n+1$, respectively:

```
E:nth-*(even) {}  
E:nth-*(odd) {}
```

With the basic syntax out of the way, let's move on to the pseudo-classes themselves.

nth-child and nth-of-type

Most of the new structural pseudo-classes allow you to select elements based on either their position in the document tree in relation to their parent element (`-child`) or their classification (`-of-type`). Often these definitions overlap, but there are crucial differences between them.

The simplest examples of these pseudo-classes are `nth-child` and `nth-of-type`. The first, `nth-child`, selects an element based on its position in a count of the total number of children in its parent element; `nth-of-type` bases its count not on the total children, but only those of the specified element type.

-
- ❶ `E:nth-child(n) {}`
 - ❷ `E:nth-of-type(n) {}`
 - ❸ `E:nth-child(2n) {}`
 - ❹ `E:nth-of-type(2n) {}`
-

In this example, rules ❶ and ❷ are equivalent because the count value (n) is left at the default; both of these simply select all child elements of type E . The difference reveals itself in the later examples: in ❸, `nth-child(2n)` selects all elements of type E from a count that includes all its siblings but only where those elements are even-numbered. In ❹, by comparison, `nth-of-type(2n)` selects all even-numbered elements of type E from a count that includes only those elements.

These rules are much easier to demonstrate than they are to explain. Take a look at these two rules:

-
- ❶ `p:nth-child(2n) { font-weight: bolder ; }`
 - ❷ `p:nth-of-type(2n) { font-weight: bolder; }`
-

I'll demonstrate the difference between them using this markup (text has been truncated for clarity):

```
<div>  
  <h2>The Picture of Dorian Gray</h2>  
  <p>The artist is the creator...</p>  
  <p>To reveal art and conceal the artist...</p>  
  <p>The critic is he who can translate...</p>  
</div>
```

The `div` element has a total of four children: one `h2` element and three `p` elements. The result of my first example rule (❶) is shown in Figure 4-1. The `nth-child(2n)` selector makes bold every second child (the first and third paragraphs). Compare that to the result of my second example rule (❷) in Figure 4-2. The `nth-of-type(2n)` selector ignores the `h2` and applies a bold weight to every second instance of the three elements of type `p`—that is, only the second paragraph.



Figure 4-1: The result of using the `nth-child` selector



Figure 4-2: The result of using the `nth-of-type` selector

As I mentioned before, and as you can no doubt deduce from the previous examples, `nth-child` and `nth-of-type` have a fair bit of overlap, and you can often use them interchangeably, as I do in the following example.

Figure 4-3 is a table showing the five-day weather forecast for London (so temperatures are given in degrees Celsius; 0°C equals 32°F). These figures were taken in January—it's not *always* this cold here! All of the information I want to convey is in the table, but without any kind of visual formatting the table is difficult to read.

Day	Weather	Max. Day (°C)	Min. Night (°C)	Wind (mph)
Sun	Sunny	8	4	8
Mon	Grey Cloud	7	4	6
Tue	Grey Cloud	5	2	13
Wed	Sleet	2	1	8
Thu	Heavy Rain	4	2	7

Figure 4-3: HTML table of a weather forecast

Now compare this table to the one shown in Figure 4-4. Here, I used *zebra striping* to aid the eye along the row and center-aligned the numeric values below their column headings for legibility.

Day	Weather	Max. Day (°C)	Min. Night (°C)	Wind (mph)
Sun	Sunny	8	4	8
Mon	Grey Cloud	7	4	6
Tue	Grey Cloud	5	2	13
Wed	Sleet	2	1	8
Thu	Heavy Rain	4	2	7

Figure 4-4: Weather forecast table formatted for readability

All of the formatting was done with a few CSS3 declarations:

```
tbody tr:nth-of-type(even) { background-color: #DDD; }
thead th, tbody td { text-align: center; }
thead th:nth-child(-n+2), tbody td:first-of-type { text-align: left; }
```

An extract of markup for this table appears next; so as not to obfuscate the example, I'm only showing the heading row and one row of the table proper:

```
<table>
<thead>
<tr>
  <th>Day</th>
  <th>Weather</th>
  <th>Max. Day (°C)</th>
  <th>Min. Night (°C)</th>
  <th>Wind (mph)</th>
</tr>
</thead>
<tbody>
<tr>
  <th>Sun</th>
  <td>Sunny</td>
  <td>8</td>
  <td>4</td>
  <td>8</td>
</tr>
```

```
</tbody>
</table>
```

I achieved the zebra striping using `nth-of-type` to change the background color of the even rows to a light gray. (I could also have used `nth-child` for this, as in my table markup, all of the `tr` elements are the only siblings on the same level of the document tree.) Then I used standard element selectors to center-align all of the head and body elements, before using `nth-child` (with a negative value) and `first-of-type` (which I'll discuss shortly) to left-align the cells containing text.

The negative value (`-n`) increments the count negatively—it starts at 0 and then counts down to `-1`, `-2`, and so on. This technique is useful when used with a positive second value—I've used `+2` here—so the count effectively begins at 2 and counts down, allowing me to select the second and first elements that are children of the `thead` element.

I'll use this technique again in the first example of the next section.

`nth-last-child` and `nth-last-of-type`

The `nth-last-child` and `nth-last-of-type` pseudo-classes accept the same arguments as `nth-child` and `nth-of-type`, except they are counted from the last element, working in reverse. For example, say that I want to use some visual shorthand to show in my weather table that the forecasts for days four and five are less certain than for the preceding days, as in Figure 4-5.

Day	Weather	Max. Day (°C)	Min. Night (°C)	Wind (mph)
Sun	Sunny	8	4	8
Mon	Grey Cloud	7	4	6
Tue	Grey Cloud	5	2	13
<i>Wed</i>	<i>Sleet</i>	2	1	8
<i>Thu</i>	<i>Heavy Rain</i>	4	2	7

Figure 4-5: Extra formatting using `nth-last-child`

In Figure 4-5, I italicized the characters in the last two rows by using the `nth-last-child` pseudo-class (although, once again, `nth-last-of-type` would serve just as well in this example), passing an argument of `-n+2`:

```
tbody tr:nth-last-child(-n+2) { font-style: italic; }
```

I used the negative value (`-n`) here to increment the count negatively, which has the effect of acting in reverse. Because `nth-last-child` and `nth-last-of-type` count backward through the tree, using a negative value here makes the count go forward! The count starts at the last `tr` element in the table and counts up in reverse order, so the last and penultimate lines are the first two counted and are, therefore, italicized. This may seem counterintuitive, but it'll become second nature as you traverse the document tree.

first-of-type, last-child, and last-of-type

In the example code I used for Figure 4-4, I introduced `first-of-type`. This is similar to `first-child`, which was introduced in CSS2, and indeed, for the example I provided, you could use them in the same way. In practice, however, they are as different as `nth-child` and `nth-of-type`.

As you're no doubt aware, the `first-child` pseudo-class is a selector used to apply rules to an element that is the first child of its parent. As with `nth-of-type`, however, `first-of-type` is more specific, applying only to the element that is the first child of the named type of its parent. A pair of counterpart pseudo-classes is also available, `last-child` and `last-of-type`, which—as you might have guessed—select the last child element or the last child element of that type, respectively, of the parent.

I'll show two examples to demonstrate the difference. Both of the examples will be applied to the same chunk of markup (I've truncated the text for clarity):

```
<div>
  <h2>Wuthering Heights</h2>
  <p>I have just returned...</p>
  <p>This is certainly...</p>
  <p>In all England...</p>
  <h3>By Emily Bronte</h3>
</div>
```

In the first example I'm going to use `first-child` and `last-child`, as shown here:

```
first-child { text-decoration: underline; }
last-child { font-style: italic; }
```

The result is shown in Figure 4-6. The `h2` element is the first child of the `div`, so it has an underline applied to it. The last child of the `div` is the `h3` element, so that is italicized. All quite straightforward.

Wuthering Heights

I have just returned from a visit to my landlord—the solitary neighbour that I shall be troubled with.

This is certainly a beautiful country!

In all England, I do not believe that I could have fixed on a situation so completely removed from the stir of society.

By Emily Bronte

Figure 4-6: Applying the `first-child` and `last-child` selectors

Now let's see the difference when we use the first-of-type and last-of-type selectors:

```
:first-of-type { text-decoration: underline; }  
:last-of-type { font-style: italic; }
```

Take a look at the result in Figure 4-7. You'll notice that three elements—h2, h3, and the first p—are underlined. This is because they are the first instance of that element type. Likewise, the h2, h3, and last p are all italicized. Again, this is because they are all the last element of that type; the h2 and h3 are both the first and last of their type, and so both rules are applied to them.

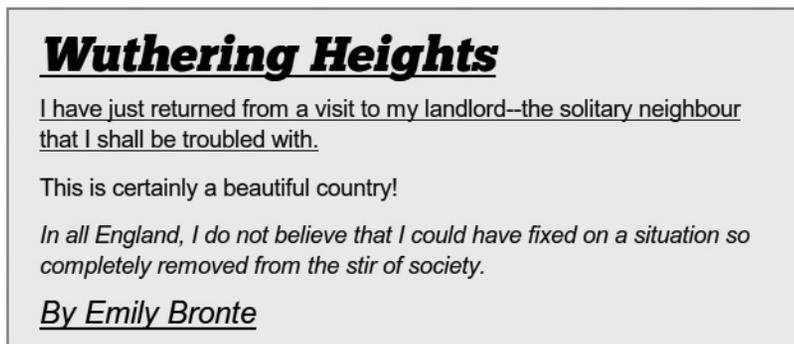


Figure 4-7: Applying the first-of-type and last-of-type selectors

As with all of the *-type and *-child pseudo-classes, the distinction is subtle, and sometimes the last child element is also the last of its type, so the selectors are interchangeable. But as I've just shown, at times, they have different applications.

only-child and only-of-type

These two pseudo-classes are used to select elements in the document tree that have a parent but either no sibling elements (*only-child*) or no siblings of the same element (*only-of-type*). As with many of the previous pseudo-classes, these two overlap substantially in function, but the following example should illustrate the difference between them. Take the following style rules:

```
p:only-of-type { font-style: italic; }  
p:only-child { text-decoration: underline; }
```

And then apply them to this markup:

```
<h2>On Intelligence</h2>  
<p>Arthur C. Clarke once said:</p>  
<blockquote>  
  <p>It has yet to be proven that intelligence has any survival value.</p>  
</blockquote>
```

You can see the result in Figure 4-8.

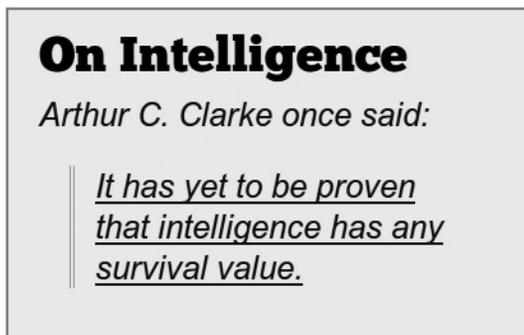


Figure 4-8: Comparing *only-child* and *only-of-type*

Both `p` elements are the only elements of their type in their level of the document tree, so the *only-of-type* rule selects both and italicizes them. The `p` element inside the `blockquote`, however, is also the only child in its level, so it's also subject to the *only-child* rule that applies the underline.

Using *only-of-type* allows you to pick an element from among others, whereas *only-child* requires the element to sit alone.

Other Pseudo-classes

In addition to the structural pseudo-classes discussed so far in this chapter, CSS3 introduces a number of pseudo-classes that allow you to select elements based on other criteria. These include link destinations, user interface elements, and even an inverse selector that permits selection based on what an element *isn't*!

target

On the Web, sites don't just link between pages but also provide internal links to specific elements. A URI can contain a reference to a unique ID or a named anchor. For example, if you had the following markup in a page:

```
<h4 id="my_id">Lorem ipsum</h4>
```

you could refer to it with the following link:

```
<a href="http://www.example.com/page.html#my_id">Lorem</a>
```

The target pseudo-class allows you to apply styles to the element when the referring URI has been followed. If you want to apply styles to the example element when the example URI is clicked, you would use:

```
#my_id:target {
```

A popular style is to highlight the subject of an internal link visually to provide a clear cue to the user. Consider, for example, the standard pattern for blog comments, which are somewhat like this simplified markup:

```
<div class="comment" id="comment-01">
  <p>Thanks for this scintillating example!</p>
  <p class="author">N.E. Boddy, April 13</p>
</div>
```

And another fairly common pattern is to include links to individual comments:

```
<p><a href="#comment-02">Latest comment</a></p>
```

Using the target pseudo-class, you can easily highlight the comment that the user wants to read:

```
.comment:target { background-color: #DDD; }
```

Figure 4-9 shows a list of comments in two states: on the left, as they appear before the referring link has been clicked, and on the right, after clicking the link the element that the link refers to has had its background color changed by the target selector.

Latest comment



Thanks for this scintillating example! N.E. Boddy, April 13
Meh. Sum1, April 14
CSS3 is going to be great! Joe Blogs, April 14

Latest comment

Thanks for this scintillating example! N.E. Boddy, April 13
Meh. Sum1, April 14
CSS3 is going to be great! Joe Blogs, April 14

Figure 4-9: Highlighting applied with the target pseudo-class

empty

The empty pseudo-class selects an element that has no children, including text nodes. Consider this markup:

```
<tr>
<td></td>
<td>Lorem ipsum</td>
<td><span></span></td>
</tr>
```

If you apply this CSS rule:

```
td:empty { background-color: red; }
```

the rule would be applied to only the first td element, as the other two contain a text node and a child element, respectively.

root

The root pseudo-class selects the first element in a document tree, which is only really useful if you're adding a stylesheet to XML documents—in HTML, the root will always be the html element. One small advantage of using root in HTML is that you can use it to give a higher specificity to the html element, which could be useful if you need to override the simple type selector:

```
html {} /* Specificity: 1; */
html:root {} /* Specificity: 2; */
```

Let's say you're creating a base stylesheet and want to set a property on the html element, which shouldn't be altered; in this case, you would use something like this:

```
html:root { background-color: black; }
```

The higher specificity gives precedence over any other properties applied to the html element, such as:

```
html { background-color: white; }
```

It's unlikely that you'll need to use this in most situations, however.

not

The negation pseudo-class (not) selects all elements *except* those that are given as the value of an argument:

```
E :not(F) {}
```

This rule selects all children of element *E* except for those of type *F*. For example, to color all the immediate child elements of a `div`, except for `p` elements, you would use this:

```
div > :not(p) { color: red; }
```

To see how useful `not` is, consider a situation where you have the following markup:

```
<div>
  <p>Lorem ipsum dolor sit amet...</p>
  <p>Nunc consectetur tempor justo...</p>
  <p>Nunc porttitor malesuada cursus...</p>
</div>
```

Now imagine you want to italicize all of the child `p` elements except for the first one.

To do this with CSS2, you would apply a style to all the `p` elements and then apply a further style to reset the first element back to its previous state:

```
p { font-style: italic; }
p:first-child { font-style: normal; }
```

With `not`, you can reduce that to a single rule:

```
p:not(:first-child) { font-style: italic; }
```

The argument that's passed into `not` must be a simple selector—therefore combinators (such as `+` and `>`) and pseudo-elements (which I'll discuss later in this chapter) are not valid values.

UI Element States

UI element states are used to select *user interface (UI)* elements based on their current state. Although HTML5 proposes a new range of UI elements (such as `command`), in HTML4, only form elements are able to have states:

```
<textarea disabled="disabled"></textarea>
<input checked="checked" type="checkbox">
```

The `textarea` has a `disabled` attribute, and the `input` of type `checkbox` has a `checked` attribute. No `enabled` attribute exists—elements that are not disabled are, by definition, classed as `enabled`, so an `enabled` pseudo-class is also available, giving you three user-state pseudo-class selectors:

```
:checked {}
:disabled {}
:enabled {}
```

To see the effect of these, consider the following style rules:

```
input[type='text']:disabled { border: 1px dotted gray; }  
input[type='text']:enabled { border: 1px solid black; }
```

I'm going to apply these rules to a form that has two text input elements, one of which has a disabled attribute (the form isn't valid as I don't have labels for the inputs, but that only clouds the issue in this demonstration):

```
<form action="">  
<fieldset>  
<legend>UI element state pseudo-classes</legend>  
<input type="text" value="Lorem ipsum" disabled>  
<input type="text" value="Lorem ipsum">  
</fieldset>  
</form>
```

You can see the results in Figure 4-10.

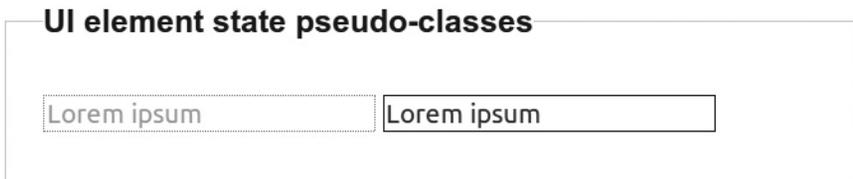


Figure 4-10: Disabled and enabled element states

As you can see, the disabled form element has grayed-out text (which is done automatically by the browser) and a gray dotted border (which I set in the stylesheet). I set a solid black border around the enabled element.

I haven't given a demonstration of the checked state here, as most browsers have very different interpretations of which style rules can be applied to checkbox inputs. For a comprehensive overview of cross-browser styling of form elements, I highly recommend "Styling Form Controls with CSS" from the blog 456 Berea Street (<http://www.456bereastreet.com/lab/styling-form-controls-revisited/>).

The specification notes that an idea is under consideration by the W3C to feature a fourth UI element state of *indeterminate*, which would be used in the occasional circumstance in which an element could be either enabled or disabled or has not yet been set into one of these states. At the time of writing, *indeterminate* is not an official part of the CSS3 specification.

Pseudo-elements

Like pseudo-classes, pseudo-elements provide information that is not specified in the document tree. But where pseudo-classes use "phantom" conditions such as an element's position in the tree or its state, pseudo-elements go further and allow you to apply styles to elements that don't exist in the tree at all.

In CSS2, the four pseudo-elements are `:first-line` and `:first-letter`, which select subelements in text nodes, and `:after` and `:before`, which allow the application of styles at the beginning and end of existing elements. CSS3 doesn't introduce any new pseudo-elements, but it refines the definitions slightly and introduces a new syntax to differentiate them from pseudo-classes. In CSS3, pseudo-elements are prefixed with a double colon (`::`), like so:

```
::first-line {}  
::first-letter {}  
::after {}  
::before {}
```

NOTE *The single colon syntax is still accepted for reasons of backward compatibility, although it is deprecated and you shouldn't use it going forward.*

The selection pseudo-element

At one stage in the formation of the CSS3 Selectors module, a proposal was made for a selection pseudo-element, which could be used to apply rules to an element that the user had selected in the browser (for example, a portion of a text node):

```
::selection {}
```

Only a limited number of properties can be applied with `selection`: `color`, `background-color`, and the background shorthand (although not `background-image`). Using `selection`, you can do something like this:

```
p::selection {  
    background-color: black;  
    color: white;  
}
```

Figure 4-11 shows a comparison of the system-native selection colors (top) and the colors I've applied with the `selection` pseudo-element (bottom).



Figure 4-11 shows two lines of text. The top line is "Lorem ipsum dolor sit amet..." where the words "ipsum dolor" are highlighted with a light gray background. The bottom line is "Lorem ipsum dolor sit amet..." where the words "ipsum dolor" are highlighted with a black background and white text.

Figure 4-11: Custom colors applied with the `selection` pseudo-element

Although at the time of writing `selection` has been dropped from the specification and its future would appear to be unknown, Opera and WebKit have already implemented it, Firefox has implemented it with the `-moz-` prefix, and it's expected to appear in IE9—so despite it no longer being part of the specification, you can continue to use it.

Summary

The new range of pseudo-classes (and any pseudo-elements that may be defined in the future) makes document traversal far more flexible and powerful with CSS3 than it ever was with its predecessors.

Some debate has arisen about whether the expanded range of pseudo-class selectors is really necessary, but I believe much of this debate comes from developers who weren't familiar with them, as they weren't yet implemented across all browsers. Internet Explorer is now the only browser to not have implemented pseudo-class selectors, but with IE9 promising to rectify this, I think the new selectors will shortly prove worthwhile.

DOM and Attribute Selectors: Browser Support

	WebKit	Firefox	Opera	IE
Structural pseudo-classes	Yes	Yes	Yes	No (expected in IE9)
:target	Yes	Yes	Yes	No (expected in IE9)
:empty	Yes	Yes	Yes	No (expected in IE9)
:root	Yes	Yes	Yes	No (expected in IE9)
:not	Yes	Yes	Yes	No (expected in IE9)
Pseudo-elements (new syntax)	Yes	Yes	Yes	No (expected in IE9)
UI element states	Yes	Yes	Yes	No (expected in IE9)
:selection	Yes	Yes	Yes	No (expected in IE9)

5

WEB FONTS



The features covered in this chapter are the oldest in this book, having been introduced in CSS2 many years ago—only to be dropped from the 2.1 spec due to a lack of implementation by browser makers. Now a new generation of browsers has revived interest in improving the typographical options available to web designers, and I, for one, welcome the return of these features in CSS3. Chief among them is the ability to specify fonts that don't already exist on the user's system—by utilizing the `@font-face` method—which frees designers from the yoke of the standard palette of “web-safe” system fonts that have been used for many years. Surprisingly, however, this capability has been available in Internet Explorer since 1997!

IE4 was the first browser to allow web fonts, but it did so with a proprietary format that prevented other browsers from following suit. Microsoft has since submitted its format to the W3C for consideration as a standard, but now that Firefox, Safari, Chrome, and Opera are all backing different formats, it's unlikely that anything will come from this.

The CSS Fonts Module Level 3 (<http://www.w3.org/TR/css3-fonts/>) currently has Working Draft status and probably won't become a recommendation any time soon. Most of the spec has already been implemented in modern browsers (with a couple of exceptions, which I'll cover later), so you can consider it pretty safe to use.

The @font-face Rule

To display web (or nonsystem) fonts on your pages, you need to use the @font-face rule. This rule defines the font and provides the browser with the location of the file to use. Here's the basic syntax:

```
@font-face {  
  ❶ font-family: name;  
  ❷ src: ❸local('fontname'), ❹url('/path/filename.otf') ❺format('opentype');  
}
```

I'll break this down a little. First, I give the font a name with the font-family property (❶). This property is one you should be familiar with, although it serves a slightly different purpose here; I'm using it to declare a font name, not to refer to one. Just like the font-family property in CSS2.1, you can use multiple, space-separated words as long as you enclose them within single quote characters.

NOTE *When you define your font's name with font-family, you can use the same name multiple times—in fact, sometimes you'll need to. I'll discuss why shortly, in “Defining Different Faces” on page 51.*

Next is the src property (❷), which tells the browser the location of the font file. This property accepts a few different values: local (❸) uses the name of the source font to check if the font is already installed on the user's machine, url (❹) provides a path to the font if it's not available locally, and format (❺) specifies the font type. In this example, I've used OpenType, but more types are available, and I'll discuss those later in “Font Formats” on page 55.

I can supply many different values for the src property by separating the values with commas, as I've done in the code example. This utilizes the power of the cascade to allow different fall-back values, which will come in handy in a later example.

To use the font I just defined, I call only its name in the font stack, as I'd normally do:

```
E { font-family: FontName; }
```

For a real-world example, I'll apply the Chunk font (available to download for free from <http://www.theleagueofmoveabletype.com/fonts/4-chunk/>) to an h2 element using @font-face.

Here's the code I'll use in my stylesheet:

```
@font-face {  
  ❶ font-family: Chunk;  
    src: ❷local('ChunkFive'), ❸url('ChunkFive.ttf') ❹format('truetype');  
}  
❺ h2.webfont { font-family: ChunkFive, sans-serif; }
```

The first step is to name my font; I've chosen Chunk (❶) because I can remember it easily, but I could use any name. Next I provide values to the `src` property: `local` uses the true name of the font, 'ChunkFive' (❷), to check if it's available on my system. Following that I enter a relative path to the font file I want to use (❸), and finally, I assign an argument of `truetype` to the `format` value (❹).

NOTE *You can usually find a font's true name by using a font management application or by right-clicking the font file to view the font information.*

In the last line of CSS (❺), I give my new font name as a value to the `font` stack to be applied to `h2` elements with a class of `webfont`. To see how that displays, here's a quick comparison using the following markup:

```
<h2>Alas, poor Yorick!</h2>  
<h2 class="webfont">Alas, poor Yorick!</h2>
```

You can see the output in Figure 5-1.

Defining Different Faces

The `@font-face` syntax we've seen so far in this chapter is pretty straightforward, but it only defines one font face—that

is, permutation of weight, slope, and so on. If you want to use a different face, such as a bolder weight or an italic type, you have to define each font face individually. You can use one of two methods to do this. The first method re-uses the same name and adds extra descriptors to the `@font-face` rule:

```
@font-face {  
  ❶ font-family: 'Museo Sans';  
    src: local('Museo Sans'), url('❷MuseoSans_500.otf') format('opentype');  
}  
@font-face {  
  ❸ font-family: 'Museo Sans';  
  ❹ font-style: italic;  
    src: local('Museo Sans'), url('❺MuseoSans_500_Italic.otf') format('opentype');  
}  
h2 { font-family: 'Museo Sans', sans-serif; }
```

Alas, poor Yorick!

Alas, poor Yorick!

Figure 5-1: The Chunk Five font (the bottom row), called using the `@font-face` rule, compared to a system font (the top row)

Here, you can see that the first `@font-face` rule defines the font name as Museo Sans (❶) and gives the URL of the regular face (❷). The second `@font-face` rule uses the same font name (❸) but adds the `font-style` property with the italic value (❹), and the URL points at the font’s italic face (❺). The advantage of this approach is that the italic style is applied automatically and appropriately, without your having to define it in the CSS, as in this example markup:

```
<h2>I knew him, Horatio</h2>
<h2><em>I knew him, Horatio</em></h2>
```

The second `h2` element uses the italic font face defined in the previous rules (you can see the result in Figure 5-2).

The second method uses unique names for each font style and calls them individually in the font stack:

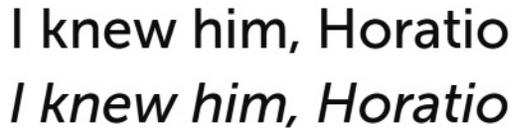


Figure 5-2: Museo Sans Regular (top) and Italic (bottom) applied using `@font-face`

```
❶ @font-face {
  font-family: 'Museo Sans';
  src: local('Museo Sans'), url('MuseoSans_500.otf') format('opentype');
}
❷ @font-face {
  font-family: 'Museo Sans Italic';
  src: local('Museo Sans'), url('MuseoSans_500_Italic.otf') format('opentype');
}
h2 { ❸font-family: 'Museo Sans', sans-serif; }
h2 em {
  ❹ font-family: 'Museo Sans Italic', sans-serif;
  ❺ font-style: normal;
}
```

In this example, the first `@font-face` rule (❶) is named Museo Sans, and the URL points to the regular font face. The second `@font-face` rule (❷) is named Museo Sans Italic, and the URL points to the italic face. The font stack for the `h2` element (❸) uses the regular face, whereas the font stack for the `em` element, which is a child of `h2` (❹), uses the italic face.

You’ll notice that I also added a `font-style` property with the value of `normal` (❺), as otherwise Firefox artificially italicizes the italic—giving you a double italic!

Which method you use is up to you; popular opinion on the Web seems to be swinging toward the second, but you should experiment and see which one most suits your purposes. In some situations, you may have no choice; for example, Safari doesn’t recognize the CSS2.1 `font-variant` property (used for applying small caps) when using the first method.

Regardless of which method you choose, Figure 5-2 shows the effect of using the italic style.

True vs. Artificial Font Faces

One thing to be aware of is that no matter which method you utilize, if you want to apply a different font face—for example, italic—make sure you define a link to the relevant file in your `@font-face` rule. If you don't, some browsers (notably Firefox) will attempt to re-create the face artificially, often with ugly results.

Here's an example of how *not* to define an italic weight:

```
@font-face {
  font-family: GentiumBookBasic;
  src: local('Gentium Book Basic Regular'), url('GenBkBasR.ttf') format('truetype');
}
h2 {
  font-family: GentiumBookBasic, sans-serif;
  font-style: italic;
}
```

You can see that my `@font-face` rule uses the regular face of the Gentium Basic font, but the `h2` element has an italic style declared on it. Before I show you how that renders, here's the way I should have done it:

```
@font-face {
  font-family: GentiumBookBasicItalic;
  src: local('Gentium Book Basic Italic'), url('GenBkBasI.ttf') format('truetype');
}
h2 { font-family: GentiumBookBasicItalic, sans-serif; }
```

In this code, the `@font-face` rule defines the italic face of Gentium Basic, which is then applied to the `h2` element. You can compare the two different approaches in Figure 5-3.

As you can see, the two examples are quite different. The first is the default font slanted to fake an italic (using the first code example); the characters are larger, slightly distorted, and spaced inconsistently. The second is the true italic font face (using the second code example), which uses characters designed specifically for this purpose.

Be aware of this issue when using Firefox and Internet Explorer; in situations where the “bad” code example is applied, WebKit ignores the `font-style` property and displays only the regular font face. The same warning applies for all the different font faces: bold, italic, bold italic, small caps, condensed, and so on.

A fellow of infinite jest

A fellow of infinite jest

Figure 5-3: Comparing an artificial italic (top) with a (true) italic font face

A “Bulletproof” @font-face Syntax

The @font-face rule carries with it some unfortunate legacy issues. I explained at the beginning of this chapter that the rule has been around for quite a while, having been implemented in Internet Explorer as far back as 1997. Also, although the current Fonts Module was updated in 2009, it had not been revised in seven years. These two facts alone account for some issues, but modern browsers also implement @font-face differently.

These issues mean you need a workaround to ensure that @font-face works correctly across all browsers. Luckily, some clever developers have done just that. But before I introduce the fully cross-browser “bulletproof” @font-face syntax, let me discuss briefly some of the problems it addresses.

Using Local Fonts

The local() value for the src property is used to check whether a user already has the defined font installed on his or her system—if the user does, the local copy can be applied rather than download a new copy. local() is a good idea, but it suffers from three drawbacks. The first drawback, and not least, is that local() isn’t supported by any versions of Internet Explorer below 9!

The workaround for this drawback depends on IE8 and below only allowing the *Embeddable Open Type (EOT)* font format (see “Font Formats” on page 55 for more on EOT). Because of EOT, you can declare the src property twice:

```
@font-face {  
    font-family: name;  
    ❶ src: url('filename.eot');  
    ❷ src: local('fontname'), url('filename.otf') format('opentype');  
}
```

The first instance (❶) has no local() value, and the EOT font format is used. In the second instance (❷), the local() value has the name of the required font, and the OTF (OpenType) format is used. This instance will be ignored by IE8 and below, which doesn’t recognize local(), but will take precedence in all other browsers.

The next drawback is that, at the time of writing, the Safari browser for Mac OS X (currently version 5.03) requires a different font-name argument for the local() value. Fonts generally have two names: a full name and a PostScript name. Safari on Mac requires the Postscript name, whereas every other browser accepts the full name. (This requirement is explained further in the Mozilla Hacks blog at <http://hacks.mozilla.org/2009/06/beautiful-fonts-with-font-face/>.) Working around this is easy: Just specify two local() values with a different argument in each:

```
@font-face {  
    font-family: name;  
    src: ❶local('fontname'), ❷local('altfontname'), url('filename.otf') format('opentype');  
}
```

For the first `local()` value (❶), you can use the PostScript name, and in the second (❷), you can use the family name (though the order you list them in actually doesn't matter). Again, a font management program will help you find the different names to specify in your stylesheets.

Font management programs can cause the third drawback, however. In some cases, the `@font-face` rule doesn't play nicely with font management software, displaying incorrect characters or opening a dialog to ask for permissions to use a font. The only way around this drawback is to add a "null" value to `local()`, forcing a download of the defined font. This null value need only be a single character and has become, by convention, a smiley face; it doesn't have to be a smiley face, but it works and it's friendly to look at!

So if you put together all the workarounds in this section, you end up with code that looks like this:

```
@font-face {
  font-family: name;
  src: url('filename.eot');
  src: local('☺'), url('filename.otf') format('opentype');
}
```

Font Formats

The next problem comes in the shape of different, and competing, formats. As I already mentioned, IE8 and below support only the proprietary EOT format. The new wave of interest in web fonts has come about because modern browsers—first Safari and then Firefox and then others—allowed the use of the more common *TrueType* and *OpenType* formats.

Many commercial font foundries won't allow their fonts to be used in this way as it makes illegal copying of their fonts a little easier (see "Licensing Fonts for Web Use" on page 57). For this reason, Mozilla consulted with some font makers and created the *Web Open Font Format (WOFF)*, which is supported from Firefox 3.6 onward and should also be supported in IE9 and forthcoming versions of Chrome and Opera (no official word yet from Safari, but I'm sure it'll be close behind). Some browsers also accept the *Scalable Vector Graphics (SVG)* font type. This font type is a vector re-creation of the font and is considerably lighter in file size, making it ideal for mobile use. As such, this format is the only one that older versions (4.1 and below) of Safari for iPhone allow.

Because the `@font-face` spec allows multiple values for the `src` property, you can create a stack to make sure the widest possible range of browsers is supported:

```
@font-face {
  font-family: name;
  src: local('☺'),
  url('filename.woff') format('woff'),
  url('filename.otf') format('opentype'),
  url('filename.svg#filename') format('svg');
}
```

The stack in the code collection of src values checks if the browser supports the WOFF, OpenType, and SVG formats (in that order) and displays the correct font accordingly.

The Final “Bulletproof” Syntax

In order to have your chosen font display the same in every browser on every platform, the required code looks something like this:

```
@font-face {
  font-family: 'GentiumBookBasicRegular';
  src: url('GenBkBasR.eot');
  src: local('☺'),
  url('GenBkBasR.woff') format('woff'),
  url('GenBkBasR.ttf') format('truetype'),
  url('GenBkBasR.svg#GentiumBookBasic') format('svg');
}
```

For this to work, the major requirement is that your chosen font be available in three or four different formats. To make this easier, I strongly recommend using the @font-face Generator by Font Squirrel (<http://www.fontsquirrel.com/fontface/generator/>). Simply upload the font file you want to use and @font-face Generator converts it into all the relevant formats—as well as generates the CSS you need to use in your pages. It’s an invaluable tool. Font Squirrel also has a library of fonts that are ready to use with @font-face embedding, saving you the task of converting.

To read more about the “bulletproof” @font-face syntax, visit the site of developer Paul Irish who is largely responsible for creating this syntax. He explains in detail why each part of it is necessary (<http://www.paulirish.com/2009/bulletproof-font-face-implementation-syntax/>).

The Fontspring Bulletproof Syntax

As this book was going to press a new, supposedly more bulletproof syntax was announced. This is simpler than the syntax shown above and relies on just a simple question mark (?) character applied to the EOT font source:

```
@font-face {
  font-family: name;
  src: url('filename.eot?') format(eot),
  url('filename.woff') format('woff'),
  url('filename.otf') format('opentype'),
  url('filename.svg#filename') format('svg');
}
```

To understand how the Fontspring syntax works, you should read the original blog post that announced it (<http://www.fontspring.com/blog/the-new-bulletproof-font-face-syntax/>). While this new syntax does seem to work, it has already changed a few times since it was announced and probably needs to be more fully tested before it becomes the new default, so I hesitate to endorse it fully.

Licensing Fonts for Web Use

As mentioned previously in “Font Formats” on page 55, many font foundries expressly forbid embedding their web fonts in your pages using @font-face. They forbid this because linked OpenType or TrueType fonts are easy to locate and download and can then be used illegally in both on- and offline applications. The new WOFF file type was created in response to this; WOFF is a web-only format and can contain licensing information to help track down copyright infringers. Many foundries have already committed to selling this format, and I hope many more will follow.

In general, the best policy is to check that the font you choose has a license explicitly allowing you to use it for web embedding; don’t assume that because a font is free to download, it is free to use online. That said, many good-quality free fonts that do allow embedding are available online; some examples are given in Appendix B.

While the licensing situation is in a state of flux, many web font service providers have created mechanisms to embed fonts legally in your pages. By adding JavaScript to your pages, the provider is authorized to serve the font files from their network, so you can call the font families in your stacks. The method is known as *Fonts as a Service (FaaS)*.

Some of these services are provided by the foundries themselves—Typotheque (<http://www.typoshede.com/webfonts/>) is one example—but probably the best known at the moment is Typekit (<http://www.typekit.com/>), a third-party provider serving fonts from many different foundries and creators. Typekit has a limited font set available for free, so you can use it on your personal website if you want to experiment.

I’ve provided many more links to font tools and services in Appendix B and will update the list on the website that accompanies this book.

A Real-World Web Fonts Example

Having discussed the intricacies and niceties of fonts, licensing, and multiple syntaxes, let’s put everything we’ve learned so far together and see @font-face in action. I’ll use an example that compares text displayed in a standard sans-serif font (Arial) to the same text displayed in three different font families (all from Font Squirrel).

Here’s the CSS for this example. Bear in mind that, for the sake of clarity, I’ve simplified this example to show only a single font format (TrueType):

```
@font-face {
  font-family: 'Candela';
  src: local(''), url('CandelaBook.ttf') format('truetype');
}
@font-face {
  font-family: 'Candela';
  font-style: italic;
  src: local(''), url('CandelaItalic.ttf') format('truetype');
}
```

```

@font-face {
  font-family: 'Candela';
  font-weight: bold;
  src: local(''), url('CandelaBold.ttf') format('truetype');
}
@font-face {
  font-family: 'ChunkFiveRegular';
  src: local(''), url('Chunkfive.ttf') format('truetype');
}
@font-face {
  font-family: 'AirstreamRegular';
  src: local(''), url('Airstream.ttf') format('truetype');
}
.font-face h1 { font-family: ChunkFiveRegular, sans-serif; }
.font-face h2 { font-family: AirstreamRegular, cursive; }
.font-face p { font-family: Candela, sans-serif; }

```

I also left out some color and size adjustments to keep the code as readable as possible. Here's the markup I used:

```

<h1>Great Expectations</h1>
<h2>By Charles Dickens</h2>
<p>My father's family name being <em>Pirrip</em>, and my Christian name
<em>Philip</em>, my infant tongue could make of both names nothing longer or
more explicit than <strong>Pip</strong>. So, I called myself <strong>Pip</
strong>, and came to be called <strong>Pip</strong>.</p>

```

You can see the output in Figure 5-4.

Great Expectations

By Charles Dickens

My father's family name being *Pirrip*, and my Christian name *Philip*, my infant tongue could make of both names nothing longer or more explicit than **Pip**. So, I called myself **Pip**, and came to be called **Pip**.

Great Expectations

By Charles Dickens

My father's family name being *Pirrip*, and my Christian name *Philip*, my infant tongue could make of both names nothing longer or more explicit than **Pip**. So, I called myself **Pip**, and came to be called **Pip**.

Figure 5-4: Text using a standard system font (left) and using different web fonts (right)

In the example on the right, I mixed three fairly distinctive font families—many designers will probably tell you mixing isn't a good idea on a production site, but it works well to illustrate my point. Whatever you think of my font choices, I hope you'll at least agree that the text looks more dynamic and enticing with those choices applied.

Despite the long and winding route we've had to take to get a cross-browser syntax, using `@font-face` isn't complicated; you'll expend some overhead in the form of extra setup time to declare the font faces and variants you want to use, but after that you can call them in your font stacks and style them in exactly the same way as system fonts.

More Font Properties

The CSS3 Web Fonts Module doesn't just re-introduce the `@font-face` rule; it also revives two other font properties that were first proposed for CSS2. These properties are potentially very useful for giving you granular control over your fonts—I say potentially because, as of this moment, they aren't widely implemented.

font-size-adjust

The only drawback to using font stacks in CSS is that fonts can vary so much in size; your first choice font may look great at 16px, but if that font's not available, the next fallback may be a lot smaller and harder to read. To combat this, you can use `font-size-adjust`. `font-size-adjust` lets you dynamically alter the `font-size` property to ensure a regular font size no matter which font is used from the stack. `font-size-adjust` takes a single decimal value. Here's the syntax:

```
E { font-size-adjust: number; }
```

The *number* value is the proportion of the total height that is occupied by a lowercase *x* character (known as the *x-height*). In other words, a font might be 16px high in total, but the height of the lowercase *x* might be half that (8px), which gives an *x-height* ratio of 0.5 (8 divided by 16):

```
p { font-size-adjust: 0.5; }
```

By using `font-size-adjust`, you can ensure that no matter what font is displayed, the *x-height* always has the same value and legibility does not suffer. To illustrate, consider the following code:

```
h2 { font: 360%/1 Georgia, serif; }  
h2.adjusted { font-size-adjust: 0.5; }  
h2.impact { font-family: Impact, serif; }
```

In the following example, I have three `h2` elements, all with the same values for `font-size`. I apply different values to them using their class names, which you can see in this markup:

```
<h2>Of most excellent fancy</h2>  
<h2 class="impact">Of most excellent fancy</h2>  
<h2 class="adjusted impact">Of most excellent fancy</h2>
```

The first `h2` is rendered in the Georgia font, the second in Impact, and the third in Impact but with the `font-size-adjust` property applied. You can see the results in Figure 5-5.

Of most excellent fancy

Of most excellent fancy

Of most excellent fancy

Figure 5-5: The effect of `font-size-adjust` on the Impact font (third line)

You can clearly see the difference between the Georgia (first line) and Impact (second line) fonts in the first two h2 elements. (Of course, you are unlikely to ever use these two in the same font stack, but because they have quite different *x*-heights, they're good for these purposes.) Georgia has an *x*-height ratio of approximately 0.5, which means the lowercase *x* is half the height of the font. By contrast, Impact has an *x*-height ratio of 0.7, which means less contrast in height between upper- and lowercase letters.

The first two lines in the example are not adjusted in any way, so Impact's lowercase characters in the second line are considerably taller than Georgia's in the first line. In the third line, however, I set the `font-size-adjust` value to 0.5 to match Georgia's:

```
h2.adjusted { font-size-adjust: 0.5; }
```

That adjusts the font size so Impact displays at a lower size—the *x*-height is 12px, half of the full 24px height. You can see this more clearly if I directly compare the two elements, as shown in Figure 5-6. Here, the characters without ascenders (lines that rise above the *x*-height)—that is, *e*, *x*, *c*, and *n*—of the adjusted Impact font are the exact same height as those of the Georgia font.

excellent excellent

Figure 5-6: Characters from the Georgia font (left) compared to characters from the Impact font adjusted with `font-size-adjust` (right)

Unfortunately, a font's *x*-height ratio isn't easily available; you can either use a graphics package to measure it manually or try to find an online resource. (I found an estimation tool at <http://www.cs.tut.fi/~jkorpela/x-height.html>; the site also lists values for common web fonts.)

At the time of writing, only Firefox supports this property.

font-stretch

Some font families contain condensed or expanded variants, and the `font-stretch` property was proposed to allow access to these. As of this writing, `font-stretch` has not been implemented in any released browser, although it is in the Beta of IE9.

Here's the syntax:

```
E { font-stretch: keyword; }
```

According to the specification, the *keyword* value can be any one of the following: normal (the default), ultra-condensed, extra-condensed, condensed, semi-condensed, semi-expanded, expanded, extra-expanded, and ultra-expanded. Each keyword relates to a font variant within a family, such as *Frutiger Condensed* or *Nova Ultra Expanded*. A pair of relative keywords, *narrower* and *wider*, were listed in the 2002 revision of the specification, and although removed since then, they have been implemented in IE9 Beta.

Let's see an example using the IE9 Beta. In this example, I'll use `font-stretch` to display two different faces of the font PT Sans, using the following code:

```
h2 { font: 360%/1 'PT Sans', sans-serif; }  
h2.narrow { font-stretch: narrower; }
```

The `h2` element is displayed twice, both times using the font PT Sans at 360 percent (36px) size. In the second instance, I've used the `font-stretch` property with the value `narrower`, which tells the browser to display any face that is narrower than the current one, without being specific about exactly which width I want. You can see the results in Figure 5-7.

Despite this property not having wide browser support yet, you can replicate the effect with the use of the `@font-face` rule (introduced at the beginning of this chapter) to specify a condensed or expanded face in your font stack.



Figure 5-7 shows two lines of text, "Lorem Ipsum Dolor". The first line is in a standard width, and the second line is narrower, demonstrating the effect of the font-stretch property.

Figure 5-7: The second example uses the narrow font face due to the effect of the `font-stretch` property in IE9.

OpenType Features

Although web typography takes a great leap forward in CSS3, it still just scratches the surface of the possibilities of type. If you compare the options available to you in a browser with what's available in a desktop application such as Adobe Illustrator, you'll see that the latter is much richer than the former.

Font formats such as OpenType are capable of much more than face or weight variations; they have a range of ligatures, swashes, special numeric characters, and much more. (If none of those terms makes any sense to you, I recommend Magnet Studio's *Beginners Guide to OpenType* at <http://www.magnetstudio.com/words/2010/opentype-guide/>).

The latest nightly builds of Firefox have an experimental new property that allows you to explore the extra features afforded by OpenType and other similar formats. The new property is called `-moz-font-feature-settings`. Here's its syntax:

```
E { -moz-font-feature-settings: "parameters"; }
```

The *parameters* value is a string that contains one or more comma-separated binary values—that is, the values are either 0 or 1. Here’s an example:

```
E { -moz-font-feature-settings: "kern=1"; }
```

The parameter `kern` has a value of 1. `kern` enables OpenType kerning and, in this case, is actually redundant as Firefox already sets this by default. If you want to disable kerning, you use the alternative binary value:

```
E { -moz-font-feature-settings: "kern=0"; }
```

You can, as I mentioned, have more than one parameter—just create a comma-separated list:

```
E { -moz-font-feature-settings: "kern=0,liga=1"; }
```

This property is very much in development, and there’s no guarantee that the final syntax will take this form.

Let’s look at some examples that show the advantage of using OpenType features. Earlier in the chapter in “True vs. Artificial Font Faces” on page 53, I showed why you should always use a true italic font rather than letting the browser create one artificially. In this example, you’ll see that the same principle holds when using the small caps font variant. Here are the relevant style rules:

```
h2.smallcaps { font-variant: small-caps; }  
h2.ot-smallcaps { -moz-font-feature-settings: "smcp=1"; }
```

Here, I use two `h2` elements: I apply the `font-variant` property with the `small-caps` value to the first; for the second, I use the new property, `-moz-font-feature-settings`, with the parameter used to toggle small caps, `smcp`. You can see the difference in Figure 5-8.

In the first `h2` element, which uses simulated small caps, the proportions are off; the difference between the large and small capitals is barely noticeable. Compare that with the second `h2` element, which has more obvious proportions and looks more pleasing to the eye.



Figure 5-8: The lower example uses OpenType’s own small caps feature

Now I’ll demonstrate the use of ligatures, which are used to join certain pairs of characters to make them appear more harmonious. As with kerning, Firefox automatically uses common ligatures unless instructed not to, so in this example, I’ll compare text with no ligatures, common ligatures, and discretionary—that is, more decorative—ligatures.

Here's the code:

```
h2.lig-none { -moz-font-feature-settings: "liga=0"; }  
h2.lig-common { -moz-font-feature-settings: "liga=1"; }  
h2.lig-disc { -moz-font-feature-settings: "dlig=1"; }
```

This code is applied to three h2 elements. The first has the parameter string `liga=0`, so common ligatures are disabled. The second uses the same parameter string except the value is 1, so common ligatures are enabled (I could have left this out as it's the default state). The third has the string `dlig=1`, which enables discretionary ligatures. Figure 5-9 compares the three.

Pay attention to the character pairs *Th*, *ft*, and *ct*. In the first h2 element, without ligatures, they are rendered as separate characters. In the second, with common ligatures, the *Th* and *ft* are joined together at the point where the characters almost meet. In the third h2 element, the ligatures are discretionary, so the *ft* is now joined in two places, and the *ct* pair is also joined with an extravagant flourish.

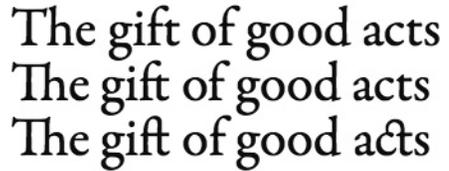


Figure 5-9: Comparing OpenType ligatures: (from top) none, common, and discretionary

You can see more examples of this property's possibilities at <http://hacks.mozilla.org/2009/10/font-control-for-designers/>. Although something like this is being discussed for inclusion in CSS3, whether this proposal will end up being the recommended syntax is not yet clear. This property is currently only available in Firefox nightly builds (and only on OS X and Windows) and is not guaranteed to make it into any full release of Firefox in the future.

Summary

While `font-size-adjust` and `font-stretch` will surely come in handy in the future, for now `@font-face` is the killer feature of the Web Fonts Module. However, `@font-face` is not without its drawbacks. For a start, bear in mind that every extra font you use adds to the page's load time, and a slight "flash" will occur as your new fonts take effect—that flash is the gap between the DOM loading and the fonts loading, and different browsers deal with it in different ways. Also, remember that misuse or overuse of different typefaces can lead to decreased legibility. Choose your fonts carefully, and check them closely across different browsers.

Despite those caveats, however, you can use this simple rule to amazing effect. In the next chapter, I'll show you some ways to further enhance your typography.

Web Fonts: Browser Support

	WebKit	Firefox	Opera	IE
@font-face	Yes	Yes	Yes	Yes
font-size-adjust	No	Yes	No	No
font-stretch	No	No	No	No (expected in IE9)
OpenType features	No	No (expected in Firefox 4)	No	No

6

TEXT EFFECTS AND TYPOGRAPHIC STYLES



Text content has been the backbone of the Web since its creation, yet for years we've had to make do with a very limited set of tools. CSS3 hugely expands its typographic toolset by introducing a range of new and updated features in the Text Module.

Chief among these features is the ability to add shadows to text. Although this addition doesn't sound particularly revolutionary—print typographers have been able to use shadows for a long time—the new syntax is flexible enough to allow for some very nice effects. A similar feature is text-outlining (or text-stroking), which, although not widely implemented, does increase the variety of options available when creating decorative headlines. In addition to these are some less flashy effects but ones that can really do wonders for the readability of your text.

The CSS Text Level 3 Module (<http://www.w3.org/TR/css3-text/>) hadn't been updated since 2007 (although a new version was released during the writing of this book) and was incomplete in some places. However, some elements of it are quite well implemented and ready for you to begin using straightaway.

Before I introduce the first new property in this module, however, I'm going to briefly introduce the concepts of coordinates and axes. If you're already familiar with these, feel free to skip this section; otherwise, read on.

Understanding Axes and Coordinates

One syntax concept that's new to CSS3 is that of the *axis* (or *axes* when you have more than one). You may know all about axes if you remember your math lessons, but if you're reading this section, I'll assume you need a refresher.

CSS uses the *Cartesian coordinate system*, which consists of two lines, one horizontal and one vertical, that cross each other at a right angle. Each of these lines is an axis: The horizontal line is known as the *x-axis*, and the vertical line is known as the *y-axis*. The point where the two lines meet is called the *origin*. You can see this illustrated in Figure 6-1.

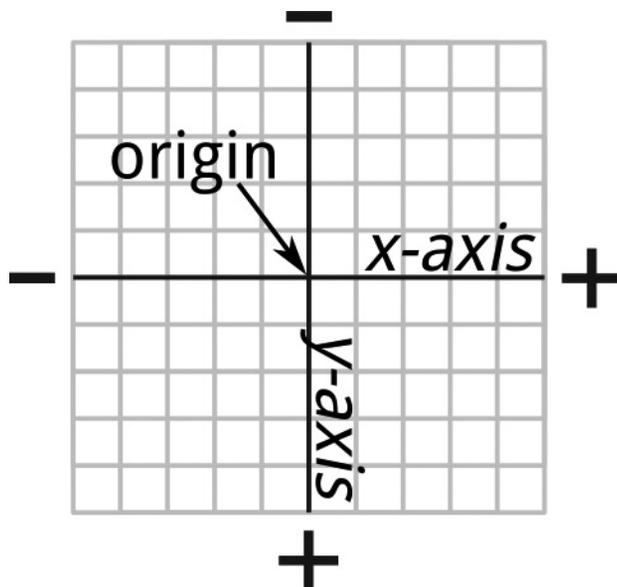


Figure 6-1: The x and y axes and the point of origin

For onscreen items, you measure the lengths of these axes in pixels. In Figure 6-1, you can see the axes and origin are overlaid on a grid. Imagine that each square corresponds to a single pixel. You'll also notice positive (+) and negative (-) labels at either end of each axis; these tell you that the distance away from the origin will be measured either positively or negatively in this direction.

Now that you understand this concept, you can find the coordinates of any point relative to the origin. The *coordinates* are a pair of values—one for each axis—which indicate the distance from the origin. The origin has coordinates (0, 0). For example, given the coordinates (4, 5) you would find the point by moving 4 pixels along the *x-axis*, and 5 pixels along the *y-axis*. Likewise,

the coordinates $(-3, -1)$ indicate a point 3 pixels in a negative direction away from the origin along the x -axis and 1 pixel away from the origin in a negative direction along the y -axis. You can see both of these values plotted on the chart in Figure 6-2.

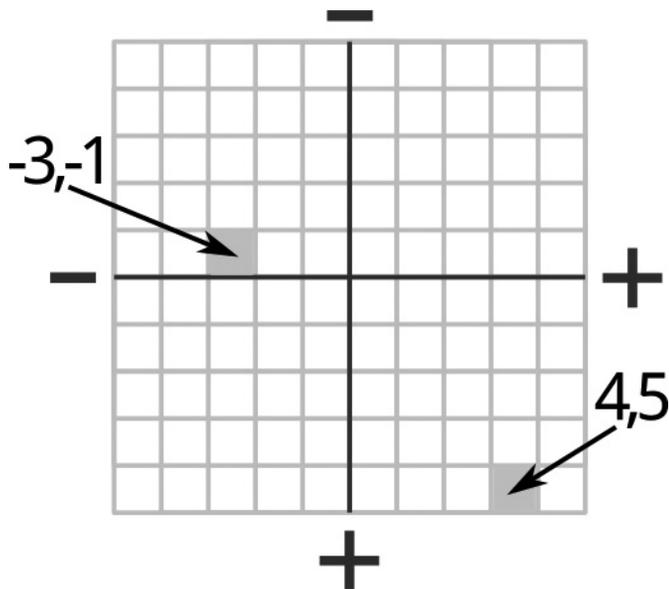


Figure 6-2: Two sets of coordinates

If this all sounds terribly complicated, don't worry—you've been using the Cartesian coordinate system already with properties like `background-position`; you just didn't realize it yet.

In CSS, all elements have a height and a width, each of which is a stated number of pixels in length (even when using other length units such as `em` or a percentage). The height and width together creates a pixel grid; for example, an element that is `10px` by `10px` in size has a pixel grid of `100px`. If you consider that the origin of the element is at the top-left corner, then the two positional values for properties like `background-position` correspond exactly to the x and y coordinates.

NOTE *In CSS, the default origin is the top-left corner of an element, but that isn't always fixed; some CSS properties allow you to change the origin's position. For instance, you could set the origin at the dead center of an element or at the bottom-right corner or anywhere you wish. We'll see this later in this book.*

Applying Dimensional Effects: text-shadow

The ability to apply drop shadows to text using the `text-shadow` property has been around for a long time; Safari first implemented it in version 1.1, which was released in 2005. So you might be wondering why I am discussing it in a book on CSS3. As with the font properties in the previous chapter, `text-shadow`

was dropped from CSS2.1 due to lack of implementation, but this property has been reinstated in the CSS3 spec and recently implemented in Firefox and Opera.

The position of the shadow is set using the *x* and *y* coordinates that I just introduced. The simplest form of the syntax accepts two values: *x* to set the horizontal distance from the text (known as the *x-offset*) and *y* to set the vertical distance (the *y-offset*):

```
E { text-shadow: x y; }
```

By default, the shadow will be the color that it inherited from its parent (usually black), so if you want to specify a different color, you need to provide a value for that, such as:

```
E { text-shadow: x y color; }
```

Here's an example of a gray (hex code #BBB) drop shadow located 3px to the right and 3px down from the original image:

```
h1 { text-shadow: 3px 3px #BBB; }
```

You can see the output of this code in Figure 6-3.

Shadows

Figure 6-3: Simple text-shadow

You don't have to provide positive integers as offset values; you can use both 0 (zero) and negative numbers to get different effects. Here are a few examples:

```
.one { text-shadow: -3px -3px #BBB; }  
.two { text-shadow: -5px 3px #BBB; }  
.three { text-shadow: -5px 0 #BBB; }
```

You can see the output of these examples in Figure 6-4.

The first (top) example uses negative values for both axes, so the shadow is rendered above and to the left of the text. The next (middle) example uses a negative value for the *x*-axis and a positive value for the *y*, so the shadow renders below and to the left. The final (bottom) example has a negative value for the *x* and a value of 0 for *y*, so the shadow renders to the left and on the same baseline.

Quick Brown Fox

Figure 6-4: Different axis offset values for text-shadow

The text-shadow property also has a fourth option: `blur-radius`. This option sets the extent of a blur effect on the shadow and must be used after the offset values:

```
E { text-shadow: x y blur-radius color; }
```

The `blur-radius` value is, like the two offset values, also an integer with a length unit; the higher the value, the wider (and lighter) the blur. If no value is supplied (as in the examples shown in Figure 6-4), the `blur-radius` is assumed to be zero. Here are a couple of examples:

```
.one { text-shadow: 3px 3px 3px #BBB; }  
.two { text-shadow: 0 0 3px #000; }
```

You can see the output of these examples in Figure 6-5.

Black

White

Figure 6-5: Different blur values for text-shadow

In the first example, I set the same offset values as in Figure 6-1, but with a `blur-radius` value of `3px`. The result is a much softer, more “natural” shadow. In the second example, I’ve set `0` values for the offsets and a `3px blur-radius`, matching the text to the background and creating the illusion that the text is raised.

Multiple Shadows

You don’t have to limit yourself to a single shadow—`text-shadow`’s syntax supports adding multiple shadows to a text node. Just supply extra values to the property, using commas to separate them, like so:

```
E { text-shadow: value, value, value; }
```

The shadows will be applied in the order you supply the values. Figure 6-6 shows two examples of multiple shadows in action.



Figure 6-6: Using multiple values with `text-shadow`

The CSS for these examples is shown here. The first example has a class of `one`, and the second has a class of `two`. Note that I’ve indented them for clarity.

```
.one {  
  text-shadow:  
    0 -2px 3px #FFF,  
    0 -4px 3px #AAA,  
    0 -6px 6px #666,  
    0 -8px 9px #000;  
}  
.two {  
  color: #FFF;  
  text-shadow:  
    0 2px rgba(0,0,0,0.4),  
    0 4px rgba(0,0,0,0.4),  
    0 6px rgba(0,0,0,0.4),  
    0 8px 0 rgba(0,0,0,0.4);  
}
```

In the first example, I've kept the *x*-offset at 0 while increasing the *y*-offset's negative value from -2px to -8px. The *blur-radius* increases from 3px to 9px, and the color gets gradually darker, creating a ghostly pale outline behind the characters, which becomes a darker shadow as it gets further from the characters.

In the second example, the *x*-offset also remains consistent, but this time the *y*-offset increases its value positively. Because the *blur-radius* isn't specified, it stays at zero. Here I've used the `rgba()` color function (which I'll explain in Chapter 10), so the color stays the same but is partially transparent, creating an overlapping effect.

Although the value changes are fairly small, the visual difference between the two elements is quite profound.

Letterpress Effect

An effect that's very popular at the moment is the letterpress style. This style gives the illusion that the characters are impressed slightly into the background, as if they'd been stamped into a material (like on a letterpress). This effect is easy to achieve with CSS3.

To create this effect, you need four tones of a color: dark for the characters, medium for the background, and lighter and darker for the shadow. Then you add `text-shadow` with multiple values—a dark (or black) and a light, as in this example:

```
body { background-color: #565656; }
h1 {
  color: #333;
  text-shadow: 0 1px 0 #777, 0 -1px 0 #000;
}
```

The body has a `background-color` value of #565656, which is a fairly medium-dark gray, and the text is a darker tone. The `text-shadow` has two values: black to give a shadow effect and a lighter gray as a highlight; the combination of the two creates the illusion of depth. You can see how this appears in Figure 6-7.



Figure 6-7: A "letterpress" effect using `text-shadow`

Be aware, however, this effect probably isn't very accessible to some users with visual impairments such as colorblindness or partial vision, as the contrast between the text color and background color may not be sufficient to make out the shapes of the characters clearly. You should use an online tool (such as <http://www.checkmycolours.com/>) to test your colors for accessibility, but I'll leave you to make a judgment on that.

Adding Definition to Text: text-outline and text-stroke

As I demonstrated previously, you can stroke the outline of a character using `text-shadow`. Using this method is a bit of a hack, however. The Text Module provides a better way to control outlines: the `text-outline` property. This property accepts three possible values:

```
E { text-outline: width blur-radius color; }
```

For example, here's the CSS to provide text with a 4px blur-radius in blue:

```
h1 { text-outline: 2px 4px blue; }
```

The `text-outline` property is currently not implemented in any browsers, but WebKit browsers offer something similar—and more flexible: the proprietary `text-stroke` property.

Actually, `text-stroke` has four properties in total: two control the appearance of the stroke itself, one is a shorthand property for the previous two, and one controls the fill color of the stroked text. The syntax of those properties is shown here:

```
E {  
  -webkit-text-fill-color: color;  
  -webkit-text-stroke-color: color;  
  -webkit-text-stroke-width: length;  
  -webkit-text-stroke: stroke-width stroke-color;  
}
```

The first property, `text-fill-color`, seems a little unnecessary at first glance, as it performs the same function as the `color` property. Indeed, if you don't specify it, the stroked element will use the *inherited* (or *specified*) value of `color`. This property allows your pages to degrade gracefully, however. For example, you could set `color` to be the same as `background-color` and use the stroke to make it stand out. But if you did this, your text would be hidden in browsers that don't support `text-stroke`. Using `text-fill-color` overcomes this problem.

The other `text-stroke-*` properties are more obviously useful, and their definition is mostly straightforward: `text-stroke-color` sets the color of the stroke, and `text-stroke-width` sets its width (in the same way as `border-width` from CSS2). Finally, `text-stroke` is the shorthand property for `text-stroke-width` and `text-stroke-color`.

Here's a real-world example of `text-stroke` syntax:

```
h1 {  
  color: #555;  
  -webkit-text-fill-color: white;  
  -webkit-text-stroke-color: #555;  
  -webkit-text-stroke-width: 3px;  
}
```

Figure 6-8 shows these properties applied to a text element (and how the text appears in browsers with no support). The first example shows its native, nonstroked state for comparison, while the second example shows some text with `text-stroke` applied.



Figure 6-8: Comparison of text without (top) and with (bottom) `text-stroke` properties applied

I could have achieved the same results with less code by using the `text-stroke` shorthand:

```
h1 {  
  color: #555;  
  -webkit-text-fill-color: white;  
  -webkit-text-stroke: 3px #555;  
}
```

One thing to bear in mind: High width values for `text-stroke-width` can make your text look ugly and illegible, so use this property with caution.

More Text Properties

As mentioned at the start of this chapter, CSS3 also offers a few new text effects that, although less impressive, can make a subtle difference to the legibility and readability of your content. Not all of these features have been implemented yet—indeed, as the spec is so uncertain, some of them may never be—but mentioning them here is worthwhile, so you can see the kind of thinking that’s going into improving text on the Web.

Restricting Overflow

Under certain circumstances—perhaps on mobile devices where screen space is limited—you’ll want to restrict text to a single line and a fixed width; perhaps when displaying a list of links to other pages, where you don’t want the link text to wrap onto multiple lines. In these circumstances, your text being wider than its container and getting clipped mid-character can be quite frustrating.

A new property called `text-overflow` is available in CSS3 for just those circumstances. It has this syntax:

```
E { text-overflow: keyword; }
```

The permitted keyword values are `clip` and `ellipsis`. The default value is `clip`, which acts in the way I just described: Your text is clipped at the point where it flows out of the container element. But the new value that's really interesting is `ellipsis`, which replaces the last whole or partial character before the overflow with an ellipsis character—the one that looks like three dots (`. . .`).

Let's look at an example using the following CSS:

```
p {  
  overflow: hidden;  
  text-overflow: ellipsis;  
  white-space: nowrap;  
}
```

On this `p` element, I've set the `overflow` to `hidden` to prevent the content showing outside of the border, the `white-space` to `nowrap` to prevent the text from wrapping over multiple lines, and a value of `ellipsis` on the `text-overflow` property. You can see the result in Figure 6-9.



Figure 6-9: The `text-overflow` property with a value of `ellipsis`

The last word in the sentence has been truncated and an ellipsis used in place of the removed characters, signifying that the truncation has taken place.

The `text-overflow` property was dropped from the last draft of the Text Module, but has been reinstated in the editor's draft and is expected to be back in the next working draft. This property is already implemented in Internet Explorer and WebKit, and in Opera with the `-o-` prefix.

The Text Module also offers a third value, which is a string of characters to be used instead of the ellipsis, like so:

```
E { text-overflow: 'sometext'; }
```

However, this value remains unimplemented in any browser to date.

Resizing Elements

Although not actually in the Text Module, another new property is useful for elements whose contents are wider than their container. The `resize` property gives users control over an element's dimensions, providing a handle with which a user can drag the element out to a different size.

The property has the following syntax:

```
E { resize: keyword; }
```

The *keyword* values state in which direction the element can be dragged: horizontal or vertical, both, or none. In the following example, I'll show a `p` element with the value of both on the `resize` property, using this code:

```
p {  
  overflow: hidden;  
  resize: both;  
}
```

In Figure 6-10, you can see this in action. The first example shows the element with the `resize` property applied—you can tell by the striped handle in the lower-right corner. The second example uses the same element, but it has been dragged out so you can see a lot more of the text.

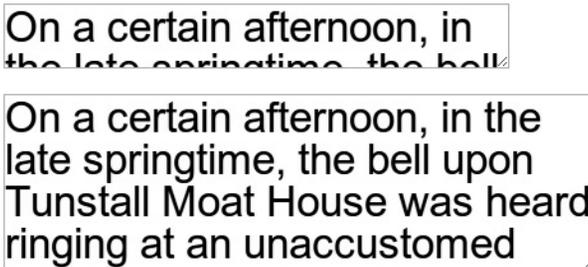


Figure 6-10: A resizable text box shown at default size (top) and expanded (bottom)

All WebKit browsers currently support `resize`, and it's also planned for inclusion in Firefox 4. In supporting browsers it has the value `both` set on `textarea` elements by default.

Aligning Text

The `text-align` property has been around for a long time, but CSS3 adds two new values to it: `start` and `end`. For people who read left-to-right, they are equivalent to the values `left` and `right` (respectively). However, their real usefulness is on internationalized sites that may also use right-to-left text. You can use these new values in Firefox and Safari.

New to CSS3 is the `text-align-last` property, which allows you to set the alignment of the last (or only) line of text in a justified block. This property accepts the same values as `text-align` but is currently implemented in only one browser—Internet Explorer, using the proprietary `-ms-` prefix:

```
E { -ms-text-align-last: keyword; }
```

So if you wanted to justify a block of text but also align the last line to the right, you would use:

```
p {
  text-align: justify;
  -ms-text-align-last: right;
}
```

Wrapping Text

An issue that's frequently encountered when working with dynamic text is line wrapping in inappropriate places. For example, if providing details about an event you would want the start time and end time to appear next to each other on the same line, but with a dynamic line break, the end time may be pushed to the subsequent line. The Text Module aims to provide more control over these kinds of issues with a pair of properties that lets you define more clearly how you want your content to wrap.

word-wrap

The first property is `word-wrap`, which specifies whether the browser can break long words to make them fit into the parent element. The syntax for it is very simple:

```
E { word-wrap: keyword; }
```

This property allows the keyword values `normal` or `break-word`. The former allows lines to break only between words (unless otherwise specified in the markup), and the latter allows a word to be broken if required to prevent overflow of the parent element.

So, for example, if I want to allow long words to be wrapped instead of overflowing their containing element, I might use:

```
p.break { word-wrap: break-word; }
```

Figure 6-11 shows this effect. The left block doesn't use word wrapping, and the right block does.

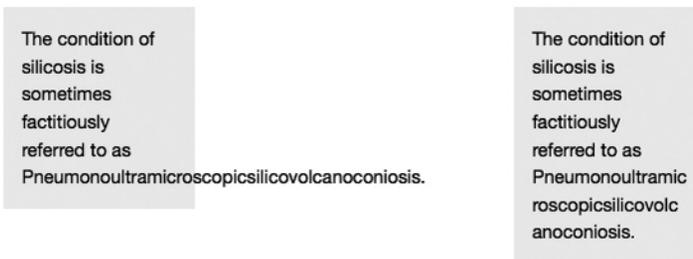


Figure 6-11: Example of text without (left) and with (right) a `break-word` value for `word-wrap`

The `word-wrap` property is widely implemented across all major browsers (yes, including Internet Explorer).

text-wrap

The `text-wrap` property functions in a similar way but sets wrapping preferences on lines of text rather than on single words. Here's the syntax:

```
E { text-wrap: keyword; }
```

It accepts four keyword values: `normal`, `none`, `unrestricted`, and `suppress`. The default is `normal`, which means wrapping will occur at any regular break point, according to the particular browser's layout algorithm, whereas `none` will prevent all wrapping, `suppress` will prevent wrapping unless there is no alternative; if no sibling elements are available with more convenient break points, then breaks are allowed to occur in the same way as the `normal` value. The final value is `unrestricted`, which means the line may break at any point without limitations.

If `text-wrap` is set to either `normal` or `suppress`, you can also apply the `word-wrap` property. For example, if you want no wrapping to occur but want to allow words to be broken if necessary, you'd use this combination:

```
E {  
  text-wrap: suppress;  
  word-wrap: break-word;  
}
```

As of this writing, `text-wrap` remains unimplemented.

Setting Text Rendering Options

Firefox and WebKit browsers support a property called `text-rendering`, which allows developers to control the optimization of speed or legibility. This new feature means the developer can choose how the browser renders the text on a page. Here is the syntax:

```
E { text-rendering: keyword; }
```

The `text-rendering` property has four specific keyword values: `auto`, `optimizeSpeed`, `optimizeLegibility`, and `geometricPrecision`. The default is `auto`, which allows the browser to make rendering choices. `optimizeSpeed` favors speed over legibility, disabling advanced font features for faster rendering, and `optimizeLegibility` will do the opposite at a slight cost of speed. The `geometricPrecision` keyword may have some future value but currently works the same as `optimizeLegibility`.

To get a better idea of what this means in practice, consider the following code example:

```
p.fast { text-rendering: optimizeSpeed; }  
p.legible { text-rendering: optimizeLegibility; }
```

This applies two different optimization effects to identical p elements. Figure 6-12 shows the results. The first example is optimized for speed, and the second example is optimized for legibility.



LYoWAT ff fi fl ffl

LYoWAT ff fi fl ffl

Figure 6-12: Comparison of text optimized for speed (top) and legibility (bottom)

The font family used in this example is DejaVu Serif. You should clearly see the differences between the two text elements, especially with the capital Y/lowercase o pairing at the beginning and the lowercase *ffl* characters at the end. This difference is apparent because the font file contains special instructions for certain combinations of characters, which improve the spacing between individual characters (known as *Kerning*) and join certain sets together (*ligatures*) for increased legibility.

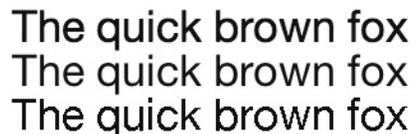
NOTE *Windows, OS X, and Linux all use different text-rendering methods, so you can't depend on this always having an effect on the user's system.*

WebKit browsers offer further control over how text is rendered: the `-webkit-font-smoothing` property. Mac OS X users will know that fonts tend to display smoother and more rounded than they do on Windows. This difference is due to *anti-aliasing*—that is, the way that the screen renders the font so it appears to be smooth rather than pixelated. To set the level of anti-aliasing on your fonts, use the `-webkit-font-smoothing` property:

```
E { -webkit-font-smoothing: keyword; }
```

The keyword value can be `subpixel-antialiased`, `antialiased`, or `none`. The default value, `subpixel-antialiased`, provides the smooth fonts you see in Mac browsers, and `none` shows jagged fonts with no smooth edges. Some people find Mac fonts a little too thick, and nobody likes unaliased fonts, so the `antialiased` value provides the balance between the two.

All three are compared in Figure 6-13. The first line of text has the value of `subpixel-antialiased` and has strong, rounded characters. The second has the `antialiased` value, so the text appears slighter but still very legible. The final line has the value of `none` and appears jagged and quite unattractive.



The quick brown fox
The quick brown fox
The quick brown fox

Figure 6-13: Comparison of `-webkit-font-smoothing` values in Safari (from top): `subpixel-antialiased`, `antialiased`, and `none`

This property is exclusive to WebKit browsers and doesn't appear in any of the CSS3 modules.

Applying Punctuation Properties

Many typographers like to hang punctuation into the margin of a block of text, like in Figure 6-14. Until now this has had to be done using a negative value for the text-indent property, like this:

```
p:first-child { text-indent: -0.333em; }
```

**“No man ever steps in the same
river twice, for it’s not the
same river and he’s not the
same man.”**
- Heraclitus of Ephesus

Figure 6-14: An example of hanging punctuation

This method requires that you have fine control over your text, however, which isn't always the case. The proposed hanging-punctuation property is an attempt to address this issue. Here is the syntax:

```
E { hanging-punctuation: keyword; }
```

The keyword value can be: start, end, or end-edge. These define whether the punctuation can hang outside the start of the first line, the end of the first line, or the end of all lines (respectively). This property is currently unimplemented and not fully described, so it may be dropped from future revisions.

A further typographic convention with punctuation is to trim the spacing of (that is, *kern*) certain marks when they appear at the beginning or end of text blocks, or when certain pairs appear next to each other (like *)]*, for example). The punctuation-trim property was created to this end and has the following syntax:

```
E { punctuation-trim: keyword; }
```

For this property, the allowed keywords are: none, start, end, and adjacent. They describe the position in the text block where the trimming is allowed to occur. This property remains unimplemented as of this writing.

Summary

The last few years have seen a noticeable upturn in the quality of typography on the Web, although the limited range of CSS text properties hasn't made that easy. But I believe that browser makers have noticed the push for better implementation and, slowly but surely, more typographic control is being placed in our hands.

Although the Text Module is under review and some of the new properties in this chapter may never see the light of day, I feel covering them is important so you can be prepared for the eventuality that they are implemented. At the very least, you can be sure that the W3C is aware of web typographers' concerns and are working to alleviate them.

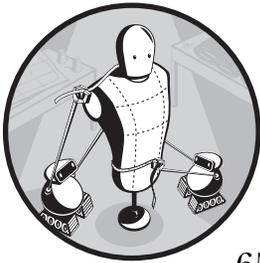
In the previous chapter, I looked at ways to increase the range and variety of fonts, and in this chapter, I've discussed methods to make those fonts more decorative, flexible, and—most importantly—readable. The next chapter will complete the triumvirate of chapters on fonts and typography by introducing a whole new way to lay out text content. Well, new to the Web, that is; printers have been doing it for centuries.

Text Effects: Browser Support

	WebKit	Firefox	Opera	IE
text-shadow	Yes	Yes	Yes	No
text-outline	No	No	No	No
text-stroke	Yes	No	No	No
text-align (new values)	Yes	Yes	No	No
text-align-last	No	No	No	Yes
word-wrap	Yes	Yes	Yes	Yes
text-wrap	No	No	No	No
text-overflow	Yes	No (expected in Firefox 4)	No	Yes
resize	Yes	No (expected in Firefox 4)	No	No
text-rendering	Yes	Yes	No	No
punctuation properties	No	No	No	No

7

MULTIPLE COLUMNS



Although computer screens are getting wider, studies still show that people have difficulty reading long lines of text. (Roughly 65 to 75 characters per line is generally considered a comfortable length to read.) This convention has led to restricted layouts and websites that don't take advantage of the opportunities wider screens present.

For years, magazines and newspapers have used multiple columns to flow content—addressing both the issue of long text lines and how to pack a lot of copy into limited spaces. Now, with the advent of the Multi-column Layout Module in CSS3 (<http://www.w3.org/TR/css3-multicol/>), websites can take advantage of multiple columns, too.

In this chapter, I'll look at the properties that have been implemented, as well as noting some that haven't. And putting your content into columns is not without potential issues, so I'll also address some of those as well.

The Multi-column Layout Module currently has *Candidate Recommendation* status, meaning the module is considered pretty complete, and both Mozilla and WebKit have implemented aspects of it in recent years (albeit with proprietary

properties—more on that later), so you have plenty of opportunities to experiment with multiple columns.

Column Layout Methods

You can divide your content into columns using two methods: either prescriptively by setting a specific number of columns or dynamically by specifying the width of columns and allowing the browser to calculate how many columns will fit into the width of the parent element.

Prescriptive Columns: column-count

The simplest way to divide your content into equally distributed columns is to use the `column-count` property:

```
E { column-count: columns; }
```

The element *E* is the parent of the content you want to divide, and the *columns* value is an integer that sets the number of columns. To split content inside a `div` element into two columns, you would use

```
div { column-count: 2; }
```

Currently, both Firefox and WebKit implement certain multi-column properties, although the Firefox implementation is marginally less standard-compliant and both have slight differences in how they render multi-columns. As these are still somewhat incomplete compared to the specification, both browsers use their proprietary prefixes: `-moz-` for Firefox, and `-webkit-` for WebKit.

In order to make columns work in both browsers, therefore, you need to call both properties in your stylesheet. So when you use `column-count` (or any of the other properties in the rest of this chapter) you have to specify:

```
E {  
  -moz-column-count: columns;  
  -webkit-column-count: columns;  
}
```

For the sake of clarity, in the remaining examples in this chapter, I'll use only the nonprefixed property titles. Bear this in mind, however, when using multiple columns in your production pages.

With that said, let's move on to a real-world example. I'll demonstrate a few paragraphs of copy displayed twice, the first distributed over three columns and the second over four columns. Here's the code I'll use:

```
div[class*='-3'] { column-count: 3; }  
div[class*='-4'] { column-count: 4; }
```

NOTE *I've used the Arbitrary Substring Attribute Value Selector in these examples, which was introduced in Chapter 3.*

You can see the results of this code in Figure 7-1.

A young man—we can sketch his portrait at a dash. Imagine to yourself a Don Quixote of eighteen; a Don Quixote without his corselet, without his coat of mail, without his cuisses; a Don Quixote clothed in a woolen doublet, the blue color of which had faded into a nameless shade between lees of wine and a	heavenly azure; face long and brown; high cheek bones, a sign of sagacity; the maxillary muscles enormously developed, an infallible sign by which a Gascon may always be detected, even without his cap—and our young man wore a cap set off with a sort of feather; the eye open and intelligent; the nose hooked, but finely	chiseled. Too big for a youth, too small for a grown man, an experienced eye might have taken him for a farmer's son upon a journey had it not been for the long sword which, dangling from a leather baldric, hit against the calves of its owner as he walked, and against the rough side of his steed when he was on horseback.	
A young man—we can sketch his portrait at a dash. Imagine to yourself a Don Quixote of eighteen; a Don Quixote without his corselet, without his coat of mail, without his cuisses; a Don Quixote clothed in a woolen doublet, the blue color of which had faded into a	nameless shade between lees of wine and a heavenly azure; face long and brown; high cheek bones, a sign of sagacity; the maxillary muscles enormously developed, an infallible sign by which a Gascon may always be detected, even without his	cap—and our young man wore a cap set off with a sort of feather; the eye open and intelligent; the nose hooked, but finely chiseled. Too big for a youth, too small for a grown man, an experienced eye might have taken him for a farmer's son upon a journey	had it not been for the long sword which, dangling from a leather baldric, hit against the calves of its owner as he walked, and against the rough side of his steed when he was on horseback.

Figure 7-1: Text broken over three and then four columns

Here's the markup (edited for brevity) I used for the example shown in Figure 7-1:

```
<div class="columns-3">
  <p>Lorem ipsum dolor...</p>
</div>
<div class="columns-4">
  <p>Lorem ipsum dolor...</p>
</div>
```

The syntax is extremely simple, and the browser takes care of distributing the content equally.

Dynamic Columns: column-width

The second method for dividing content into columns is perhaps a better choice for flexible layouts. Instead of specifying the number of columns, you use `column-width` to specify the width of each column, and the browser fills the parent element with as many columns as can fit along its width. The syntax is just as easy:

```
E { column-width: length; }
```

As with `column-count`, *E* is the parent element of the content you want to divide into columns. But `column-width` differs in that it requires a *length* value: either a unit of length (such as `px`, or `em`) or a percentage. Here's an example:

```
div { column-width: 100px; }
```

This code would divide the child elements of the div into columns 100px wide and repeat the columns along the width of the div. Let's see how this would work, using the following style rules:

```
.columns {  
    column-width: 100px;  
    width: 950px;  
}
```

Here I have an element with a class name of `columns`, which is 950px wide; the content will be distributed into 100px-width columns. You can see how this renders in Figure 7-2.

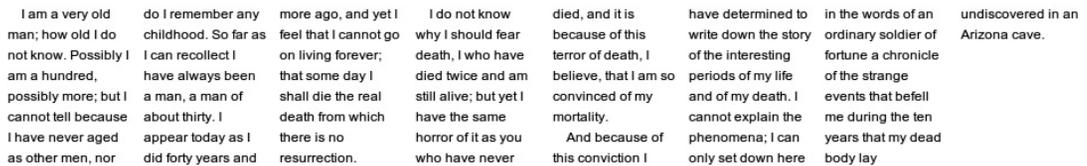


Figure 7-2: Text broken across dynamically created, equally spaced columns

With the `column-width` set to 100px, the browser has created eight columns to fill the parent element. But all is not as it seems. Remember, the parent element is 950px wide, and even with a 10px gap between each column (we'll talk about that shortly), the total width comes to only 870px, so where is the extra white space?

The algorithm that creates the columns is actually quite intelligent and resizes the columns automatically so they better fit the parent. It uses the 100px as a *minimum* value and makes each column slightly wider (in this case, 105px) and the resulting gap between each column wider (in this case, a fraction over 15px) so the total width matches that of its parent.

A Note on Readability

When using the Multi-column Layout Module, be aware that margins and padding will still be applied to your content and can lead to unnecessary blank lines between paragraphs and lines of floating text at the top or bottom of columns, both of which affect the readability of your content.

To illustrate what I mean, Figure 7-3 shows a formatting style that's common on the Web—left-aligned, with a margin between each paragraph—as it appears on text flowing over multiple columns. As you can see, you can easily end up with floating lines of copy, or even individual words, as at the top of the third column.

<p>The Time Traveller (for so it will be convenient to speak of him) was expounding a recondite matter to us. His grey eyes shone and twinkled, and his usually pale face was flushed and animated. The fire burned brightly, and the soft radiance of the incandescent lights in the lilies of silver caught the</p>	<p>bubbles that flashed and passed in our glasses.</p> <p>Our chairs, being his patents, embraced and caressed us rather than submitted to be sat upon, and there was that luxurious after-dinner atmosphere when thought roams gracefully free of the trammels of</p>	<p>precision.</p> <p>And he put it to us in this way — marking the points with a lean forefinger — as we sat and lazily admired his earnestness over this new paradox (as we thought it) and his fecundity.</p>
---	--	---

Figure 7-3: A common type-style in multiple columns

When using multiple columns, it's a good idea to follow the typesetting examples provided by most newspapers and magazines: Indent at the start of each paragraph, and don't leave margins between paragraphs. You can do this easily with existing CSS properties:

```
.columns p {
  margin-bottom: 0;
  text-indent: 1em;
}
```

I applied these properties to the previous example, which you can see in Figure 7-4. This shows the same text, but I've removed the `margin-bottom` from the paragraphs and indented the first sentence of each paragraph.

<p>The Time Traveller (for so it will be convenient to speak of him) was expounding a recondite matter to us. His grey eyes shone and twinkled, and his usually pale face was flushed and animated. The fire burned brightly, and the soft radiance of the incandescent lights</p>	<p>in the lilies of silver caught the bubbles that flashed and passed in our glasses.</p> <p>Our chairs, being his patents, embraced and caressed us rather than submitted to be sat upon, and there was that luxurious after-dinner atmosphere when thought roams</p>	<p>gracefully free of the trammels of precision.</p> <p>And he put it to us in this way — marking the points with a lean forefinger — as we sat and lazily admired his earnestness over this new paradox (as we thought it) and his fecundity.</p>
--	--	--

Figure 7-4: Text in multiple columns—indented and with the margin removed

This example is somewhat easier to read, but the text is ragged on the right edge, which doesn't always look as nice. To combat this, you could justify the text using the `text-align` property:

```
.columns p { text-align: justify; }
```

Justification is only really recommended if you have absolute control over your content, as it places irregular spaces between your words and can actually *decrease* readability unless you hyphenate correctly. (You can add soft hyphens using the HTML code `­`.) Figure 7-5 shows the example text once more, but here I've set the text to justify and inserted soft hyphens into longer words so they break nicely and don't cause too many irregular spaces between words.

The Time Traveller (for so it will be convenient to speak of him) was pounding a recondite matter to us. His grey eyes shone and twinkled, and his usually pale face was flushed and animated. The fire burned brightly, and the soft radiance of the incandescent lights in the lilies of silver caught the bubbles that flashed and passed in our glasses. Our chairs, being embraced and caressed us rather than submitted to be sat upon, and there was that luxurious after-dinner atmosphere when thought roams gracefully free of the trammels of precision. And he put it to us in this way — marking the points with a lean fore finger — as we sat and lazily admired his earnestness over this new paradox (as we thought it) and his fecundity.

Figure 7-5: Text in multiple columns—justified and hyphenated

These values provide, in my opinion, the most readable presentation of text, although using them does require more work from the content author; adding soft hyphens has to be done manually, which takes time, and when new content is added, you have to recheck the formatting to make sure that no errors were introduced.

Different Distribution Methods in Firefox and WebKit

One caveat to using multiple columns is that Firefox and WebKit use slightly different algorithms for calculating columns. As a result, text may be distributed differently among columns in each browser. Firefox’s approach is to make the majority of columns have the same number of lines, with the final column longer or shorter than the rest. In contrast, WebKit will make all of the columns as equal as possible.

Showing this difference is easier than trying to explain it. Figure 7-6 shows text distributed over four columns, as rendered in Firefox.

<p>The intense interest aroused in the public by what was known at the time as "The Styles Case" has now somewhat subsided. Nevertheless, in view of the world-wide notoriety which attended it, I have been asked, both by my friend Poirot and</p>	<p>the family themselves, to write an account of the whole story. This, we trust, will effectually silence the sensational rumours which still persist.</p> <p>I will therefore briefly set down the circumstances which led to my</p>	<p>being connected with the affair. I had been invalided home from the Front; and, after spending some months in a rather depressing Convalescent Home, was given a month's sick leave. Having no near relations or friends, I was trying to</p>	<p>make up my mind what to do, when I ran across John Cavendish. I had seen very little of him for some years. Indeed, I had never known him particularly well.</p>
--	--	--	---

Figure 7-6: Text distributed over four columns in Firefox

The first three columns have the same number of lines, but the fourth column is much shorter; in this example, the majority of the columns have the same number of rows. Now compare this example to the same text rendered in WebKit, shown in Figure 7-7.

<p>The intense interest aroused in the public by what was known at the time as "The Styles Case" has now somewhat subsided. Nevertheless, in view of the world-wide notoriety which attended it, I have been asked, both by my friend Poirot and</p>	<p>the family themselves, to write an account of the whole story. This, we trust, will effectually silence the sensational rumours which still persist.</p> <p>I will therefore briefly set down</p>	<p>the circumstances which led to my being connected with the affair. I had been invalided home from the Front; and, after spending some months in a rather depressing Convalescent Home, was given a month's sick leave. Having no near</p>	<p>relations or friends, I was trying to make up my mind what to do, when I ran across John Cavendish. I had seen very little of him for some years. Indeed, I had never known him particularly well.</p>
--	--	--	---

Figure 7-7: Text distributed over four columns in WebKit

Here you see that the first and third columns have one line of text more than the other two; rather than rendering one column much shorter than the others as Firefox does, WebKit renders the columns more equally overall, thus distributing any surplus more evenly among columns.

This came about because previous versions of the specification (from 2001 and 2005) never made clear how the copy should be balanced over the columns, so both browser makers came up with their own solutions. The W3C addressed this in later versions of the specification with the `column-fill` property:

```
E { column-fill: keyword; }
```

The keyword values are *auto*, which fills columns sequentially, as Firefox does, and *balance*, which distributes in the same way as WebKit does. The default is *balance*, so we can say that the WebKit approach is correct in this case. However, neither of the browsers actually implements this property yet, so until they do you'll just have to be aware of the different approaches when working with columns.

Combining column-count and column-width

You can set both `column-count` and `column-width` properties on an element, though, at first, you might think this would create a conflict. This possibility has been taken into consideration, however: If both properties are applied to the same element, the `column-count` value acts as a maximum. To illustrate, let's refer to Figure 7-5, but change the CSS to also include the `column-count` property:

```
.columns {  
  column-count: 5;  
  column-width: 100px;  
}
```

So the logic behind this would be: divide the text into columns of 100px each, unless that would create five or more columns, in which case make five with a minimum width of 100px.

If you refer back to our example you'll remember that, given the parent element's width of 950px, the `column-width` property rendered eight columns. As you're applying both properties on the same element, however, the `column-count` property takes precedence and only five columns are distributed, with their widths dynamically altered to best fit the parent element.

WebKit implements this rule incorrectly, however, and instead creates five columns of 100px each, leaving empty white space, as shown in Figure 7-8.

The first example shows Firefox's correct interpretation of the spec; the second example shows WebKit's incorrect interpretation. As these properties use proprietary prefixes, however, there is an opportunity for this inconsistency to be corrected before the final implementation of the nonprefixed properties.

Buck did not read the newspapers, or he would have known that trouble was brewing, not alone for himself, but for every tide-water dog, strong of muscle and with warm, long hair, from Puget Sound to San Diego.	Because men, groping in the Arctic darkness, had found a yellow metal, and because steamship and transportation companies were booming the find, thousands of men were rushing into the Northland.	These men wanted dogs, and the dogs they wanted were heavy dogs, with strong muscles by which to toil, and furry coats to protect them from the frost.	Judge Miller's place, it was called. It stood back from the road, half hidden among the trees, through which glimpses could be caught of the wide cool veranda that ran around its four sides. The house was	approached by gravelled driveways which wound about through wide-spreading lawns and under the interlacing boughs of tall poplars.
Buck did not read the newspapers, or he would have known that trouble was brewing, not alone for himself, but for every tide-water dog, strong of muscle and with warm, long hair, from Puget Sound to San	Diego. Because men, groping in the Arctic darkness, had found a yellow metal, and because steamship and transportation companies were booming the find, thousands of men were rushing into the	Northland. These men wanted dogs, and the dogs they wanted were heavy dogs, with strong muscles by which to toil, and furry coats to protect them from the frost. Buck lived at a big	house in the sun-kissed Santa Clara Valley. Judge Miller's place, it was called. It stood back from the road, half hidden among the trees, through which glimpses could be caught of the wide	cool veranda that ran around its four sides. The house was approached by gravelled driveways which wound about through wide-spreading lawns and under the interlacing boughs of tall poplars.

Figure 7-8: How Firefox (top) and WebKit (bottom) interpret combined column properties

If you want to use these two properties together, a shorthand property is available:

```
E { columns: column-width column-count; }
```

This property is currently only implemented in WebKit browsers (again, with its proprietary prefix), so if you use the values from Figure 7-8, the required code would be:

```
div { -webkit-columns: 100px 5; }
```

Column Gaps and Rules

When using a prescriptive multi-column layout, the browser should place a default 1em gap between each column. (Remember that when using dynamic layouts, as in Figure 7-5, 1em is the *minimum* width.) You can, however, alter that default and specify your own distances by using two new properties: `column-gap` and `column-rule`.

The first property, `column-gap`, sets the space between columns, and its syntax is very simple:

```
E { column-gap: length; }
```

The `length` value is any number with a standard CSS length unit. Here's an example that would put a 2em gap between each of your generated columns (though not on the outside of either the first or last column):

```
div { column-gap: 2em; }
```

The second property, `column-rule`, draws a line, similar to a border, equidistantly between columns. The syntax for `column-rule` is actually shorthand for three subproperties: `column-rule-width`, `column-rule-style`, and `column-rule-color`. These subproperties take values in exactly the same way as their equivalent `border-*` properties from CSS2. Here's the syntax:

```
E {
  column-rule-width: length;
  column-rule-style: border-style;
  column-rule-color: color;
  column-rule: length border-style color; /* Shorthand of previous 3 */
}
```

If you add real values, you would have something like this:

```
div {
  column-rule-color: red;
  column-rule-style: solid;
  column-rule-width: 1px;
}
```

You can then use the `column-rule` shorthand to set all three subproperties at once, like this:

```
div { column-rule: 1px solid red; }
```

Now that I've demonstrated the syntax of both new properties, let's look at an example demonstrating the `column-gap` and the `column-rule` shorthand:

```
.columns {
  column-count: 4;
  column-gap: 2em;
  column-rule: 0.3em double silver;
}
```

This example splits the element's children into four columns, each with a gap of 2em and a 0.3em border between the columns. The gap is distributed equally on either side of the border; this example uses a 1em gap, a 0.3em border, and then another 1em gap. The width of the rule is added to the width of the gap, so the total distance between columns is 2.3em. You can see the result of this example in Figure 7-9.

The intense interest aroused in the public by what was known at the time as "The Styles Case" has now somewhat subsided. Nevertheless, in view of the world-wide notoriety which attended it, I have been asked, both by my friend Poirot and the family themselves, to

write an account of the whole story. This, we trust, will effectually silence the sensational rumours which still persist. I will therefore briefly set down the circumstances which led to my being connected with the affair. I had been invalidated home from the

Front; and, after spending some months in a rather depressing Convalescent Home, was given a month's sick leave. Having no near relations or friends, I was trying to make up my mind what to do, when I ran across John Cavendish. I had seen very little of him for some years.

Indeed, I had never known him particularly well. He was a good fifteen years my senior, for one thing, though he hardly looked his forty-five years. As a boy, though, I had often stayed at Styles, his mother's place in Essex.

Figure 7-9: Intercolumn gaps and rules

Containing Elements within Columns

So far in these examples I've only used blocks of text, which flow neatly into columns. But what happens with larger elements, like images, that are more than a column wide? Firefox and WebKit treat them differently, with the latter conforming to the specification.

Let's look at the different resolutions in each browser, by adding an image inside a column layout, using the `img` element. First, take a look at the result in Firefox, which is displayed in Figure 7-10.

The intense interest aroused in the public by what was known at the time as "The Styles Case" has now somewhat subsided. Nevertheless, in view of the world-wide notoriety which attended it, I have been asked, both by my friend Poirot and the family themselves, to write an account of the whole story. This, we trust, will effectually silence the sensational rumours which still persist.

I will therefore briefly set down the

circumstances which led to my being connected with the affair.



I had been invalidated home from

the Front; and, after spending some months in a rather depressing Convalescent Home, was given a month's sick leave. Having no near relations or friends, I was trying to make up my mind what to do, when I ran across John Cavendish. I had seen very little of him for some years. Indeed, I had never known him particularly well. He was a good fifteen years my senior, for one thing, though he hardly looked his forty-five years.

Figure 7-10: An `img` element wider than a column in Firefox

As you can see, the image displays in the second column, and flows across into the third, where it's visible behind the text. Now take a look at the same content displayed in WebKit, in Figure 7-11.

The intense interest aroused in the public by what was known at the time as "The Styles Case" has now somewhat subsided. Nevertheless, in view of the world-wide notoriety which attended it, I have been asked, both by my friend Poirot and the family themselves, to write an account of the whole story. This, we trust, will effectually silence the sensational rumours which still persist.

I will therefore briefly set down

the circumstances which led to my being connected with the affair.



I had been invalidated home from

the Front; and, after spending some months in a rather depressing Convalescent Home, was given a month's sick leave. Having no near relations or friends, I was trying to make up my mind what to do, when I ran across John Cavendish. I had seen very little of him for some years. Indeed, I had never known him particularly well. He was a good fifteen years my senior, for one thing, though he hardly looked his forty-five years.

Figure 7-11: An `img` element wider than a column in WebKit

The image is displayed in the same position as in the previous example, but now the overflow is hidden; it is clipped at a point halfway inside the column-gap. This implementation is correct, as this quote from the specification makes clear:

Content in the normal flow that extends into column gaps (e.g., long words or images) is clipped in the middle of the column gap.

But what happens if your content spans multiple columns—if, for example, you want this image displayed across the second and third columns? And what if you have a heading at the end of a column, which you don't want to have split onto the next? The good news is that the authors of the Multi-column Layout Module have anticipated these possibilities and provided properties to deal with them. The bad news is that, as of this writing, no browsers have implemented them.

Elements Spanning Multiple Columns

To deal with elements that need to be displayed in full across more than one column, the module introduces the `column-span` element. The syntax is:

```
E { column-span: value; }
```

Here *value* can be only one of two possibilities: `1` or `all`. The default is `1`, which will keep the element in the column flow. The alternative value, `all`, will provide a break in the flow—all content before the element (*E*) will be distributed into columns, and all content after the element will be distributed into columns, but the element itself will not.

Elements Breaking over Multiple Columns

So far we've dealt mostly with text, which flows naturally over multiple columns. But what happens with other elements, such as subheadings or lists, which shouldn't be split between columns? The point at which a column ends and the next one begins is known as a *break*, and you can override the automatic generation of breaks with the `break-after`, `break-before`, and `break-inside` properties:

```
E { break-after: keyword; }  
E { break-before: keyword; }  
E { break-inside: keyword; }
```

Each property accepts a range of keyword values: all three accept `auto` and `avoid`, and the `break-after` and `break-before` properties allow an extra value, `column`. The `avoid` value will ensure that no break occurs either immediately after or immediately before (whichever property applies) the specified element, and the `column` value behaves in the opposite way, forcing a break after or before the element. The default value for all of the properties is `auto`, which neither forces nor forbids a break, letting the browser make the decision about whether a break should happen after, before, or inside the element.

The properties are applied to the element that you want to avoid breaking over more than one column, and they instruct the browser where the column break should be applied. Here are two examples:

```
h2 { break-after: column; }  
img { break-before: avoid; }
```

In the first example, a column break will be forced immediately after the `h2` element, regardless of whether a break would occur there naturally. In the second, a break can never occur before the `img` element, so in a situation where that may occur, the `img` will be positioned at the bottom of a column with the break after the `img`.

Summary

Although CSS3 makes flowing your content into columns very easy, the challenges it presents are not so much technical as they are practical: What happens if you want to use images that are wider than a column or if you want to use long headlines in narrow columns?

Although using multiple columns is definitely appropriate in certain situations, think twice about whether your content is suitable. Make sure you have full control over your content before using these properties, and don't design websites that rely on multiple columns, if your client doesn't have a dedicated and conscientious web content team.

Also, bear in mind that screen resolutions can vary dramatically, and content that looks readable to you may not be readable to your website's visitors. If they have to scroll up and down too frequently, which can cause a lot of confusion, they may be put off from visiting your site altogether. But with all that said, clever use of columns can make your content much more readable.

In the previous three chapters, I've described how CSS3 provides methods to format and present your text copy, allowing you better control over your typography and layout. Next I'm going to introduce ways you can improve the overall visual presentation of your websites, starting with new background and border effects.

Multiple Columns: Browser Support

	WebKit	Firefox	Opera	IE
<code>column-count</code>	Yes (with prefix)	Yes (with prefix)	No	No
<code>column-width</code>	Yes (with prefix)	Yes (with prefix)	No	No
<code>column-gap</code>	Yes (with prefix)	Yes (with prefix)	No	No
<code>column-rule</code>	Yes (with prefix)	Yes (with prefix)	No	No
<code>columns</code>	Yes (with prefix)	No	No	No
<code>column-span</code>	No	No	No	No
<code>break-*</code>	No	No	No	No

8

BACKGROUND IMAGES AND OTHER DECORATIVE PROPERTIES



Adding decorative elements to make our websites more visually appealing is surprisingly resource- and time-intensive. Even simple graphical effects can require a lot of unnecessary markup. You are limited to only one background image per element, so to achieve a desired effect, which often involves duplicating images at different sizes, you commonly have to use as many as three extraneous elements. This leads to obfuscated markup, longer loading times, and more strain on the server to load those extra graphical elements.

CSS3 introduces a number of new and extended properties aimed at overcoming this issue, and the browser makers have been quick to implement them and to add a number of their own implementations as well. Over the next few chapters, we'll take a look at the new range of features that we can use to prettify our pages.

Background Images

I'll begin by taking you on a walk through one of the specifications that the W3C is making a priority, the Backgrounds and Borders Module (<http://www.w3.org/TR/css3-background/>). Due to high demand from web developers, many of its new properties have already been implemented by browsers, so plenty of real-world test results are available to learn from.

Unlike previous versions of CSS, CSS3 allows multiple images to be applied to elements and those images can be resized on the fly. Just these two new features alone would be enough to please most of us, but the specification goes further to provide more control over image placement and image tiling.

The Backgrounds and Borders Module currently has Candidate Recommendation status and is likely to become a Recommendation in the near future. Although, as mentioned previously, many of its features are ready for use, some properties still require proprietary browser prefixes.

Multiple Background Images

The first new feature in the Backgrounds and Borders Module isn't a new property, but an extension of an existing one—or, rather, several existing ones. Almost all of the `background-*` properties now accept multiple values, so you can add many background images to the same element (`background-color` is the exception to this).

To add multiple background layers to an element, just list the values separated by commas. For example, here's the syntax with `background-image`:

```
E { background-image: value, value; }
```

For each background layer you create, you can add appropriate values to all of the relevant `background-*` properties. Here's a real-world example:

```
h2 {  
  background-image: url('bunny.png'), url('landscape.jpg');  
  background-position: 95% 85%, 50% 50%;  
  background-repeat: no-repeat;  
}
```

You can see the output in Figure 8-1. The layers are created in reverse order—that is, the first layer in the list becomes the topmost layer, and so on. In my example code `bunny.png` is a layer above `landscape.jpg`. The `background-position` property follows the same order: The landscape is positioned at 50% left and 50% top (the horizontal and vertical center) of its containing element and the bunny at 95% left and 85% top.

Rabbit in a landscape



Figure 8-1: Two background images on the same element¹

Notice that `background-repeat` has only one value. If a property has fewer values than there are background layers, the values will repeat. In this example that means `no-repeat` will be applied to all background layers.

You can use multiple values with the `background` shorthand property; as with the individual properties, you just need to provide a comma-separated list. To get the result seen in Figure 8-1, I could have also used this code:

```
h2 {  
  background:  
    url('bunny.png') no-repeat 95% 85%,  
    url('landscape.jpg') no-repeat 50% 50%;  
}
```

As I mentioned at the start of this section, `background-color` is the only `background-*` element that doesn't accept multiple values; the color layer will always be stacked below the image layers. If you want to specify a background color when using the shorthand property, you must place it in the last instance of the comma-separated list. In the example code that would be in the instance with the landscape picture value:

```
h2 {  
  background:  
    url('bunny.png') no-repeat 95% 85%,  
    #000 url('landscape.jpg') no-repeat 50% 50%;  
}
```

1. The bunny image is by Flickr user Andrew Mason (http://www.flickr.com/photos/a_mason/42744470/), and the landscape image is by Flickr user Nicholas_T (http://www.flickr.com/photos/nicholas_t/1426623052/). Both images are published under a Creative Commons Attribution license.

Multiple backgrounds are supported in Firefox (version 3.6 and up), WebKit, and Opera, and are scheduled for inclusion in IE9.

Keep in mind, however, browsers that don't support the new multiple syntax—e.g., older versions of Internet Explorer or Firefox 3.5 and below—will ignore it and refer to the previous rule in the cascade. You should declare a property on the element *before* the multiple values as a fallback. For example:

```
h2 {  
  background: url('landscape.jpg') no-repeat 50% 50%;  
  background:  
  url('bunny.png') no-repeat 95% 85%,  
  url('landscape.jpg') no-repeat 50% 50%;  
}
```

In this case, the nonsupporting browser will ignore the property with multiple values and use the single value that precedes it. Note, however, that in Internet Explorer, you will have to use the shorthand background property for both; if you use individual background-* properties, they will take preference over the shorthand property and no image will be displayed.

Background Size

A new property to CSS3 is background-size. This property, as you can probably guess, allows you to scale your background images. Here's the syntax:

```
E { background-size: value; }
```

Firefox, WebKit, and Opera have implemented this property already—in previous versions of Firefox (before 4) and Safari (before 5), you'll have to use the browser's proprietary prefixes (-moz- and -webkit-, respectively). No prefix is required in later versions—and it is scheduled for inclusion in IE9 (also with no prefix). As always, for the sake of clarity, I'll leave out the browser-specific prefixes in the examples in the rest of this chapter, but remember to use them in your own code.

This property's value can be a pair of lengths or percentages, a single length or percentage, or a keyword. If a pair is used, the syntax is:

```
E { background-size: width height; }
```

So to resize a background image to be 100px wide and 200px high, you would use:

```
div { background-size: 100px 200px; }
```

The length can be any standard unit of measurement. If you use percentages, the dimension is based on the containing element, *not* the background image. So a width and height of 100%, for example, will make the background image fill the container. To make the image appear at its natural size, use the auto keyword.

If only a single value is specified, that value will be considered the width, and the height will be assigned the default value of `auto`. Therefore, these two examples are exactly equivalent:

```
div { background-size: 100px auto; }  
div { background-size: 100px; }
```

You can use your newly learned multiple background method with `background-size` as well. For example, let's revisit Figure 8-1, but repeat the bunny image a few more times, adding different values to the `background-position` and `background-size` properties. Here's the code:

```
h2 {  
  background:  
    url('bunny.png') no-repeat 95% 85%,  
    url('bunny.png') no-repeat 70% 70%,  
    url('bunny.png') no-repeat 10% 100%,  
    url('landscape.jpg') no-repeat 50% 50%;  
  background-size: auto,5%,50%,auto;  
}
```

Figure 8-2 shows this method in action.

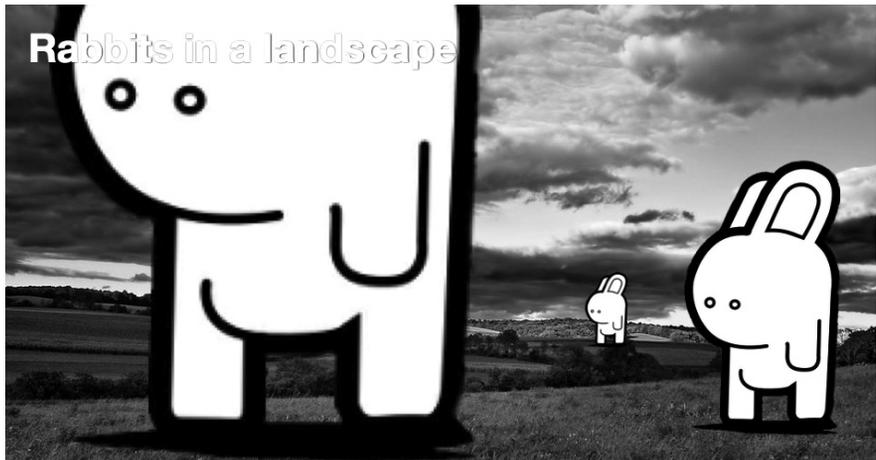


Figure 8-2: Example of resized background images

The spec states that the `background-size` property can be used in the background shorthand property, as long as it occurs *after* `background-position`. However, this hasn't been implemented in any browser yet, so for now, you must call the property separately.

Two other keywords are available: `contain` and `cover`. The `contain` keyword sets the image should to scale (proportionately) as large as possible, without exceeding either the height or width of the containing element; `cover` sets the image to scale to the size of either the height or width of the containing element, whichever is larger.

Take a look at the following code to see what I mean:

```
div[class^='bunny'] {
    background-image: url('bunny.png');
    background-position: 50% 50%;
}
div[class$='-1'] { background-size: contain; }
div[class$='-2'] { background-size: cover; }
```

I used two `div` elements, with a class of `bunny-1` and `bunny-2`, and set different keyword values for `background-size` on each. The result is shown in Figure 8-3.

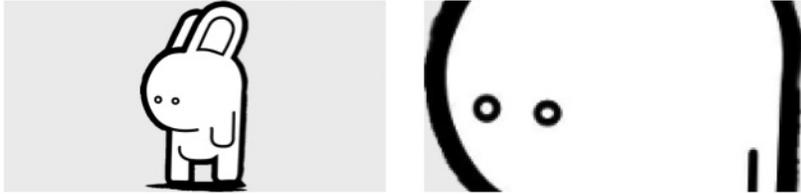


Figure 8-3: `background-size` keywords: `contain` (left) and `cover` (right)

The box on the left has the `contain` keyword value, so the background image fills the box vertically (the shortest length); the box on the right has the `cover` keyword value, so the background image fills the box horizontally (the longest length).

Background Clip and Origin

In CSS2, the position of a background image is defined relative to the outer limit of its containing element's padding, and any overflow will extend underneath its border. CSS3 introduces two new properties that provide more granular control over this placement.

The first property is `background-clip`, which sets the section of the box model that becomes the limit of where the background (either color or image) is displayed. Here's the syntax:

```
E { background-clip: keyword; }
```

According to the spec, the keyword can be one of three possible values: `border-box`, `content-box`, or `padding-box`. The `border-box` value is the default, and it means the background will be displayed behind the border (you can see it if you use a transparent or semi-opaque border color). A value of `padding-box` means the background will display only up to, and not behind, the border. `content-box` means the background will stop at the element's padding.

I'll illustrate the difference using the following code:

```
h2[class^='clip'] {
    background: url('landscape.jpg') no-repeat 50% 50% #EFEFEF;
    border-width: 20px;
    padding: 20px;
}
```

```
h2[class$='-brdr'] { background-clip: border-box; }
h2[class$='-pddng'] { background-clip: padding-box; }
h2[class$='-cntnt'] { background-clip: content-box; }
```

I used three h2 elements with classes of `clip-brdr`, `clip-pddng`, and `clip-cntnt`, respectively. Figure 8-4 illustrates the difference between the values.



Figure 8-4: Showing the effect of different values on the `background-clip` property: `border-box` (left), `padding-box` (center), and `content-box` (right)

Here, I've used a semi-opaque border; you can see the image paint beneath it in the box on the left, which has the `border-box` value. The central box has the `padding-box` value, and as you can see, the background stops at the limit of the padding. In the box on the right, the value is `content-box`, so the background does not show behind the padding.

This property is implemented correctly in Opera and should be in both Firefox 4 and IE9. Older versions of Firefox have a nonstandard implementation where the values `border` and `padding` are used in place of `border-box` and `padding-box`, respectively, with the proprietary `-moz-` prefix on the property.

With WebKit it gets more complicated still; the oldest implementations have `border` and `padding` values, like Firefox, and also the `content` keyword in place of `content-box`, all with the `-webkit-` prefix on the property. More recent versions use `border-box`, `padding-box`, and `content-box`, still with the prefixed property. The most recent versions (since Safari 5, for example) have dropped the prefix requirement when using `border-box` and `padding-box`, but not `content-box`. That being the case, if you want the example code to work in all browser versions, you'd have to adapt it like so:

```
h2[class$='-brdr'] {
  -moz-background-clip: border;
  -webkit-background-clip: border;
  -webkit-background-clip: border-box;
  background-clip: border-box;
}
h2[class$='-pddng'] {
  -moz-background-clip: padding;
  -webkit-background-clip: padding;
  -webkit-background-clip: padding-box;
  background-clip: padding-box;
}
```

```
h2[class$='-cntnt'] {
  -webkit-background-clip: content;
  -webkit-background-clip: content-box;
  background-clip: content-box;
}
```

WebKit browsers allow a further, nonstandard keyword value: `text`. When used on text with a transparent background color this fills the text with the background image. To make this work compatibly with non-WebKit browsers, we should use the `-webkit-text-fill-color` that was introduced back in Chapter 6. Here's an example:

```
h2 {
  background: url('landscape.jpg') no-repeat 50% 60%;
  -webkit-background-clip: text;
  -webkit-text-fill-color: transparent;
}
```

The result is shown in Figure 8-5, where the background image is clipped to the text.

The second property that gives you more granular control is `background-origin`. Using `background-origin`, you can set the point where the background is calculated to begin. As I mentioned before, CSS2 background positions are calculated relative to the limit of the padding, but `background-origin` lets you change that. Here's the syntax:



Figure 8-5: Applying a background image to text with the `-webkit-background-clip` property

```
E { background-origin: keyword; }
```

The `background-origin` property accepts the same *keyword* values as we've just seen in `background-clip: border-box, content-box, and padding-box`. Once more, I'll call on my bunny to help illustrate the difference. The next demonstration uses this code:

```
h2[class^='origin'] { background: url('bunny.png') no-repeat 0 100%;}
h2[class$='-brdr'] { background-origin: border-box; }
h2[class$='-cntnt'] { background-origin: content-box; }
h2[class$='-pddng'] { background-origin: padding-box; }
```

The effects of the different values are illustrated in Figure 8-6. As you can see, the bunny is in a different position in each box because the background-position is calculated relative to a different point in each box.

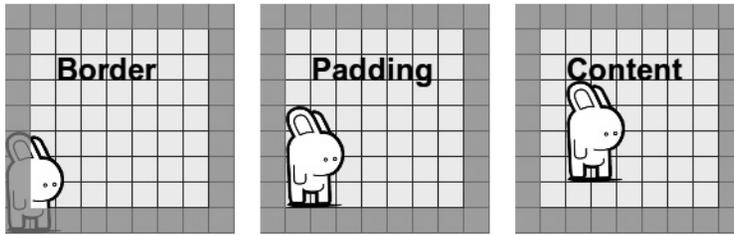


Figure 8-6: The background-origin property with values of border-box (left), padding-box (center), and content-box (right)

The background-position is always set at 0 100%, which is the bottom left. The point where the bottom left is measured from changes depending on the background-origin value, however. In the first box, the background originates at the limit of the border; in the second, from the limit of the padding; and in the third, from the limit of the content box.

A couple of things to bear in mind: First, this property has no effect if the background-position is set to fixed. Second, both background-clip and background-origin accept multiple values, using the same syntax used in “Multiple Background Images” on page 94.

As with background-clip, Opera and the latest WebKit browsers have an implementation that is the same as the spec, and Firefox 4 and IE9 should have likewise. Older versions of Firefox and much older versions of WebKit have nonstandard versions of this property and have implemented the three possible values as border, content, and padding using the prefixed property, while older but pre-Safari 5 versions of WebKit require the correct values, also with the prefixed property. Therefore, to support every browser version the previous code example would have to be extended like so:

```

h2[class$='-brdr'] {
  -moz-background-origin: border;
  -webkit-background-origin: border;
  -webkit-background-origin: border-box;
  background-origin: border-box;
}
h2[class$='-pddng'] {
  -moz-background-origin: padding;
  -webkit-background-origin: padding;
  -webkit-background-origin: padding-box;
  background-origin: padding-box;
}
h2[class$='-cntnt'] {
  -moz-background-origin: content;
  -webkit-background-origin: content;
  -webkit-background-origin: content-box;
  background-origin: content-box;
}

```

background-repeat

The background-repeat property currently has four possible values: no-repeat, repeat, repeat-x, and repeat-y. These values allow you to tile images across either axis of an element (or both axes) but don't allow for any finer control than that. CSS3 extends the range of tiling options with two new values.

The first is space, which will set the background image to repeat across its containing element as many times as possible without clipping the image. All of the repetitions (except the first and last) will then be equally spaced, so the image is evenly distributed.

The second is round, which will likewise set the background image to repeat as many times as possible without clipping, but instead of equally spacing the repetitions, the images will scale so a whole number of images fills the containing element.

Although these new values are once again planned for inclusion in IE9, the only browser to implement them to date is Opera (version 10.5+), so I'll provide an example using that browser:

```
.bunny {  
  background-image: url('bunny_sprite.png');  
  background-repeat: repeat;  
  background-size: 20%;  
}  
.space { background-repeat: space; }  
.round { background-repeat: round; }
```

First, I've resized *bunny_sprite.png* with the background-size property and then applied it to three different boxes, each with a different background-repeat value. Figure 8-7 displays the results.

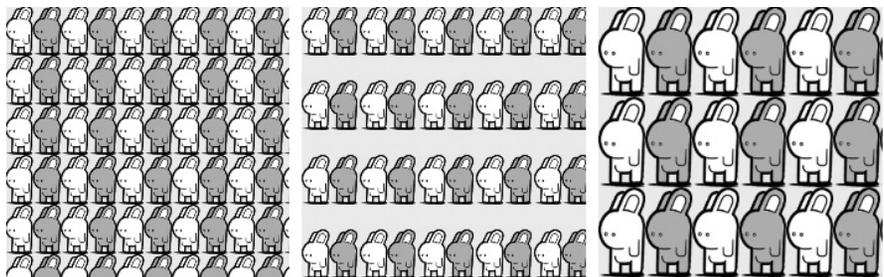


Figure 8-7: background-repeat values: repeat (left), space (center), and round (right)

You can see some slight rounding errors, but nothing major. The box on the left has a background-repeat value of repeat and shows the behavior you would currently expect. The box in the middle has a value of space, and the maximum number of images that can be tiled without clipping or scaling are displayed with empty space between them. Finally, the box on the right has a value of round, which calculates the maximum whole number that can fit in the containing element both horizontally and vertically.

WebKit recognizes these keywords, but treats them both incorrectly as no-repeat. Firefox ignores them and uses the previous cascaded or inherited value.

Background Image Clipping

A technique that's become a part of every web developer's arsenal in the last few years has been the use of *image sprites*. An image sprite is when you have many different images in a single file, but you show only a part of the image on any given element (if you're not familiar with CSS sprites, you can find an explanation at <http://www.alistapart.com/articles/sprites/>). This technique is borrowed from old video games and is useful for loading fewer images, which means fewer hits to the server.

This technique is not without its drawbacks, though: It relies on the element being able to mask the parts of the image that you don't want to be shown, and it can make it little harder to maintain your graphics.

Mozilla has introduced a new proprietary property in Firefox 4 (Beta), which aims to get around those issues. Called `image-rect`, the syntax looks like this:

```
E { background-image: -moz-image-rect(url, top, right, bottom, left); }
```

The first value is the path to the image you want to clip, and the next four values define the four sides of the rectangle you want to show, using the same format as the CSS2 `clip` property: in order, top, right, bottom, and left, measured from the top-left corner.

Here's an example to illustrate exactly what I mean. I'm going to use a single background sprite and then apply various values to `image-rect` to display different areas of the image. Here's the code:

```
.bunny { background: url('bunny_sprite.png') no-repeat 50% 50% #EFEFEF; }  
.white { background-image: -moz-image-rect(url('bunny_sprite.png'),0,57,100,0); }  
.gray { background-image: -moz-image-rect(url('bunny_sprite.png'),0,114,100,57); }
```

Figure 8-8 shows the results of this technique. Three boxes are shown in the figure; each box uses the same background image (shown in full in the first box).



Figure 8-8: Image sprite (left) cropped using `-moz-image-rect` (center and right)

All the boxes have a class of `bunny`, so browsers that don't support `image-rect` will revert to that background image. The image `bunny_sprite.png` is 114px wide and 100px high.

The second box has a class of `white` and only shows the white bunny. The `image-rect` property has the image URL followed by the dimensions of the clip box: the top is 0px from the top; the right is 57px from the left; the bottom is 100px from the top; and the left is 0px from the left. This box shows only the left-hand side of the image—that is, the white bunny.

You can see the box “drawn” by those clip values in Figure 8-9.

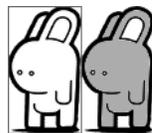


Figure 8-9: The bounding box drawn by the `-moz-image-rect` property

The third box has a class of `gray`, and only shows—guess what?—the gray bunny. Again the `image-rect` property has the

same URL value, but this time the dimensions are different: The top is 0px from the top, the right is 114px from the left, the bottom is 100px from the top, and the left is 57px from the left.

The W3C are considering providing a property to substitute the CSS sprites technique in a future revision of CSS3 (you can see the proposed syntax in Chapter 17), but as far as I’m aware, no final decision has yet been made on which syntax it will use.

Image Masks

In the last few years, the WebKit team has created a series of new properties that add “eye candy” to CSS. Known loosely as *CSS Effects*, some of these properties have been adopted by the W3C and drafted for inclusion in CSS3. I’ll introduce those later in Chapters 12–14.

One of the properties that hasn’t yet been adopted is the CSS Image Mask, and although other browser makers are not currently showing an interest in implementing this property, all WebKit browsers support Image Masks.

Anyone familiar with graphics should be familiar with the concept of *masking*. Essentially, masking places an image layer over a background, making the area that is not transparent, transparent—in other words, the background will show through in the shape of the image layer. Take a look at the example in Figure 8-10.



Figure 8-10: From left to right, a background, an image mask, and the mask applied to the background

Figure 8-10 is an image mask in three parts: On the left is the background; in the center, the mask; on the right, I applied the mask to the background—the black area becomes transparent, allowing only that area of the background to show through.

WebKit introduces a new set of properties that allows you to do this to any element. The syntax (which has the `-webkit-` prefix) is the same as the `background-*` set of properties and includes `-webkit-mask-image`, `-webkit-mask-position`, `-webkit-mask-repeat`, and so on.

As an example, I'm going to set a background image on an element (using my good friend, the bunny) and then use another image called `bw.png` to apply the mask.

Here's the relevant code:

```
div {
  background: url('images/bunny.png');
  -webkit-background-size: 17px 30px;
  -webkit-mask-image: url('images/bw.png');
  -webkit-mask-position: 50% 50%;
  -webkit-mask-repeat: no-repeat;
  -webkit-mask-size: 100%;
}
```

In Figure 8-11, I've put together an illustration of the three stages of the process. In the first box, you can see the image `bunny.png` tiled across the back of the element. In the second box is the image `bw.png`, which is the mask that will be applied to the element. And in the third box, the mask is applied and the background image shows through the black area.



Figure 8-11: Demonstrating the `-webkit-mask-` properties²

Note that this is similar to what we did earlier in the chapter with the text value for `-webkit-background-clip`.

As with the `background-*` properties, there is also a shorthand property, `-webkit-mask`, which accepts values in exactly the same way. I could have abbreviated the previous code like so:

```
div {
  background: url('images/bunny.png');
  -webkit-background-size: 17px 30px;
  -webkit-mask: url('images/css3.png') no-repeat 50% 50%;
  -webkit-mask-size: 100%;
}
```

2. This image is by Flickr user pasukaru76 (<http://www.flickr.com/photos/pasukaru76/5164791212/>) and is published under a Creative Commons Attribution license.

One property that doesn't have a `background-*` counterpart is `-webkit-mask-box-image`, which allows you to use a mask as a border. The syntax for this is based on the `border-image` property, which will be introduced in the next chapter and works in the same way. Note that here you can also use the CSS gradient syntax, which I'll fully introduce in Chapter 11.

You can read more about the WebKit mask properties in the blog post that introduced them (<http://webkit.org/blog/181/css-masks/>).

Summary

The new features introduced in this chapter are a big step toward the stated aim of CSS: to separate a page's content from its presentation. The more non-essential markup we can remove from our documents, the easier our pages will be to maintain and the better it will be for semantics.

The current leader in implementing these features is Opera, which from version 10.5 has very solid and stable support for this module, though I expect all other major browsers will catch up in the near future (including IE9, I hope).

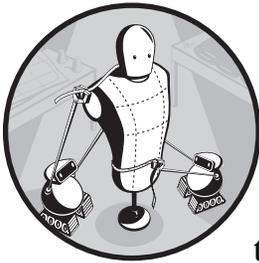
In the next chapter, I'll cover the other part of the Backgrounds and Borders Module: borders.

Background Images: Browser Support

	WebKit	Firefox	Opera	IE
Multiple background images	Yes	Yes	Yes	No (expected in IE9)
<code>background-size</code>	Yes (with prefix in Safari 4 and earlier)	Yes (with prefix; expected in Firefox 4 without prefix)	Yes	No (expected in IE9)
<code>background-clip</code>	Yes (with prefix in Safari 4 and earlier)	Yes (with prefix; expected in Firefox 4 without prefix)	Yes	No (expected in IE9)
<code>background-origin</code>	Yes (with prefix in Safari 4 and earlier)	Yes (with prefix; expected in Firefox 4 without prefix)	Yes	No (expected in IE9)
<code>background-repeat</code> (new values)	No	No	Yes	No (expected in IE9)
<code>image-rect</code>	No	Yes (with prefix)	No	No
<code>mask-*</code>	Yes (with prefix)	No	No	No

9

BORDER AND BOX EFFECTS



The ability to add borders to page elements has been around, almost unchanged, since the days of CSS1. What developers *wanted* to do with borders, however, outstripped what they *could* do with them years ago. Adding border effects such as rounded corners or shadows has probably been responsible for more extraneous empty markup elements than almost anything else in the web development world. Things that should be simple often involve some incredibly complex workarounds.

The second part of our look at the Backgrounds and Borders Module explores new methods of decorating elements without extra markup. You'll learn how to make rounded corners, how to use images for borders, and how to add drop shadows.

Giving Your Borders Rounded Corners

Since the earliest days of the Web, designers have been putting rounded corners on page elements. That they've had no way to create them without using images seems crazy. To create a box of flexible width with four rounded corners has meant creating four images and adding at least two extra nonsemantic elements, which makes maintaining a website much harder than it needs to be.

But no more. The Backgrounds and Borders Module introduces a way to round the corners of your elements using CSS alone. Each corner is treated as a quarter ellipse, which is defined by a curve that is drawn between a point on the *x*-axis and a point on the *y*-axis (you may remember those from Chapter 6). Figure 9-1 illustrates this more clearly.

A quarter ellipse can be *regular*, which means the length along both axes is the same, or *irregular*, which means the length along each axis is different. Figure 9-2 shows examples of both.

CSS3 defines these curves using the `border-radius` property. This property allows you to define the radius of the quarter ellipse simply using the following syntax:

```
E { border-v-h-radius: x y; }
```

In this syntax, *v* is a keyword value of `top` or `bottom`, *h* is a keyword value of `left` or `right`, and the *x* and *y* values are lengths along the axes that define the curve of the quarter ellipse. That sounds like a mouthful, but here's an example that should make it clearer:

```
div { border-top-right-radius: 20px 20px; }
```

This syntax will round the top-right corner of a `div` element with a radius of 20px horizontally and vertically, which is a regular curve.

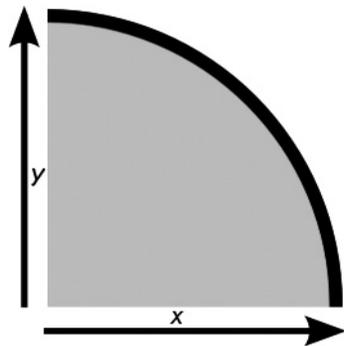


Figure 9-1: A quarter ellipse made by the curve between lengths on the *x* and *y* axes

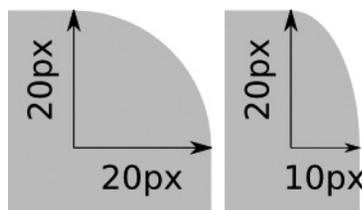


Figure 9-2: A regular curve (left) has identical values on both axes; an irregular curve has different values on each axis.

In fact, for regular curves `border-radius` lets you simplify even further by leaving out either the *x* or the *y* value; if one value is not specified, both are assumed to be equal. So if you wanted to apply that radius to each corner of your element, you would use this code:

```
div {
  border-top-left-radius: 20px;
  border-top-right-radius: 20px;
  border-bottom-right-radius: 20px;
  border-bottom-left-radius: 20px;
}
```

Figure 9-3 shows the result.

To create irregular curves, you just use different values on the individual properties, like so:

```
div {
  border-top-left-radius: 10px 20px;
  border-top-right-radius: 10px 20px;
  border-bottom-right-radius: 10px 20px;
  border-bottom-left-radius: 10px 20px;
}
```

This syntax creates the shape shown in Figure 9-4.



Figure 9-3: Four equal, regular rounded corners



Figure 9-4: Four irregular rounded corners

border-radius Shorthand

If having to write a different property for each corner strikes you as quite repetitive, you'll be happy to learn that a shorthand property is available. As with `border-width`, `margin`, and `padding`, you can specify one, two, three, or four values. Where those values refer to sides, however, the `border-radius` values refer to corners, starting at the top left:

```
E { border-radius: top-left top-right bottom-right bottom-left; }
E { border-radius: top-left top-right & bottom-left bottom-right; }
E { border-radius: top-left & bottom-right top-right & bottom-left; }
E { border-radius: top-left & top-right & bottom-right & bottom-left; }
```

The shorthand syntax can only be used like this if you have regular corners; that is, corners where the horizontal and vertical values are the same and are defined with a single length. (I'll cover the shorthand for irregular corners momentarily.)

To demonstrate the shorthand property in action, I'm going to draw three boxes, each with a different set of values:

```
.radius-1 { border-radius: 0 20px; }  
.radius-2 { border-radius: 0 10px 20px; }  
.radius-3 { border-radius: 0 0 20px 20px; }
```

You can see the output in Figure 9-5.



Figure 9-5: Effects of different values for the `border-radius` shorthand property

The first (left) box has two values for `border-radius`: The top-left and bottom-right have a value of 0, so are square, but the top-right and bottom-left are rounded with a radius of 20px. The second (middle) box has three values: The top-left is once again square, but now the top-right and bottom-left have a 10px radius, and the bottom-right has a value of 20px. Finally, the last (right) box has four values: The top-left and top-right have a value of 0, so are squared, whereas the bottom-right and bottom-left have radii of 20px.

You can also use the shorthand syntax with irregular curves. To achieve this effect, you'll need to list the values separated by a slash (/), like so:

```
border-radius: { horizontal-radius / vertical-radius; }
```

Each side of the slash can contain between one and four values, as with the shorthand for regular curves. This means, for example, you could have one value for the horizontal radius and four separate values for the vertical radii. Again, I'll demonstrate what this looks like in actual practice:

```
.radius-1 { border-radius: 20px / 10px; }  
.radius-2 { border-radius: 20px / 10px 20px; }  
.radius-3 { border-radius: 10px 20px 20px / 20px 10px; }
```

The results are shown in Figure 9-6.

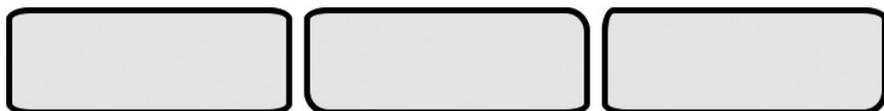


Figure 9-6: Irregular rounded corners produced with the `border-radius` shorthand property

The first (left) box has four equal corners of 20px horizontal and a 10px vertical radius. The second (middle) box has two corners of 20px/10px and two of 20px/20px. The last (right) box has a top-left corner of 10px/20px, a top-right and a bottom-left corner of 20px/10px, and a bottom-right corner of 20px/20px.

Differences in Implementation Across Browsers

The syntax used so far in this chapter is what appears in the specification, so you can consider it canonical. But to use rounded corners today, you should know how the various browsers implement `border-radius`.

Whereas Opera (10.5+) and recent versions of WebKit support the syntax according to the specification, previous WebKit browsers (such as Safari 4.04 and below) require the proprietary `-webkit-` prefix before each property, and Firefox versions below 4 use a different pattern altogether. Mozilla's browser currently uses the following syntax:

```
E { -moz-border-radius-vh: x y; }
```

That means to apply a radius to the top-left corner of an element the code has to be written like so:

```
div { -moz-border-radius-topleft: 20px 20px; }
```

Also, older versions of WebKit—such as that used in Safari until at least version 4.04—have an incorrect implementation of the `border-radius` shorthand property, which accepts only a single value. If you want to have anything other than four regular rounded corners, you have to specify individual values for each of the `border-*-radius` properties, as shown at the beginning of this chapter.

Using Images for Borders

Another common way to style elements is to use background images as decorative borders. With CSS2, however, you had no way to achieve this, and you had to use a lot of extra markup to get the required effect, with a subsequent penalty on semantics and maintainability. CSS3 introduces `border-image`, which provides a simple syntax to apply decorative borders:

```
E { border-image: source slice / width repeat; }
```

This property is supported by Opera, as well as Firefox and Safari (with their proprietary prefixes). However, as only Firefox supports the `width` value, the safe implementation would use this syntax:

```
E { border-image: source slice repeat; }
```

The first value, *source*, is the URL of the image you want to use for your borders. Next, *slice* is a length or percentage value (or series of values) that sets a distance from each edge of the image marking the area that will be used to “frame” the element; I’ll clarify this with an example shortly. *slice* can be between one and four values, similar to *margin*, *padding*, *border-radius*, and so on.

The *repeat* value takes either one or two keywords, which set the way the image is repeated along the top and bottom (first keyword) and left and right (second keyword) of the element. The possible values are *stretch*, which stretches the image to fill the border length; *repeat*, which tiles the image across the length; *round*, which tiles the image across the length using a whole number of tiles (the element will be resized to accommodate this); and *space*, which also tiles the image across the length using a whole number of tiles, but using spaces if the image doesn’t fit the element.

This all probably sounds a bit complicated, so here’s an example that should help to illustrate. Consider the following CSS:

```
div {  
  border-image: url('frame.png') 32 39 36 41 stretch;  
  border-width: 20px;  
}
```

Note here that no unit values are included for the numbers. The numbers serve two purposes: for bitmap images (such as JPG or PNG), they are pixel values; but for vector images (such as SVG), they are coordinate values. I mentioned previously that percentages are also allowed, in which case, you would use the percent sign (%).

Back to the example. Figure 9-7 shows the original image I’m going to use as my *border-image* (*frame.png*) and then, on the right, that image applied as a border using the previous code.

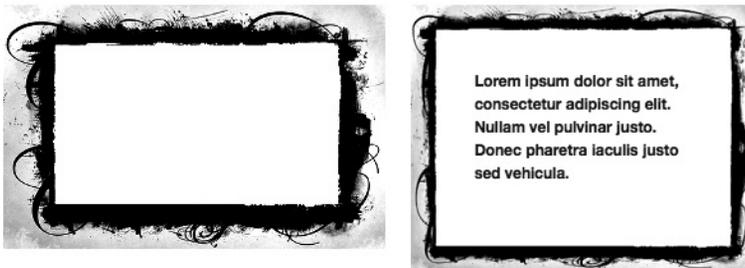


Figure 9-7: The original image (left) and the image as it appears when displayed with *border-image* (right)¹

Let me walk you through the code. First, I specify *frame.png* as the image I’ll be using for the border. The next four numerical values set the areas I want to slice: 32px from the top, 39px from the right, 36px from the bottom,

1. The border image is by Flickr user SkeletalMess (<http://www.flickr.com/photos/skeletalmess/4396262635/>) and is published under a Creative Commons Attribution license.

and 41px from the left. These values specify which parts of *frame.png* I want to use as my top, right, bottom, and left borders, respectively. Figure 9-8 shows where my slices are on my frame image.

The image is cut into nine slices: four sides, four corners, and the center. The corners and the center will always remain constant, scaling to fill their set positions, but the four sides can be modified with the values specified in the property. In this example, I've set the *repeat* value to stretch, which means the side slices will be stretched to fill the length (height or width) of the element. I also set a border-width of 20px on each side, so my image slices are scaled to fit inside that limit.

Now to illustrate the difference between the stretch and round keywords for the *repeat* value. Check out this code:

```
.bimage-1 { border-image: url('frame.png') 32 39 36 41 round stretch; }  
.bimage-2 { border-image: url('frame.png') 32 39 36 41 stretch; }
```

Figure 9-9 compares the two results.

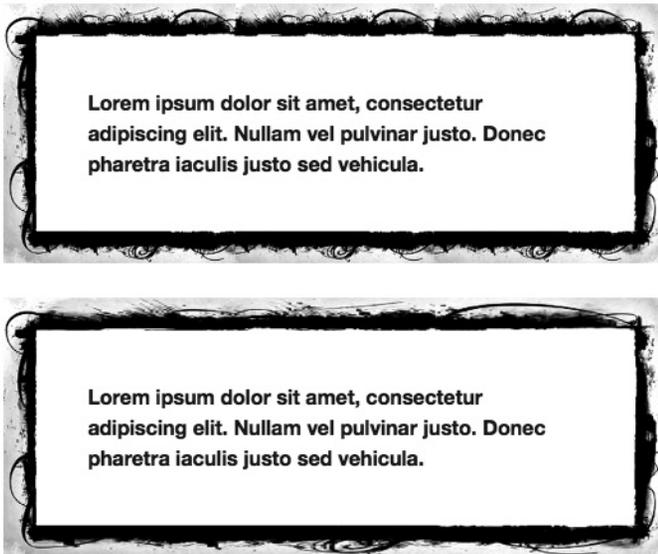


Figure 9-9: *border-image* repeated horizontally (top) and stretched horizontally (bottom)

The first box has a value of *round stretch*, so the image is repeated three times horizontally but scaled vertically. The second box has a value of *stretch*, so the image is scaled on both axes.

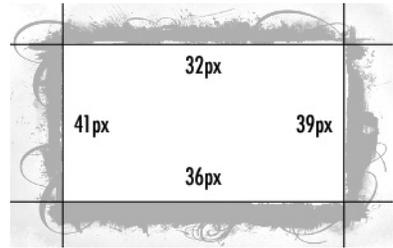


Figure 9-8: Where the four specified values in *border-image* will slice the image

In the module itself, `border-image` is said to be the shorthand for a series of subproperties:

```
E {
  border-image-source: image;
  border-image-slice: length; /* Up to four values allowed */
  border-image-width: length; /* Up to four values allowed */
  border-image-outset: length; /* Up to four values allowed */
  border-image-repeat: keyword; /* Up to two values allowed */
}
```

You'll be familiar with some of these properties from the shorthand property: `border-image-source` is the same as the *source* value; `border-image-slice` is the same as *slice*; and `border-image-repeat` is the same as *repeat*.

The `border-image-width` property has the same syntax as the *slice* value but a different function: It sets the width of the element's border and takes precedence over `border-width`. (`border-image-width` is used to provide a fallback state, as with `text-fill-color`, which I introduced in Chapter 6.) As I mentioned previously, you can use this property in Firefox's shorthand property but not in other browsers. The `border-image-outset` property also uses the same syntax as *slice*, but it sets the length that the border can extend outside the element.

These properties were only introduced to the specification at the end of 2009, however, and as of this writing, no browser supports them, although they are likely to be implemented in the future.

Multicolored Borders

Firefox has a series of proprietary properties that let you define multiple colors on a border. The syntax is very similar (one letter different!) to the `border-*-color` properties:

```
E { -moz-border-*-colors: colors; }
```

The asterisk represents a keyword for a side—*top*, *right*, *bottom*, *left*—and *colors* is a list of space-separated color values, each of which will color a single pixel-width of the border. For example, if you have a 3px left border and wanted to color it like the Italian flag, you would use:

```
div {
  border-left-width: 3px;
  border-left-colors: green white red;
}
```

As each color only applies to a single pixel-width, you may have borders with a thicker width than you have defined colors for. In this case, the last color specified will repeat for the rest of the border's width.

Here's the code for two examples:

```
.colors-stripe { -moz-border-*-colors: black white black white black white; }  
.colors-fade { -moz-border-*-colors: #222 #444 #666 #888 #AAA #CCC; }
```

Figure 9-10 shows these examples in action. Note, however, that to give clarity to this code example I've used only one property for each element; in practice, you have to specify all four sides to get the results shown. In other words, this property has *no shorthand* property.



Figure 9-10: Different values supplied to Firefox's `border-*-colors` property (detail)

Both boxes have a border width of 6px. The left box's border alternates black and white colors, and the right uses several shades of gray that get darker as you get farther away from the element.

Adding Drop Shadows

In Chapter 6, we looked at a way to add drop shadows to text with the `text-shadow` property, but CSS3 also has a method for adding shadows to box elements using the `box-shadow` property. The syntax for `box-shadow` is similar to that of `text-shadow`:

```
E { box-shadow: inset horizontal vertical blur spread color; }
```

The first value, *inset*, is an optional keyword that sets whether the shadow sits inside or outside of the element. If *inset* is not specified, the shadow will sit outside. The next two values are lengths, which set the *horizontal* and *vertical* distance of the shadow from the box; if you want to have a shadow, these values are required.

The next value sets the *blur* radius and is again a length value. Then you have *spread*, which sets the distance the shadow spreads. A positive length makes the shadow larger than its element, and a negative length makes it smaller. Both of these values are optional.

Finally you have the *color* value. In WebKit, this value is required, but in Firefox and Opera, it's optional. If left unspecified, *color* will default to black.

Now I'll put them together in an example:

```
div { box-shadow: 4px 4px 3px #666; }
```

That syntax creates a dark gray shadow outside of the element, positioned at a distance of 4px, both horizontally and vertically, with a blur radius of 3px.

Next, I'll demonstrate this and further examples of the effects of different values on the `box-shadow` property, using the following CSS:

```
.shadow-one { box-shadow: 4px 4px; }  
.shadow-two { box-shadow: 4px 4px 3px; }  
.shadow-three { box-shadow: 12px 12px 2px -6px; }  
.shadow-four { box-shadow: #999 4px -4px 2px 0; }  
.shadow-five { box-shadow: #999 4px -4px 2px 0, -4px 4px 2px; }
```

The results are shown in Figure 9-11. The elements in the code correspond to the boxes in the illustration, moving from left to right.

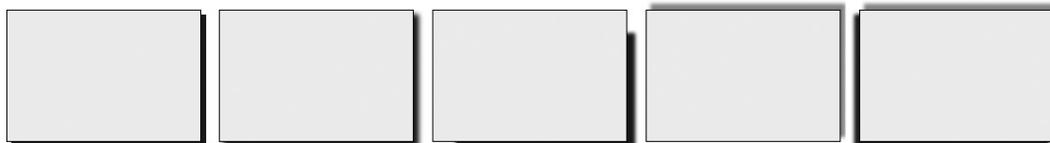


Figure 9-11: The effects of using different values for the `box-shadow` property

The first is the simplest shadow, simply distanced 4px both horizontally and vertically from the element. The second has the same distance values as the first but also adds a blur radius of 3px to soften the shadow's edges. The third has a distance of 12px along both axes but a negative *spread* value (-6px), which makes the shadow smaller than its box. The fourth example has a medium gray-colored shadow with a negative vertical distance, meaning the shadow falls above the element instead of below it. Finally, the fifth box has two shadows applied to it: The first is the same as in the fourth box, and the second is a black shadow with a negative horizontal distance, making the shadow fall to the left of the box.

I briefly mentioned the optional `inset` keyword at the beginning of this section. If present, this keyword draws a shadow on the interior of the box, but also has the effect of “flipping” the shadow to the other side of the box. What I mean is that where a regular—that is, *outset*—shadow with positive *x* and *y* values would appear at the bottom right of the box, an *inset* shadow would appear at the top left.

To illustrate I'll use the same code as for the previous example but add the `inset` keyword to each one:

```
.shadow-one { box-shadow: inset 4px 4px; }  
.shadow-two { box-shadow: inset 4px 4px 3px; }  
.shadow-three { box-shadow: inset 12px 12px 2px -6px; }  
.shadow-four { box-shadow: inset #999 4px -4px 2px 0; }  
.shadow-five { box-shadow: inset #999 4px -4px 2px 0, inset -4px 4px 2px; }
```

The result is shown in Figure 9-12.



Figure 9-12: Inset shadows

Here you see almost the inverse of Figure 9-11; all offset, blur radius, and color values are the same, but the shadows now appear on the interior of the boxes and in the opposite corners.

The `box-shadow` property is pretty well implemented in browsers: WebKit and older version of Firefox (3.6 and below) implement it with a proprietary prefix and Opera, IE9, and Firefox 4 with the standard syntax. One caveat: As with `border-radius`, older versions of WebKit (such as in Safari 4.04 and below) use a different implementation—one with no *spread* value. This means that some of the code used in this chapter may display incorrectly in those browsers.

Summary

I mentioned at the beginning of the previous chapter that the Backgrounds and Borders Module is a priority for the W3C because of the clamor from web developers. The new properties it introduces are extremely useful for removing extraneous elements from markup and giving developers finer control over the way pages are laid out. With a multitude of new background and border properties at our disposal, creating websites that can be viewed at many different sizes and resolutions is going to be much easier, and our sites will be more suited to the cross-platform ideal of the age.

In the next chapter, I'll continue with the theme of page decoration, as we take a look at the new color and opacity controls that CSS3 provides.

Border and Box Effects: Browser Support

	WebKit	Firefox	Opera	IE
<code>border-radius</code>	Yes (with prefix in Safari 4 and below)	Yes (with prefix; incorrect syntax on subproperties; expected in Firefox 4 without prefix)	Yes	No (expected in IE9)
<code>border-image</code>	Yes (with prefix)	Yes (with prefix)	Yes	No
Multiple <code>border-color</code> values	No	Yes	No	No
<code>box-shadow</code>	Yes (with prefix)	Yes (with prefix)	Yes	No

10

COLOR AND OPACITY



Color in CSS2 was based around the *Red, Green, Blue (RGB)* model; whether you used hexadecimal notation or the *rgb* color value, you had to combine those three colors to add color to your pages. Of course, designers speak in terms of shades and tints: When one of them says to use a “50 percent tint” of a certain color, developers have had to use the RGB model to match that color, which has often involved some chicanery with a graphics package to find the exact tone needed.

The CSS Color Module (<http://www.w3.org/TR/css3-color/>) has a solution to that problem—and more besides. For starters, it introduces the concepts of opacity and transparency through the *opacity* property and the *Alpha* color channel. In addition, the CSS Color Module adds an entirely new color model, which is more intuitive and easier to tweak to find the perfect tone.

The Color Module is a Proposed Recommendation and is pretty well implemented in every browser except Internet Explorer (at least up to version 8), so with a little bit of careful coding to provide a fallback state for those older versions of Internet Explorer, you can begin to use it straight away.

Setting Transparency with the opacity Property

Opacity and transparency sit at opposite ends of a scale. Strictly speaking, they are the measure of an object's resistance to light—the more opaque something is, the less light it lets through; the more transparent something is, the more light it lets through. An object with no transparency is fully opaque, and an object with no opacity is fully transparent. In CSS both are measured using the opacity property. In essence, with `opacity`, you are setting how much of the background can be seen through the specified element.

The opacity property has the following syntax:

```
E { opacity: number; }
```

The *number* value is a decimal fraction—i.e., a number between 0.0 and 1.0—where 0.0 is fully transparent, 1.0 is fully opaque, and any value between those two is a blend of opacity and transparency. For example, to set a `p` element to be 50 percent opaque (or 50 percent transparent, depending on whether your glass is half empty or half full), you would use the following code:

```
p { opacity: 0.5; }
```

To further demonstrate, I'll show three `div` elements (each with a single child `p` element) and display each element in the same way, except for a change to the opacity value. Here's the relevant code for this example:

```
div { background-color: black; }  
p { color: white; }  
.semi-opaque-1 { opacity: 0.66; }  
.semi-opaque-2 { opacity: 0.33; }
```

You can see the output in Figure 10-1.

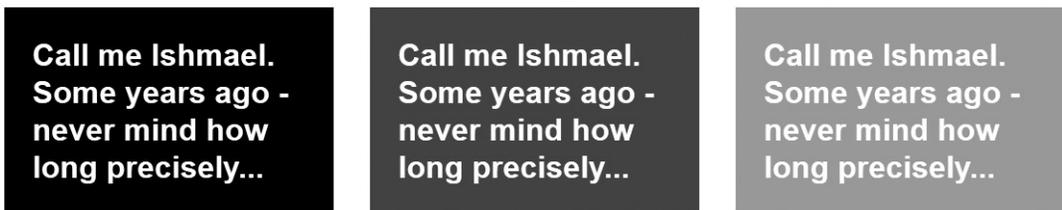


Figure 10-1: The effect of differing opacity values on three identical elements

The first (left) box has no explicitly set value for opacity, so it defaults to 1.0, or fully opaque. The next (middle) box has a value of 0.66, so its opacity is reduced by a third, causing the black to appear as a dark gray (a blend of the black background color of the box and the white background color of the body element, which shows through). Finally, the last (right) box has an opacity value of 0.33, so it can be considered 67 percent transparent, making the box a lighter gray color.

Now, here's a very important point to remember about this property: *The value of the opacity property is inherited by all of its child elements.* If I set an `opacity` value of 0.5 on an element, its children will never be able to be more opaque than that. This setting can't be overruled by any other property—or rather, you can never make an element more opaque than its parent, but you *can* make it less opaque.

The example in Figure 10-1 doesn't make this point very obviously, as the `p` element has the color value of `white`, which, on the white background of the element behind it, doesn't make any noticeable difference. But in the following example, you'll see the difference more clearly. Consider these CSS rules:

```
.box { background-image: url('bunny_sprite.png'); }
.text { background-color: white; }
p { color: black; }
```

I'll apply them to a box that uses the following markup:

```
<div class="box">
  <div class="text">
    <p>Call me Ishmael</p>
  </div>
</div>
```

You can see the result in Figure 10-2.



Figure 10-2: Nested `div` elements with some styling applied

Now I'll add the `opacity` property to the inner `div`, like so:

```
.box { background-image: url('bunny_sprite.png'); }
.text {
  background-color: white;
  opacity: 0.7;
}
p { color: black; }
```

You can see the result in Figure 10-3. The text that was fully opaque in the original version (Figure 10-2) had a true black color. But in the second version (Figure 10-3), to which I've applied the `opacity` property, the text box is semi-opaque and so is the text contained within it.



Figure 10-3: The inner `div` has an `opacity` value of 0.7

The `opacity` property may seem somewhat limiting, as you can't apply it to elements with children without also applying it to those children, as I've just demonstrated. CSS3 has a new method to get around this, however; it's called the Alpha channel, and I'll explain it in the next section.

The `opacity` property is currently supported in Firefox, WebKit, and Opera, and planned for inclusion with IE9.

New and Extended Color Values

CSS2.1 allowed three methods for specifying color values: keywords (like `black`), hexadecimal notation (like `#000`), and RGB (like `0,0,0`). In CSS3, the range is expanded by a completely new method of specifying colors, as well as the introduction of opacity through the Alpha channel.

The Alpha Channel

The *Alpha* channel (Alpha for short) is the measure of the transparency of a color—as opposed to the `opacity` property, which is the measure of the transparency of an element. So although color values using Alpha can be inherited by child elements like any other color value, the overall opacity of the element is not affected.

CSS3 introduces Alpha as a value in the *RGBA* color model. *RGBA* stands for *Red*, *Green*, *Blue*, *Alpha*, and the syntax is the same as for the RGB value used in CSS2 but with the Alpha value specified by an extra comma-separated argument at the end:

```
E { color: rgba(red, green, blue, alpha); }
```

The value of that *alpha* argument is the same as the value provided for *opacity*: a decimal fraction from 0.0 to 1.0, which is once again a measure between full transparency (0.0) and full opacity (1.0). If you wanted to set a *p* element to have a black color and 50 percent opacity, you would use the following code:

```
p { color: rgba(0,0,0,0.5); }
```

As mentioned, *rgba* differs from *opacity* in a couple of ways: First, *rgba* is a color value, so you couldn't, for example, use it to change the opacity of an image (or an element with a background image). Second, the *rgba* value applies only to the element it is specified on, so child elements can overrule any inheritance.

To more precisely illustrate the difference between the two, I'll show a modified version of the examples shown in Figures 10-2 and 10-3. I'll use the same markup, but this time I'll set the *opacity* of one and give an *rgba* value to the *background-color* of another, both with the same value:

```
.opacity { opacity: 0.5; }  
.rgba { background-color: rgba(255,255,255,0.5); }
```

You can see the results in Figure 10-4.

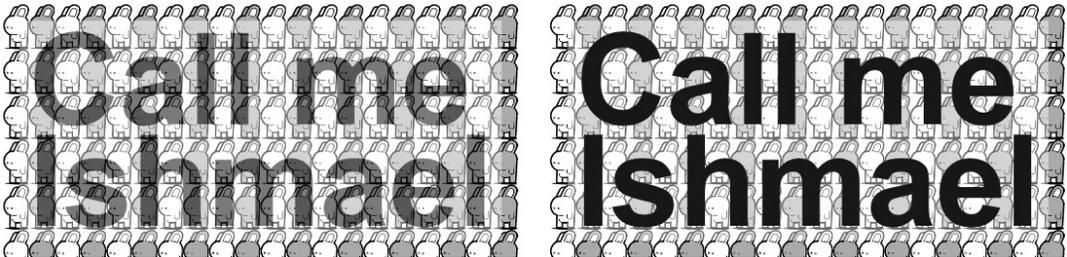


Figure 10-4: Comparing opacity (left) and RGBA (right)

The difference is pretty clear. Both boxes have the same level of transparency, but in the first (left) box, the *opacity* value has been inherited by its child *p* element, making the text semi-transparent also. In the second (right) box, the *rgba* value for the *color* property has not been inherited, so the *p* element retains the fully opaque black color.

Having established that *rgba* is not the same as *opacity*, let's see how it works. Being a color value, you can obviously use it for backgrounds, borders, shadows, and so on. The following code shows some examples of *rgba* applied to different properties:

```
❶ .shadow .text { box-shadow: 10px 10px 4px rgba(0,0,0,0.7); }  
❷ .border .text { border: 10px solid rgba(0,0,0,0.5); }  
❸ .text-semi p { color: rgba(0,0,0,0.6); }  
❹ .text-shadow p { text-shadow: 5px 5px 1px rgba(0,0,0,0.6); }
```

Figure 10-5 shows these properties in action.

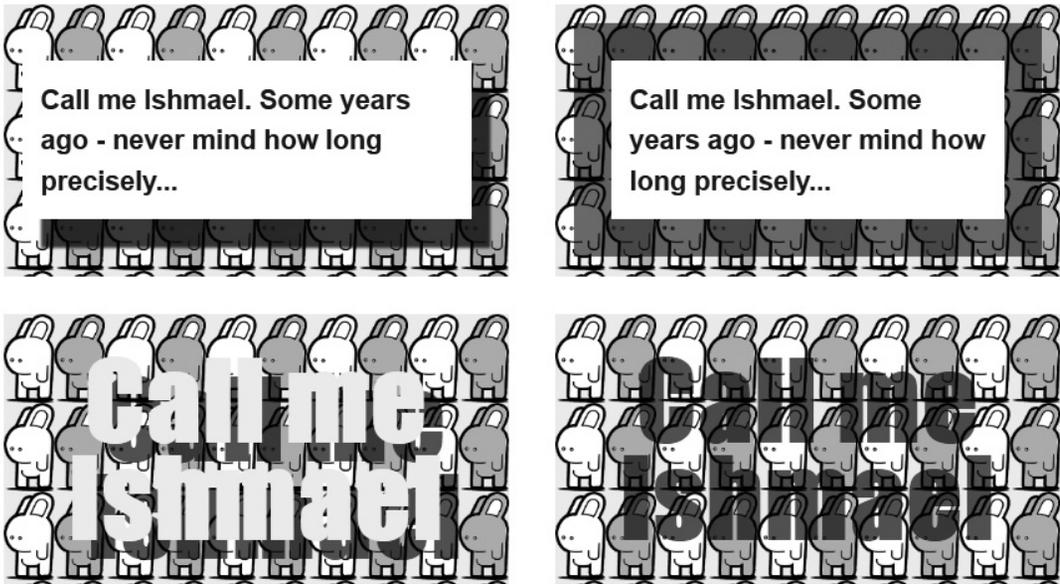


Figure 10-5: RGBA applied to different properties

Moving clockwise from top left, in the first box, `rgba` lowers the opacity of box-shadow; setting the `alpha` value to 0.7 ❶ allows some of the background to show through, making the shadow more “realistic.” The next example shows a 50 percent opaque black border ❷ (which I used in the example shown in Figure 8-4 on page 99). (I’ll explain an important issue to bear in mind when using this technique in WebKit browsers in “Border Overlap Issue in WebKit” on page 125.) In the next example, the `alpha` value of the color property has been set to 0.6 ❸, which makes the text appear semi-opaque. And finally the last example shows another shadow effect, this time on the text-shadow property. The `alpha` value is set at 0.6 ❹, which, once again, makes for a more realistic shadow.

Like opacity, RGBA color values are currently supported in Firefox, WebKit, and Opera, and are planned for inclusion with IE9.

RGBA and Graceful Degradation

Older browsers that don’t support RGBA values will ignore any rules that use them and default to a previously specified or inherited value. To compensate, you should specify the color twice, using the cascade to ensure the right color is implemented:

```
p {
  color: #F00;
  color: rgba(255,0,0,0.75);
}
```

In this example, browsers that don't support RGBA values ignore the second color property and apply the first color property. Of course, this result means that a fully opaque color will be used instead of a semi-opaque one, so check your design thoroughly to make sure it isn't affected negatively.

The same goes for all new color values introduced in the rest of this chapter.

Border Overlap Issue in WebKit

As I mentioned, all WebKit browsers currently have an issue regarding the implementation of RGBA on the border-color property. In those browsers, borders that have RGBA color values will show an overlap of the colors on the corners.

To see what I mean, consider the following CSS rule:

```
div { border: 50px solid rgba(0,0,0,0.5); }
```

When viewed in a WebKit browser, this code gives the result shown in Figure 10-6.

As you can see, the corners are darker due to the overlap of two semi-opaque edges. The WebKit team is aware of this bug but currently has no fix for it. Neither Firefox nor Opera have this issue.

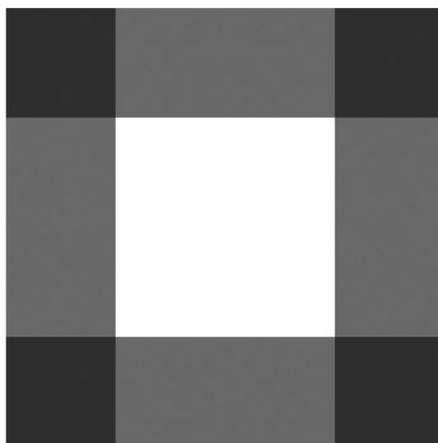


Figure 10-6: Issue with RGBA colors on borders in WebKit

Hue, Saturation, Lightness

Summing up exactly what HSL is without providing a course in color theory is hard, but I'll do my best: *HSL*—which stands for *Hue*, *Saturation*, *Lightness* (sometimes called *luminance*)—is a cylindrical-coordinate representation of RGB. Still not clear? Okay, what about: HSL is like all the possible RGB colors mapped in three dimensions? No? Alright, take a look at Figure 10-7.

All the possible colors are arranged in a cylinder with a central axis. The angle around the axis is the *hue*, the distance from the axis is the *saturation*; and the distance along the axis is the *lightness*. The combination of those three values creates a unique color.

Hue represents the major colors, starting and ending with red (0 or 360) and including all the main colors between. Think of the colors of the visible spectrum (or the colors of the rainbow) you learned about in school—red, orange, yellow, green, blue, indigo, violet—arranged around the circumference of a circle; the value of hue is a degree around that circumference that points to a specific color.

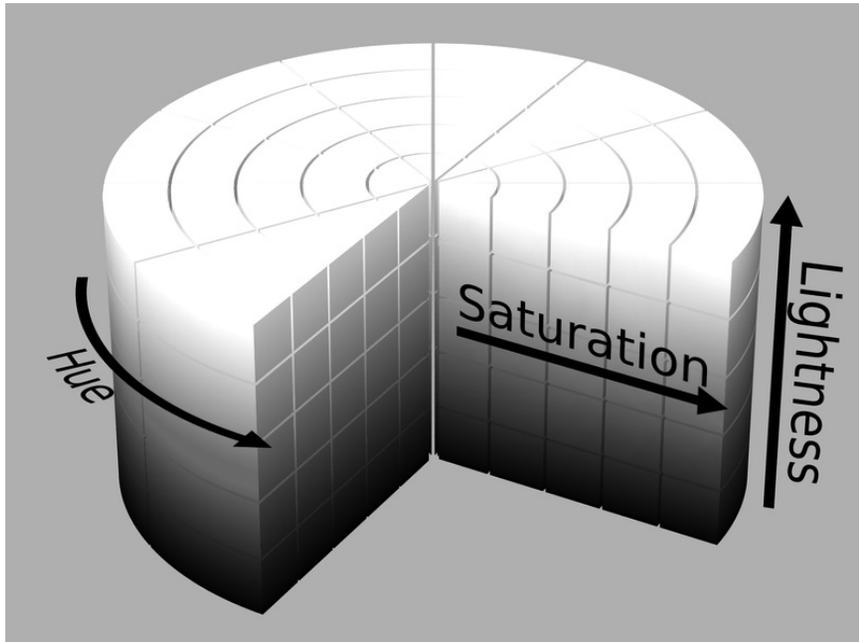


Figure 10-7: Diagram explaining the HSL color method¹

Saturation is the strength or intensity of that color: 0 percent is no strength, which would make your color a shade of gray, and 100 percent is full strength, the most intense version of that color.

Lightness is the brightness or darkness of the color: 50 percent is the true color, 0 percent is black, and 100 percent is white.

If you've never encountered HSL before and my explanation has still left you baffled, don't worry—for our purposes, you just need to understand that `hsl` is a color value that takes three arguments, with a similar syntax to RGB:

```
E { color: hsl(hue,saturation,lightness); }
```

Hue accepts a value of 0 to 360, and saturation and lightness accept values from 0 percent to 100 percent. Some simple color examples with their hexadecimal and RGB equivalents are shown in Table 10-1.

Table 10-1: Common Color Equivalents for Four Color Values

HSL	RGB	Hexadecimal	Keyword
0,0%,0%	0,0,0	#000000	Black
360,0%,0%	255,255,255	#FFFFFF	White
0,100%,50%	255,0,0	#FF0000	Red
120,100%,25%	0,128,0	#008000	Green
240,100%,50%	0,0,255	#0000FF	Blue

1. This image is taken from Wikimedia (http://en.wikipedia.org/wiki/File:HSL_color_solid_cylinder_alpha_lowgamma.png) and is published under a Creative Commons Attribution license.

NOTE *Even if you are using a 0 (zero) value for saturation and lightness, you must still specify the value as a percentage.*

The table doesn't actually tell the whole story—many different combinations of values allow you to create pure white, for example. Really, the best way to learn more about HSL is to get hold of a color picker that supports it and play around for a while. HSL plug-ins are available for Photoshop and most other graphics packages, and you should also be able to find color conversion tools online. Be careful, however, not to confuse HSL with Hue, Saturation, Value (HSV). Although they use an identical syntax, the color values are mapped differently so you will not get the same results.

The advantage of HSL over RGB (or hexadecimal) is that it allows you to more quickly try different variants of a color, making it more useful for people who design websites. If you're a developer coding from other people's designs, you may want to continue using RGB. HSL is simply a new alternative to consider.

HSL color values are currently supported in Firefox, WebKit, and Opera and, once more, are planned for inclusion in IE9.

HSLA

If you've decided that HSL is the color method for you, then you'll also be able to utilize the Alpha channel for transparency with the `hsla` color value. Like its counterpart `RGBA`, `hsla` simply extends the color scheme with an extra argument in the syntax:

```
E { color: hsl(hue,saturation,lightness,alpha); }
```

So, for example, if you wanted a `p` element with a color value of red and 50 percent transparency, you'd use the following code:

```
p { color: hsl(0,100%,50%,0.5); }
```

Note that the overlapping borders issue with `RGBA`, seen in Figure 10-6, also applies to `HSLA`.

The Color Variable: `currentColor`

In addition to the new color methods I've just described, CSS3 also introduces a new color value keyword: `currentColor`. This keyword acts as a variable, which means its value is inherited, and it will have a different value depending on where you apply it in the document tree.

The value of `currentColor` for an element is the value of its own `color` property and, like `color`, `currentColor` can be inherited. Let's say an element has a `color` value of red, the value of `currentColor` is red. You can then use that to set a color value on a different property without having to specify red again.

The following example should clarify the usefulness of `currentColor`. First, I take the following markup:

```
<p>The Central Intelligence Agency (<abbr>CIA</abbr>).</p>
<p>The Federal Bureau of Investigation (<abbr>FBI</abbr>)</p>
```

And I apply this CSS to it:

```
p { color: black; }
p:last-child {
  background-color: black;
  color: white;
}
p abbr { border-bottom: 6px double currentColor; }
```

One paragraph will display in black (black) text on the default (white) background, and the other in white text on a black background. Next, I used the `currentColor` keyword as a value for the `border-bottom` property on the `abbr` elements. You can see the result in Figure 10-8.

The Central Intelligence Agency (CIA).



Figure 10-8: A demo of the `currentColor` color value keyword

Because the first paragraph has a color value of black, the color value of the `border-bottom` property of the `abbr` element is also black. Because the second paragraph has a color value of white, the `border-bottom` property of the `abbr` element has the same color value. These values have inherited the color property of their parent elements.

The `currentColor` keyword means I don't have to specify the color of the border for every instance of the `abbr` element. In this example, I don't have to use an extra rule—without it, I would have to use something like this:

```
p abbr { border-bottom: 6px double black; }
p:last-child abbr { border-bottom-color: white; }
```

Although this may not seem like a big savings, it means I can update the parent element color and not have to worry about setting the color on any relevant children. On a large site with many different color combinations, you can see that `currentColor` would be extremely handy.

The `currentColor` value is currently implemented in Firefox, WebKit, and Opera, and is planned for inclusion in IE9.

Matching the Operating System's Appearance

In CSS2, you could use colors from different aspects of your operating system to give websites a more “native” appearance. You could, for instance, match the background color of a button element on a web page with that of a button element on your system by using the following code:

```
button { background-color: ButtonFace; }
```

This functionality has been deprecated in CSS3 and has been replaced by the `appearance` property, which is introduced in the Basic User Interface Module. Here's the syntax:

```
E { appearance: keyword; }
```

The *keyword* is from a long list of user interface elements, including *button*, *radio-button*, *password*, and so on. Firefox and WebKit both support this property with proprietary prefixes, and both have different lists of accepted values. Rather than list all of the values here, take a look at the links to each browser's CSS reference pages, which you can find links to in Appendix B.

To provide a quick demonstration, I'll use this code:

```
p.button { appearance: button; }  
button.native { appearance: none; }
```

And I'll apply it to this markup:

```
<button>Go</button>  
<p class="button">Go</p>  
  
<button class="native">Go</button>
```

The result is shown in Figure 10-9.



Figure 10-9: Different button appearances in WebKit on Ubuntu

The first two elements appear to be pretty much identical, but they aren't: The first is a `button` element with the default appearance value of `button`, taking its style from the operating system; the second is a `p` element with the `button` value explicitly set on the `appearance` property, matching the `button` element itself. The third example is also a `button` element, but it has an `appearance` value of `none`, so it displays as a browser default, without any reference to the operating system.

NOTE *The examples shown in Figure 10-9 are as they appear in WebKit on Ubuntu Linux and will look different if you use any other operating system.*

Summary

The introduction of transparency to CSS may seem minor, but transparency could potentially cause some significant changes to page designs; overlapping colors have long been a staple of print design, but this style has yet to make serious inroads into web design because of how difficult it is to implement.

The `appearance` property is also a small change with big implications. HTML5 is arriving and bringing with it expanded scope for building web applications and further blurring the boundaries between desktop and Web. More web applications will frequently blend with a user's operating system in the near future.

In the next chapter, I'll complete the quartet of chapters on backgrounds, borders, and colors with a look at the bleeding edge of CSS3: the gradient background.

Color and Opacity: Browser Support

	WebKit	Firefox	Opera	IE
opacity	Yes	Yes	Yes	No (expected in IE9)
RGBA values	Yes	Yes	Yes	No (expected in IE9)
HSL values	Yes	Yes	Yes	No (expected in IE9)
HSLA values	Yes	Yes	Yes	No (expected in IE9)
currentColor value	Yes	Yes	Yes	No (expected in IE9)
appearance	Yes	Yes	No	No

11

GRADIENTS



The word “gradient” has many different meanings, but in CSS, a *gradient* is strictly defined as a gradual transition between a range of (at least two) colors. CSS gradients are a fairly recent development but have already been through a number of changes. They were originally proposed by the WebKit team in April 2008, modified from the syntax proposed for the canvas element in HTML 5. In August 2009, Mozilla announced that an implementation slightly modified from that of WebKit’s would be in the next version of Firefox (3.6).

Since then, however, the W3C’s CSS Working Group proposed a further modified syntax, and this syntax is in the latest revisions of the Image Values Module (<http://www.w3.org/TR/css3-images/>). Following a quick turnaround from the Mozilla team, the new syntax made it into Firefox 3.6.

The WebKit team has indicated that its existing implementation will almost certainly be changed to the modified one, but the team wants to discuss one or two elements of the implementation first. As of this writing, the original syntax is implemented in all WebKit browsers.

Unlike most of the other chapters in this book, I'll show the two different syntaxes in each example. We have no way of knowing when WebKit will change its syntax, so I'll discuss the methods as they are currently implemented. Throughout the chapter I'll refer to "the Firefox syntax" rather than "the W3C syntax," as the specification is still very new and subject to change.

The two syntaxes have a number of differences, but the first and most obvious is that the Firefox implementation uses two functions (`-moz-linear-gradient` and `-moz-radial-gradient`), whereas WebKit uses only a single function (`-webkit-gradient`) with two type values (*linear* and *radial*). The syntaxes will become clearer as the chapter progresses.

NOTE *Just as this book was going to print, the WebKit team implemented the W3C standard properties, with the `-webkit-` prefix: So although these properties are not in any browser as I write this, in the future you should use them. However, for the sake of backward compatibility, the current implementation that's explained in this chapter will be kept in WebKit browsers for the foreseeable future.*

Linear Gradients

A *linear gradient* is one that gradually transitions between colors over the distance between two points in a straight line. At its simplest, a linear gradient will change proportionally between two colors along the full length of the line.

Linear Gradients in Firefox

Here's the syntax for a linear gradient in Firefox:

```
E { background-image: -moz-linear-gradient(point or angle, from-stop, color-stop, to-stop); }
```

I'll explain each part of that syntax in detail throughout the rest of this section, but I want to start with an example of the simplest possible set of values in Firefox:

```
div { background-image: -moz-linear-gradient(white, black); }
```

Each different color that you want the gradient to pass through is supplied as a value known as a *color-stop*, and the simplest gradient requires two: a start, which I'll refer to as a *from-stop* (to avoid using the term "start-stop"!), and an end, which I'll call a *to-stop*. In this first example, the gradient will start white and end black, passing gradually through all the variations between the two colors. You can see this in Figure 11-1.



Figure 11-1: A simple top-bottom, two-color linear gradient

Note that this gradient begins at the top of the box and moves vertically to the bottom—this is because of the *point* value in the syntax. The *point* is the position the gradient starts from and is specified with a similar syntax to

background-position; that is, either by keywords (top, bottom, left, right, center) or a percentage value. Point requires two arguments (left center, top right, 0% 50%, and so on) but if only one is specified, then the other is assumed to be center (or 50%). Therefore, an argument of left is assumed to be left center, and an argument of 100% is assumed to be 100% 50%. If no *point* value is provided, as in the first example, the value is assumed to be top center or 0% 50%, which is the top of the box.

I could have used any combination of the following to get the same result:

```
div { background-image: -moz-linear-gradient(center top, white, black); }
div { background-image: -moz-linear-gradient(top, white, black); }
div { background-image: -moz-linear-gradient(50% 0%, white, black); }
```

An alternative to *point* is to use an *angle* argument. Angles can be declared using a few different units: the Values and Units Module allows degrees, grads, radians, and turns, but I'm going to stick with degrees (deg) as they're the most commonly understood. (See <http://www.w3.org/TR/css3-values/#angle/> for more information on the others.) The *angle* value sets the angle of the gradient: 0deg (or 360deg) goes from left to right, 45deg from bottom left to top right, 90deg from bottom to top, and so on. You can also use negative values, which go counterclockwise: -45deg is the same as 315deg, -315deg is the same as 45deg . . . you get the idea.

So for the previous example, you also have these possible options:

```
div { background-image: -moz-linear-gradient(270deg, white, black); }
div { background-image: -moz-linear-gradient(-90deg, white, black); }
```

Linear Gradients in WebKit

WebKit's implementation is pretty similar to that of Firefox's, although angle values are not permitted, and the syntax is somewhat more precise and verbose:

```
E { background-image: -webkit-gradient(type, start-point, end-point,
from(from-stop), color-stop(color-stop), to(to-stop)); }
```

The immediate point of interest is that, as mentioned in the introduction to this chapter, you have to specify the type of gradient as a value. This value can be either linear or radial, but for now I'll stick to linear. Also, both start and end points are required, and from-stop and to-stop are specified with the from() and to() functions.

The WebKit way to achieve the simple example shown in Figure 10-1 requires the following declaration:

```
div { background-image:
  -webkit-gradient(linear, center top, center bottom, from(white), to(black));
}
```

As with Firefox, I could also use percentage values for the start and end points:

```
div { background-image:
      -webkit-gradient(linear, 50% 0%, 50% 100%, from(white), to(black));
}
```

But, unlike Firefox, angle values are not permitted, and two arguments are required for each; no assumption is made if one is left out.

Using Linear Gradients

Keeping the differences between the two syntaxes in mind, I'm going to present five different examples and then walk you through the code required to create them. Here's the relevant CSS snippet:

```
❶ .gradient-1 {
  background-image: -moz-linear-gradient(left, white, black);
  background-image: -webkit-gradient(linear, left center, right center, from(white), to(black));
}
❷ .gradient-2 {
  background-image: -moz-linear-gradient(right, white, black);
  background-image: -webkit-gradient(linear, right center, left center, from(white), to(black));
}
❸ .gradient-3 {
  background-image: -moz-linear-gradient(50% 100%, white, black);
  background-image: -webkit-gradient(linear, 50% 100%, 50% 0%, from(white), to(black));
}
❹ .gradient-4 {
  background-image: -moz-linear-gradient(0% 100%, white, black);
  background-image: -webkit-gradient(linear, 0% 100%, 100% 0%, from(white), to(black));
}
❺ .gradient-5 {
  background-image: -moz-linear-gradient(225deg, white, black);
  background-image: -webkit-gradient(linear, 100% 0%, 0% 100%, from(white), to(black));
}
```

These examples are shown in Figure 11-2.



Figure 11-2: Examples of different point values for linear gradients

The first example (❶) is a left-right gradient. In Firefox, I simply use `left` as a value for the *point* argument as the pair of this value is assumed to be center; also, I don't need to specify an end point. But in WebKit, I must

specify a pair of values for both the start and end points. The second example (❷) uses the same syntax, but the gradient begins on the right and ends on the left.

In the third example (❸), I set a bottom-top gradient using percentage values. Once again, in Firefox, the end point is automatically set, whereas in WebKit, I must set it manually. The fourth example (❹) uses the same syntax, but this time runs diagonally from bottom left to top right.

In the final example (❺), I use an *angle* value for Firefox. The `225deg` value sets the gradient to run from top right to bottom left. I use percentage values in WebKit to achieve the same effect.

Adding Extra color-stop Values

So far in my examples, I've used a simple gradient with only two color-stops, but you can, of course, use more. I must note, however, that this book is printed in black and white, so I'm limited by the palette that I can choose!

Each different color you add is declared in a color-stop. In Firefox, you just add more values (comma-separated) between the from-stop and to-stop, like so:

```
div { background-image: -moz-linear-gradient(left, black, white, black); }
```

The color-stops are processed in the order listed, so this example will create a gradient that goes from black to white and back to black again. The Firefox syntax will evenly distribute the color-stops along the length of the gradient unless otherwise specified. In this example, the white color-stop will be exactly halfway between the two blacks.

In WebKit, the same effect is achieved with the following syntax:

```
div { background-image: -webkit-gradient(  
  linear, left center, right center, from(black), color-stop(50%,white), to(black)  
); }
```

Notice here that I declare the color-stop using a `color-stop()` function, which requires two values: the position along the gradient where the stop should be implemented and the color. Unlike Firefox, the distribution of colors is not automatically calculated.

As before, the best way to illustrate the differences between the two syntaxes is with a demonstration; for that, I'll use the following code:

```
❶ .gradient-1 {  
  background-image: -moz-linear-gradient(left, black, white, black);  
  background-image: -webkit-gradient(linear, left center, right center,  
  from(black), color-stop(50%,white), to(black));  
}
```

```

❷ .gradient-2 {
    background-image: -moz-linear-gradient(left, black, white 75%, black);
    background-image: -webkit-gradient(linear, left center, right center,
from(black), color-stop(75%,white), to(black));
}
❸ .gradient-3 {
    background-image: -moz-linear-gradient(bottom, black, white 20px, black);
    background-image: -webkit-gradient(linear, center bottom, center top,
from(black), color-stop(0.2,white), to(black));
}
❹ .gradient-4 {
    background-image: -moz-linear-gradient(45deg, black, white, black, white,
black);
    background-image: -webkit-gradient(linear, left bottom, right top,
from(black), color-stop(25%,white), color-stop(50%,black), color-
stop(75%,white), to(black));
}

```

You can see the output in Figure 11-3.

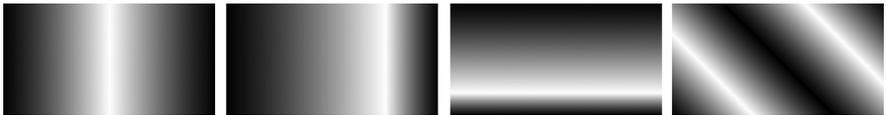


Figure 11-3: Examples of different *color-stop* values

The first example (❶) uses the values I introduced at the beginning of this section, a left-right gradient starting and ending black with a white color-stop between. As mentioned, in the WebKit syntax, I set the color-stop to be exactly halfway between the start and end, which Firefox does automatically.

The next example (❷) shows the color-stop specified to begin 75 percent of the way along the length of the gradient. You've seen how to do this in WebKit already, but in Firefox, it's slightly different: You add the percentage value after the color value, with no comma, in the `color-stop()` function.

The third example (❸) shows some different units in the `color-stop()` function. Firefox allows length units (I've used `px` here), which WebKit does not, but WebKit does allow a number from 0.0 to 1.0 instead of a percentage. As my box is 100px high, the Firefox and WebKit values are equivalent.

Finally, the fourth example (❹) has five color-stops alternating black and white. Here is where the WebKit syntax starts to become really unwieldy, requiring that I specify the `color-stop()` function for each one and that I calculate the percentage values that will distribute them equally along the length.

Radial Gradients

A *radial gradient* is the gradual transition between colors, radiating from a central point in all directions. At its simplest, a radial gradient will graduate between two colors in a circular or elliptical shape.

Radial Gradients in Firefox

Here's the syntax to implement a radial gradient in Firefox:

```
E { background-image: -moz-radial-gradient(  
    position or angle, shape or size, from-stop, color-stop, to-stop  
); }
```

The first two values, *position* and *angle*, work in the same way as their counterparts in `-moz-linear-gradient`, as do *from-stop* and *to-stop*. The two new arguments are *shape* and *size*; *shape* takes a keyword constant value of either `circle` or `ellipse` (the default is `ellipse`), and *size* accepts one of six different keyword constants, which I'll cover in due course.

The simplest way to create a radial gradient using the Firefox syntax is:

```
div { background-image: -moz-radial-gradient(white, black); }
```

The result is in a simple white-black gradient in an ellipse that extends to the farthest corner of its parent element, as you can see in Figure 11-4.

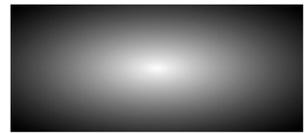


Figure 11-4: A simple two-color radial gradient in Firefox

Radial Gradients in WebKit

The WebKit syntax is substantially quite different. It looks like this:

```
E { background-image: -webkit-gradient(  
    type, inner-center, inner-radius, outer-center, outer-radius, from(from-  
stop), color-stop(color-stop), to(to-stop)  
); }
```

As with the linear gradient, you have the *type* argument, but here, I'll use the value `radial`. Next you see two pairs of arguments: *inner-center* and *inner-radius*, and *outer-center* and *outer-radius*. These arguments set the start and end points of the gradient and the distance of their radii.

Both *inner-center* and *outer-center* accept the same values, as do *inner-radius* and *outer-radius*. The values permitted for *inner-center* and *outer-center* are the same as those for *start-point* and *end-point* for a linear gradient—that is, a pair of values as keywords (`center center`) or percentages (`50% 50%`)—whereas *inner-radius* and *outer-radius* accept only an integer value that represents a length in pixels.

The `from()`, `color-stop()`, and `to()` functions are the same as those used on linear gradients, as described earlier in the chapter.

Using Radial Gradients

Because you are required to set a limit on the outer radius, WebKit radial gradient syntax doesn't allow you to create an elliptical gradient—they must all be circles. I can't, therefore, replicate the example shown in Figure 11-4. Instead, I'll show a simple radial gradient with the following syntax:

```
div { background-image:
      -webkit-gradient(radial, 50% 50%, 0, 50% 50%, 50, from(white), to(black));
}
```

The inner gradient starts at the center of the element and has a radius value of 0, and the outer gradient starts at the same place but has a radius value of 50. These values create a single gradient from white to black from the center of the element to its top and bottom edges. The container is 100px high, so the 50px radius is exactly half that.



Figure 11-5: A circular radial gradient

You can see the result in Figure 11-5.

You can replicate that exactly in Firefox with this code:

```
div { background-image: -moz-radial-gradient(contain circle, white, black); }
```

Note two new keywords here. The first is *circle*, which is a value for the *shape* argument, and simply sets the gradient to be circular instead of elliptical. The next—*contain*—is a value for the *size* argument I mentioned earlier. The *contain* value means the gradient stops at the side of the box closest to its center. You can also use the keyword *closest-side* if you prefer, as these keywords are synonymous.

The other keyword constant values for the *size* argument are: *cover*, which stops the gradient at the corner of the element farthest from its center (you can also use the *farthest-corner* keyword, as this is synonymous with *cover*); *closest-corner*, which stops the gradient at the corner closest to its center; and *farthest-side*, which stops the gradient at the side farthest from its center. Pardon the tautologous definitions, but the keywords are pleasantly obvious!

You should be able to infer from this example the key difference between the two syntaxes: WebKit requires explicitly set limits, whereas in Firefox, the limits are defined by the dimensions of the element that the gradient is applied to.

To illustrate some of the ways you can apply radial gradients, I'll show four examples using the following code:

```
❶ .gradient-1 {
  background-image: -moz-radial-gradient(circle farthest-side, black, white);
  background-image: -webkit-gradient(radial, center center, 0, center
  center, 95, from(black), to(white));
}
```

```

❷ .gradient-2 {
    background-image: -moz-radial-gradient(left, circle farthest-side, black,
    white);
    background-image: -webkit-gradient(radial, left center, 0, left center,
    190, from(black), to(white));
}
❸ .gradient-3 {
    background-image: -moz-radial-gradient(right top, circle farthest-corner,
    white, black);
    background-image: -webkit-gradient(radial, right top, 0, right top, 214.7,
    from(white), to(black));
}
❹ .gradient-4 {
    background-image: -moz-radial-gradient(80% 50%, circle closest-side,
    white, black);
    background-image: -webkit-gradient(radial, 80% 50%, 0, 80% 50%, 38,
    from(white), to(black));
}

```

You can see the output in Figure 11-6.



Figure 11-6: Examples of different values for radial gradients

The first example (❶) shows a black-white gradient that starts at the center of the box and radiates to its farthest (horizontal) side. In Firefox, I use the `farthest-side` keyword, and in WebKit, I set the value to 95px (half the width of the box).

In the next example (❷), I again set the limit of the radius to the farthest side of the box, but this time, I set the center point as the center of the left side. In WebKit, I defined the second radius value as 190px, which is the width of the box.

The third example (❸) is where WebKit's syntax really starts to take its toll. I set the gradient to start at the top-right corner of the box and the radius at the farthest corner (bottom left). In Firefox, I accomplish this with the `farthest-corner` constant, but to achieve the same effect in WebKit, I have to calculate the diagonal of the box using the formula: $\text{sqrt}(x^2 + y^2)$ —that is, the square root of the square of the height plus the square of the width, which in this example is 214.7. You really don't want to have to do this every time you use this type of gradient!

In the fourth example (❹), I've positioned the center of the gradient at 80 percent of the width and 50 percent of the height of the box and set the limit of the radius to the nearest (right) side. Again, this is simplicity itself in Firefox—I need only the `closest-side` keyword. But WebKit requires more tricky math. The box is 190px wide; the center point is, therefore, 152px from the left, so the limit of the radius must be 38px (190px – 152px).

Multiple color-stop Values

As with their linear counterparts, radial gradients accept multiple color-stop values. As before, in Firefox, you simply add the color values between the from-stop and to-stop, and in WebKit, you declare each one with a color-stop() function. You may end up with something like this:

```
E {  
    background-image: -moz-radial-gradient(circle, black, white, black);  
    background-image: -webkit-gradient(radial, 50% 50%, 0, 50% 50%, 100,  
    from(black), color-stop(50%, white),to(black));  
}
```

I'll illustrate the use of multiple color-stop values with the following code:

```
❶ .gradient-1 {  
    background-image: -moz-radial-gradient(circle farthest-side, black, white,  
    black);  
    background-image: -webkit-gradient(radial, center center, 0, center  
    center, 95, from(black), color-stop(50%, white), to(black));  
}  
❷ .gradient-2 {  
    background-image: -moz-radial-gradient(circle farthest-side, black, white  
    25%, black);  
    background-image: -webkit-gradient(radial, center center, 0, center  
    center, 95, from(black), color-stop(25%, white), to(black));  
}  
❸ .gradient-3 {  
    background-image: -moz-radial-gradient(left, circle farthest-side, white,  
    black 25%, white 75%, black);  
    background-image: -webkit-gradient(radial, left center, 0, left center,  
    190, from(white), color-stop(25%, black), color-stop(75%, white), to(black));  
}  
❹ .gradient-4 {  
    background-image: -moz-radial-gradient(40% 50%, circle closest-side,  
    white, white 25%, black 50%, white 75%, black);  
    background-image: -webkit-gradient(radial, 40% 50%, 0, 40% 50%, 50,  
    from(white), color-stop(25%, white), color-stop(50%, black), color-stop(75%,  
    white), to(black));  
}
```

All of the results can be seen in Figure 11-7.

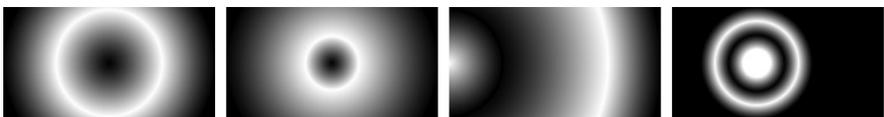


Figure 11-7: Different color-stop values for radial gradients

In the first example (❶), I created a gradient with three color-stops (black-white-black) from the center of the box to its farthest side. Remember that Firefox calculates the proportions automatically, whereas in WebKit I have to specify the length along the radius that the color-stop must occur. The

second example (❷) is similar, except I've specified the color-stop to begin 25 percent along the length of the radius. In the third example (❸), I set the gradient to begin at the left side of the box and end at the right side, with color-stops at 25 percent and 75 percent of the length.

The final example (❹) uses five colors, but by specifying both the from-stop and first color-stop to use the same color, I create the solid white circle in the center. Note that here the limit of the gradient is the long horizontal side of the box, so in WebKit, I set the outer-radius value to 50px, half the vertical height of the box.

The WebKit Advantage

One advantage that the WebKit syntax has over the Firefox syntax is the `inner-*` and `outer-*` arguments, which permit two different centers and radii to be set on the same gradient. These arguments allow for gradient patterns that the Firefox syntax can't replicate. Take the following code, for example:

```
div { background-image: -webkit-gradient(  
    radial, 45% 35%, 10, 50% 50%, 40, from(black), color-stop(75%, white),  
    color-stop(95%, black), to(white)  
);}
```

I set the inner gradient to have a center at 45 percent and 35 percent of the element and a radius of 10px and the outer gradient to have a center at 50 percent and 50 percent and a radius of 40px. Therefore, the two gradient positions are offset from each other, and the gradient has a total radius of 30px, giving the effect shown in Figure 11-8.



Figure 11-8: A radial gradient only possible with the WebKit syntax

Although unique to the WebKit syntax, this particular gradient carries the penalty of two extra arguments, which many times simply aren't needed.

Multiple Gradients

Because gradients are applied with the `background-image` property, you can use the multiple background values' syntax that's been newly introduced in CSS3 (see Chapter 8) to apply multiple gradients to an element using comma-separated values.

Here are two examples; the first uses linear gradients, the second, radial:

```
.linears {  
    background-image:  
    -moz-linear-gradient(left top, black, white, transparent),  
    -moz-linear-gradient(right top, black, white, transparent);
```

```

background-image:
  -webkit-gradient(linear, left top, right bottom, from(black), color-
stop(50%, white), to(transparent)),
  -webkit-gradient(linear, right top, left bottom, from(black), color-
stop(50%, white), to(transparent));
}
.radials {
background-image:
  -moz-radial-gradient(20% 50%, circle contain, white, black 95%,
transparent),
  -moz-radial-gradient(50% 50%, circle contain, white, black 95%,
transparent),
  -moz-radial-gradient(80% 50%, circle contain, white, black 95%,
transparent);
background-image:
  -webkit-gradient(radial, 20% 50%, 0, 20% 50%, 50, from(white), color-
stop(95%, black), to(transparent)),
  -webkit-gradient(radial, 50% 50%, 0, 50% 50%, 50, from(white), color-
stop(95%, black), to(transparent)),
  -webkit-gradient(radial, 80% 50%, 0, 80% 50%, 50, from(white), color-
stop(95%, black), to(transparent));
}

```

Both of these examples are shown in Figure 11-9.



Figure 11-9: Multiple gradient background values

The first example shows two linear gradients, one from top-left to bottom-right, the other from top-right to bottom-left. The to-stop has a value of transparent to allow the second gradient to show through below it—remember, if you don't set transparency, then the gradient will fill the rest of the box and the layer below it will be hidden.

The second example shows three radial gradients with a radius of 50px each. Again, the to-stop has a value of transparent to allow the layers below to show through.

Repeating Gradients in Firefox

One limitation of the current gradient syntaxes is that they become more and more unwieldy as you specify extra color-stops, meaning they require a lot of repetitive code if you want to do anything more than a simple gradient. The Firefox developers have proposed a pair of functions that go some way toward solving this problem by repeating the gradient to fill the box. These are currently proprietary functions that have not been accepted into the Image Values Module, so you must use the `-moz-` prefix.

Repeating Linear Gradients

Repeating a linear gradient is done with the `-moz-repeating-linear-gradient` function, which accepts the same fundamental set of values as `-moz-linear-gradient`:

```
E { background-image: -moz-repeating-linear-gradient(<point> or <angle>, <from-stop>, <color-stop(s)>, <to-stop>); }
```

The difference between the two is that with `-moz-repeating-linear-gradient`, a length or percentage value is required for the to-stop. For example:

```
div { background-image: -moz-repeating-linear-gradient(white, black 25%); }
```

This value sets the point at which the gradient should end and then start repeating. This example creates a top-bottom gradient (remember, this is the default) between white and black that covers 25 percent of the height of the box, meaning it would repeat four times.

This is best illustrated with some examples using different values, which I'll then explain. Here's the code I've used for the examples:

```
❶ .gradient-1 {  
    background-image: -moz-repeating-linear-gradient(white, black 25%);  
}  
❷ .gradient-2 {  
    background-image: -moz-repeating-linear-gradient(left, black, white, black 25%);  
}  
❸ .gradient-3 {  
    background-image: -moz-repeating-linear-gradient(45deg, black, white 2px, black 10px);  
}  
❹ .gradient-4 {  
    background-image: -moz-repeating-linear-gradient(315deg, black, black 2px, white 2px, white 4px);  
}
```

You can see the output in Figure 11-10.

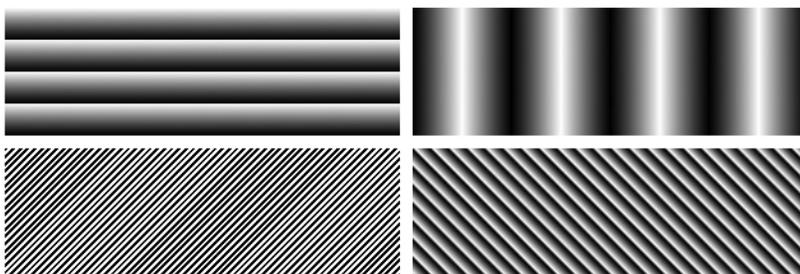


Figure 11-10: Repeating linear gradients in Firefox

Moving clockwise from top left, the first example (❶) uses the code with which I introduced this section: top-bottom, two colors, repeated four times. Next, I introduce an extra color-stop and change the point so the gradient goes from left-right (❷). Here the gradient again covers 25 percent of the element, but this time the gradient goes black-white-black and is evenly distributed.

For the third example (❸), I use an angle value of 45deg so the gradient is diagonal, and I use px units for the color-stops. Again the gradients are black-white-black but this time distributed unequally, so the black-white covers 2px, whereas the white-black covers 8px.

In the final example (❹), I use four color-stops: black-black over 2px and then white-white over 2px. The low length values don't allow any gradual change between the two colors, creating the hard diagonal lines you see here.

Repeating Radial Gradients

Accompanying the `-moz-radial-gradient` function in Firefox is `-moz-repeating-radial-gradient`, which repeats the values supplied until its specified limit is reached. It works in the same way as `-moz-repeating-linear-gradient`, requiring a length value for to-stop, such as:

```
E { background-image:
    -moz-repeating-radial-gradient(circle, black, white 20px);
}
```

This example creates a black-white gradient that is repeated every 20px. You can see this in action along with some further demonstrations in this code:

```
❶ .gradient-1 {
    background-image: -moz-repeating-radial-gradient(circle farthest-side,
    black, white 20px);
}
❷ .gradient-2 {
    background-image: -moz-repeating-radial-gradient(right top, circle cover,
    black, white 10%, black 15%);
}
❸ .gradient-3 {
    background-image: -moz-repeating-radial-gradient(left, circle cover,
    white, white 10px, black 15px);
}
❹ .gradient-4 {
    background-image: -moz-repeating-radial-gradient(circle cover, white,
    black 1px, white 2px);
}
```

The results are displayed in Figure 11-11.

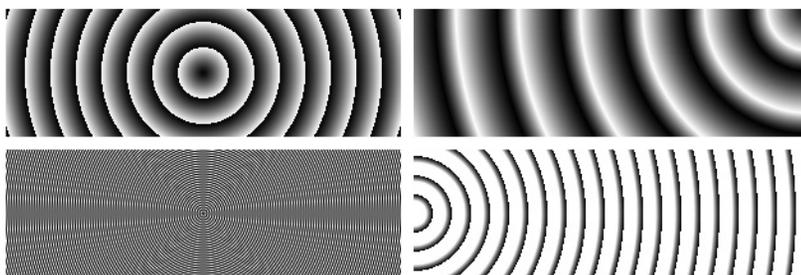


Figure 11-11: Repeating radial gradients in Firefox

Once again moving clockwise from top left, the first example (❶) is the one I used in the introduction to this section, a black-white circular gradient repeated every 20px. The second example (❷) radiates from the top-right corner and passes through three color-stops over 15 percent of the box width—the limit is set by the cover keyword constant, meaning it goes to the farthest (bottom-left) corner. In the third example (❸), I set the center of the gradient to the left side of the box and the limit to the farthest corner, using a white-white (solid) gradient for 10px and then white-black for 5px. In the final example (❹), I seem to have created a work of art! I set a repeating white-black-white gradient over a very low radius of 2px, which has created the interference pattern you see here. I didn't do it intentionally, but the result certainly is interesting!

Summary

The WebKit syntax allows for fine control over your gradients, but that control comes at a cost: complexity. Although the Firefox syntax doesn't allow you to do everything that you can with WebKit, you'll find it much, much simpler to apply a gradient to an element.

Currently, adding gradients to your pages with CSS may seem like a fairly painful process, but really that's only because W3C adoption of the method is quite recent, and actual browser implementation is in a transitional phase requiring two quite dissimilar syntaxes in order to achieve the desired effects.

Barring extraordinary circumstances, the current WebKit syntax is not going to make it into the final draft of the specification, so you may feel you'd be better off not using it. As long as you have provided a fallback image (or even a plain color)—which you should do for other browsers anyway—you can feel free to use just the Firefox implementation for now:

```
div {  
  background-image: url('gradient.png');  
  background-image: -moz-linear-gradient(black,white);  
}
```

Now that we've finished looking at ways to decorate your page elements, I'm going to move on to cover a whole new field of expertise: transforming and animating page elements.

Gradients: Browser Support

	WebKit	Firefox	Opera	IE
Gradients	Yes (with prefix; incorrect syntax)	Yes (with prefix)	No	No
Repeating Gradients	No	Yes (with prefix)	No	No

12

2D TRANSFORMATIONS



Because of the way HTML works, with all of its elements composed of rectangular blocks and right-angled corners, web pages have traditionally appeared mostly boxy, with lots of straight horizontal and vertical lines, and the only way to provide any variation to this rule was to use images. But back in 2008, the WebKit team proposed a new module that allows elements to be rotated, resized, skewed, and generally messed around with. This module has since been adopted by the W3C and formalized as the 2D Transformations Module (<http://www.w3.org/TR/css3-2d-transforms/>).

The module's components are adapted from functions in the *Scalable Vector Graphics (SVG)* language, which is a specification for drawing two-dimensional images. SVG is supported by most modern browsers, so Firefox and Opera were quick to implement 2D Transformations in their own products. The Release Candidate of IE9 also has an implementation, so it should be in the final release of that browser. All this means you can start practicing with these new features right away.

One thing to note before I introduce the syntax: By the end of this chapter, I'll have introduced some fairly complex functions, so you might want to brush up on your trigonometry. Are you nervous? Don't worry; I'll try to make it as painless as possible.

The transform Property

A range of different transformations can be applied, but all are declared as functions in the `transform` property. Here's the basic syntax:

```
E { transform: function(value); }
```

A number of possible functions are available; I'll explore each in turn throughout the rest of this chapter. Each function takes either a single value or a comma-separated list of values. I'll also explain what this means when I discuss each function individually.

As I mentioned in the beginning of the chapter, Firefox (3.5+), Opera (10.5+), IE9 (Release Candidate), and WebKit all developed implementations of the `transform` property, each with its proprietary prefix, which means that to use this property currently, you have to specify it four times:

```
E {  
  -moz-transform: function(value); /* Firefox */  
  -ms-transform: function(value); /* Internet Explorer */  
  -o-transform: function(value); /* Opera */  
  -webkit-transform: function(value); /* WebKit */  
}
```

Ordinarily, I would recommend adding the nonprefixed property after each of the browser-specific ones, so future browser version releases that implement the nonprefixed property are accounted for, like so:

```
E {  
  -moz-transform: function(value); /* Firefox */  
  -ms-transform: function(value); /* Internet Explorer */  
  -o-transform: function(value); /* Opera */  
  -webkit-transform: function(value); /* WebKit */  
  transform: function(value); /* Future-proofing */  
}
```

Some browsers, however, have implemented `transform` very slightly differently, and because the module is still at Working Draft status, the syntax is subject to change. For that reason, I caution against using the future-proofing method of including the nonprefixed property after the prefixed one, just in case the final syntax differs from the current one. In my examples, I'll follow my custom of only using the nonprefixed rule, but remember, when creating your own pages, you have to specify all four.

rotate

Probably the simplest of all the functions is rotate, which does what it sounds like it should do: It rotates the element around a set point. Here's the syntax:

```
E { transform: rotate(value); }
```

The *value* here is a single angle value just like you used with the CSS Gradients introduced in Chapter 11. And, like in that chapter, I'm sticking with the commonly understood degrees (deg) unit for my examples. Note that you can also use negative values here: for example, -90 degrees is equivalent to 270 degrees.

To show you rotate in action, I'm going to rotate an h2 element by -25 degrees (or 335 degrees), using this rule:

```
h2 { transform: rotate(-25deg); }
```

You can see how this displays in Figure 12-1.



Figure 12-1: An h2 element rotated by 10 degrees

In Figure 12-1, I positioned another h2 element with a light gray background in the same place as the first one so you can compare the rotated element with the default (nonrotated) element. I'll do the same for most examples in this chapter.

Position in Document Flow

Once an element has been transformed, it acts as if it had `position: relative` applied to it; that is, the element is almost like two elements. The original, pretransformation element retains its place in the document flow, so all subsequent elements are affected by it and its margin and padding. The transformed element does not affect the page layout but sits in a new layer above the rest of the page, which means the new element can cover subsequent elements.

In the next example, I'll rotate the h2 element again, but this time flow some text under it so you can see the effect of the transformation. Here's the code to rotate the element:

```
h2 { transform: rotate(-15deg); }
```

Figure 12-2 shows the results of this transformation.

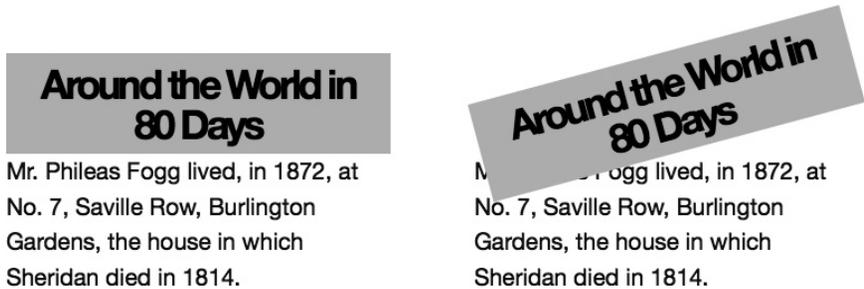


Figure 12-2: The effect of a transformed element on the document flow

Both examples are identical, except for the rotate transformation. You can clearly see the rotated element covers the text under it, which immediately follows the nontransformed element in the layout. This same rule applies to any element that has the transform property applied to it, so be aware of this going forward.

transform-origin

The *origin* of a transformation is the point on an element about which that transformation happens. This concept is easiest to illustrate using rotate, although you can apply it to any of the transformation functions introduced in this chapter.

In the case of rotate, you can visualize the origin by imagining you have a piece of paper (the element) and a pin (the origin of that element). If you use the pin to stick the paper to a flat surface, you can rotate the paper around the pin. By sticking the pin in different places on the paper, you can change how the rotation happens—if the pin is stuck in the center of the paper, the rotation has a short radius, and the paper on either side of the pin moves equally; if the pin is placed at one edge, the whole length of the paper rotates around it.

The default point of origin in the CSS transform property is the *absolute* (that is, horizontal and vertical) center. You can change this using the transform-origin property:

```
E { transform-origin: value(s); }
```

The *value* for this property is either one or two length or keyword values. Lengths can be any accepted CSS unit (em, px, etc.) or a percentage. The keywords are left, right, top, bottom, and center. If two values are supplied, the first sets the horizontal point and the second sets the vertical; if only one is supplied, that point sets the horizontal, with the vertical presumed to be center (or 50 percent).

If you want to change the point of origin to the top-left corner, you can use either of the following:

```
E { transform-origin: 0 0; }  
E { transform-origin: left top; }
```

And if you want the point of origin to be the bottom-right corner, you can use these values (let's say the element has a height of 50px and a width of 200px):

```
E { transform-origin: 200px 50px; }  
E { transform-origin: 100% 100%; }  
E { transform-origin: right bottom; }
```

Let me demonstrate the effects of changing the origin of transformation. This example shows three identical elements with the same transform property applied to each but with a different transform-origin value:

```
h2 { transform: rotate(-10deg); }  
h2.example-1 { transform-origin: left center; }  
h2.example-2 { transform-origin: 100% 50%; }
```

You can see the effects on the three elements in Figure 12-3.



Figure 12-3: Different transform-origin values on a rotated element

The first element has the default values of center center, so the element rotates around the absolute center. The second element has values of left center, so the element rotates around the vertical center of the left-hand side. And the third element has values of 100% 50%, so the element rotates around the vertical center of the right-hand side.

translate

The next function we'll look at is `translate`, which moves the element from its default position. Three functions are actually involved: `translateX`, `translateY`, and `translate`:

```
E {  
  transform: translateX(value);  
  transform: translateY(value);  
}  
E { transform: translate(translateX,translateY); }
```

The first two functions, `translateX` and `translateY`, move the element along an axis—if you need a refresher, refer to the explanation of axes in Chapter 6—for the length that you specify. You can use any length units or percentage values here, so, for example, you could have:

```
E {  
  transform: translateX(20px);  
  transform: translateY(15%);  
}
```

This code would move the element 20px to the right (along the *x*-axis), and 15 percent of its own height down (along the *y*-axis). You can also use negative values, which would move the element in the opposite direction along the axis—that is, up or to the left.

The next function, `translate`, is shorthand for `translateX` and `translateY`. You could use it with the previous example values like so:

```
E { transform: translate(20px,15%); }
```

Here are two examples using `translate`:

```
h2.translate-1 { transform: translate(20px,20px); }  
h2.translate-2 { transform: translate(20px,-20px); }
```

You can see the results of this code displayed in Figure 12-4, again with elements in a lighter gray showing the original position.



Figure 12-4: Elements showing effects of different values in the `translate` function

The first example uses a pair of values of 20px, so the element is offset from the original by 20px both horizontally and vertically. The second example has a negative second value (-20px), so the element is moved in the opposite direction along the y-axis, so it's offset above the original.

It's permissible to use only one value with the translate shorthand, like so:

```
E { transform: translate(20px); }
```

If this is the case, the value provided will be presumed to be the translateX value, and a value of 0 (zero) will be used for translateY. That being the case, both of the following declarations are the same:

```
E { transform: translate(20px,0); }  
E { transform: translate(20px); }
```

You might think translate seems very similar to using relative positioning and the left and top properties, but remember that the transformed element retains its position and only *appears* to have moved; the image of the element is transformed, not the element itself.

skew

The skew function allows you to alter the angle of the horizontal or vertical axis (or both axes) of an element. As with translate, each axis has an individual function, and a shorthand function is available to indicate both:

```
E {  
  transform: skewX(value);  
  transform: skewY(value);  
}  
E { transform: skew(skewX,skewY); }
```

The values for the skew functions are angle values (I'll use degrees in my examples). Negative values are permitted, and the skew shorthand can take either one or two values—as before, if only one is specified, then the value is presumed to be skewX, and skewY defaults to 0.

Let me show you how skew works by providing three demonstrations, using this code:

```
h2.transform-1 { transform: skewX(45deg); }  
h2.transform-2 { transform: skewY(10deg); }  
h2.transform-3 { transform: skew(-45deg,5deg); }
```

These demonstrations are illustrated in Figure 12-5.



Figure 12-5: Elements transformed by different values in the skew function

In the first example, the element is skewed by 45 degrees along its x -axis, causing the vertical edges to slope diagonally. In the second example, the skew is by 10 degrees on the y -axis, so the horizontal edges slope diagonally while the vertical edges remain unchanged. The final example shows the effect of two values being applied using the shorthand function. The values are -45 degrees on the x -axis and 5 degrees on the y -axis, so the element is sloped on both axes.

By looking at that last example, you can see replicating the rotate function using skew is possible. To do this, the angle that you want to rotate the element by is given as a value to `scaleX` and the inverse value to `scaleY`; that is, if `scaleX` is 10 degrees, then `scaleY` should be -10 degrees, and vice versa. Therefore, the two functions in this code example perform the same job:

```
E {
transform: rotate(15deg);
transform: skew(15deg,-15deg);
}
```

You'll find this useful to know when I introduce the `matrix` function later in this chapter.

scale

You can make an element larger or smaller than the original by using the `scale` function. Once again, you have functions for the horizontal and vertical values and a shorthand function:

```
E {
  transform: scaleX(value);
  transform: scaleY(value);
}
E { transform: scale(scaleX,scaleY); }
```

The values for `scaleX` and `scaleY` are unitless numbers, which give a size ratio. The default size is 1; 2 is twice the default, 0.5 is half the default, and so on. You can also use negative numbers—I'll explain the effect of negative numbers shortly.

To make an element double its original size on both axes, you would use:

```
E {  
  transform: scaleX(2);  
  transform: scaleY(2);  
}
```

Of course, you could also use the shorthand property, `scale`. Note, however, that the `scale` function works differently from the other shorthand properties you've seen so far in this chapter, in that if only one value is provided, the other is presumed to be identical. Using the previous example, you could opt instead for the shorthand:

```
E { transform: scale(2); }
```

I'll demonstrate `scale` in action with a few examples. Here's the code I'll use:

```
h2.transform-1 { transform: scale(0.5); }  
h2.transform-2 { transform: scaleX(0.5); }  
h2.transform-3 { transform: scale(1,-1); }
```

The results are shown in Figure 12-6.



Figure 12-6: The effects of different values in the `scale` function

The first example has a scale value of 0.5, so the transformed element is half the size of the original—remember I specified only one value in the shorthand, so the other is presumed to be equal. In the second example, I used 0.5 as a value but this time for the `scaleX` function, meaning the transformed element is the same height as the original but only half the width.

In the final example, I supplied two values to the `scale` shorthand: The first is 1 (one), which sets the horizontal size to be the same as the original, but the second is -1 (negative one). Using a negative value has the effect of flipping the element vertically, creating a “reflection” of the original element at the same scale.

NOTE *WebKit browsers have another way to do this as well, with the `box-reflect` property, which I'll introduce later in "Reflections with WebKit" on page 159.*

Multiple Transformations

You can apply multiple transformations to a single element by simply listing functions, space-separated, in the transform property:

```
E { transform: function(value) function(value); }
```

So you could, for example, rotate, scale, and translate an element with code like this:

```
h2 { transform: rotate(-40deg) scale(0.75) translate(-46%,-400%); }
```

You can see how this code looks in a real-world demonstration shown in Figure 12-7.

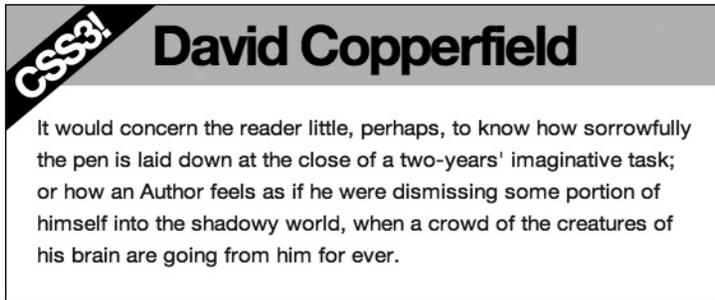


Figure 12-7: Multiple functions applied to the h2 element to create a "ribbon" effect

The h2 element has been transformed with multiple functions and sits across the top-left corner of its parent element in a ribbon effect that's fairly popular on the Web but, until now, had to be done with images.

Transforming Elements with Matrices

You can use one final transformation function to apply multiple values to an element; this function is called `matrix`. Rather than explaining the theory behind transformation matrices (which is quite complex and could easily be a chapter of its own), I'll try to keep the explanation as simple as possible and just give you the practical basics. If you really want to dig into the theory, I suggest you read the W3C's explanation at <http://www.w3.org/TR/SVG/coords.html#TransformMatrixDefined>.

The `matrix` function accepts six values. By combining them, you can replicate the functions introduced already in this chapter. Here's the syntax:

```
E { transform: matrix(a,b,c,d,X,Y); }
```

All of the default values are 0 (zero), and they behave slightly differently depending on which values are supplied—I'll explain what this means as I go along. I said that you can perform all of the functions introduced so far in this chapter with `matrix`, but the process is not quite that simple—you need to know some trigonometry first.

Before getting to the hard stuff, I'll start with something simple that doesn't require any trig: scaling. If you want to scale an element, you can use `a` and `d` to equate to `scaleX` and `scaleY` and set values accordingly, leaving `b` and `c` at 0. Therefore, to double an element's size, you would use:

```
E { transform: matrix(2,0,0,2,0,0); }
```

You can also translate an element with `matrix` by providing horizontal and vertical offset values to `X` and `Y` (respectively). Firefox implements this slightly differently than other browsers by requiring length units for the values, whereas WebKit, Opera, and IE9 accept only unitless numbers, which represent pixel values. That being the case, for this next example I'm going to list all of the different prefixed properties:

```
E {  
  -moz-transform: matrix(2,0,0,2,15px,15px);  
  -ms-transform: matrix(2,0,0,2,15,15);  
  -o-transform: matrix(2,0,0,2,15,15);  
  -webkit-transform: matrix(2,0,0,2,15,15);  
}
```

The result of this code would be to double an element's size and offset it by 15px both vertically and horizontally.

Mozilla has suggested the specification be modified to support their proposed change, as the current spec means only pixel values can be used for matrix transformations. This is the key difference among the different browsers' implementations. In the examples in the rest of this chapter I'll use unitless values, because they are more common.

If you want to skew an element, well, this is where it becomes a lot more complex—here's where I need to introduce the trigonometric functions. You can read a full explanation of these functions on Wikipedia (http://en.wikipedia.org/wiki/Trigonometric_functions#Sine.2C_cosine_and_tangent), but here's a quick and dirty summary: The trigonometric functions are ratio values used to calculate angles in a triangle.

The first trigonometric function I'll use is *tan* (*tangent*), which is required to skew an element along the *x*- or *y*-axis. Referring to the original matrix syntax, the *x*-axis is supplied as a value to *b* and the *y* as a value to *c*. Here's the syntax for each:

```
E { transform: matrix(1,tan(angle),0,1, X,Y); } /* X Axis */
E { transform: matrix(1,0,tan(angle),1, X,Y); } /* Y Axis */
```

The *angle* here refers to the degrees (counterclockwise) of the angle you want to skew by. If you want to skew an element by 15 degrees, the value you're looking for is the tangent of 15. So whip out your scientific calculator—if you don't own one, your operating system's calculator should have a scientific mode—and get the result that $\tan(15) = 0.27$. This result is what you provide to the *matrix* function. For example, if you want the skew to be along the *x*-axis, the syntax would be:

```
E { transform: matrix(1,0.27,0,1,0,0); }
```

NOTE *As I'm using degrees in my examples, make sure your calculator's trigonometric type is set to degrees if you want to follow along. If you'd prefer working in radians or gradians, all of these examples can be updated accordingly.*

As mentioned previously, skew can also be used to rotate an element—and you can do the same with *matrix*. This time you have to make use of the *sin* (*sine*) and *cos* (*cosine*) trigonometric functions. To rotate an element, the *matrix* syntax is:

```
E { transform: matrix(cos(angle),sin(angle),-sin(angle),cos(angle),X,Y); }
```

Note that *a* and *d* take the same value, and *b* and *c* take inverse values (if *b* is a positive value, *c* is the negative of the same value, and vice versa). Once again, *angle* refers to the degrees of the angle you want to rotate the element by. To rotate by 60 degrees, you would go back to your scientific calculator and calculate the *cos* and *sin* of 60. My calculator tells me that $\cos(60) = 0.5$ and $\sin(60) = 0.87$, so the required code would be:

```
E { transform: matrix(0.5,0.87,-0.87,0.5,0,0); }
```

Now let's look at a few examples. Here's the code I'll use:

```
h2.transform-1 { transform: matrix(1,0,0,-1,0,-24); }
h2.transform-2 { transform: matrix(1,0,1,1,18,-24); }
h2.transform-3 { transform: matrix(0.98,-0.17,0.17,0.98,0,0); }
```

The output is shown in Figure 12-8.



Figure 12-8: Examples of transformations made with the *matrix* function

In the first example, I’ve flipped the element vertically (as I did earlier using *scale* in Figure 12-6) and translated it by -24px along the *y*-axis to sit neatly under the original element. In the next example, I’ve skewed the element by 45 degrees along the *y*-axis (after calculating that $\tan(45) = 1$) and translated it along both axes. The final example shows the element rotated by 10 degrees; the values are the results of the calculations I showed you previously: $\cos(10) = 0.98$ and $\sin(10) = 0.17$. As mentioned, the *sin* value is negative in position *b* and positive in position *c*, which makes the rotation uniform.

I know this is all quite complex, but hopefully I’ve been able to simplify it enough for you to understand, without making it seem so simple you can’t see the scope for great flexibility—as long as you keep your scientific calculator with you at all times! And if this does seem overly complex, and you’re having trouble understanding it, remember you can perform all of these transformations using the individual functions, so you can happily forget about *matrix* and trigonometry, too, if you so desire.

Reflections with WebKit

One of the common design tropes of “Web 2.0” was the reflected image—that is, an image made to appear as if it were reflected in a shiny surface. Earlier in the chapter, I demonstrated how you can use *scale* to flip and reflect an element, but this requires two separate page elements: the original and the reflection.

To address this issue, WebKit introduces a new property that removes the need for that extra element: *-webkit-box-reflect*. The syntax looks like this:

```
E { -webkit-box-reflect: direction offset mask-box-image; }
```

The first value, *direction*, is a keyword that sets where the reflection should appear in relation to the element: *above*, *below*, *left*, or *right*. Next is *offset*, which is a length value that sets the distance between the element and the reflection (the default is 0). The final value is *mask-box-image*, which is an optional value that allows you to set an image to use as a mask.

Let's put aside the *mask-box-image* value briefly, and instead I'll show you an example of the simplest possible reflection:

```
h2 { -webkit-box-reflect: below; }
```

Here, I've set the reflection to appear below the element; remember, *offset* defaults to 0 and *mask-box-image* is optional, so *direction* is the only required value. You can see how this appears in Figure 12-9.

This code creates an exact mirror of the image, but in order to get a shiny Web 2.0 reflection you need to create the illusion that the reflection fades out as it gets farther away from the original element. You do this with the *mask-box-image* value—or, rather, series of values. *mask-box-image* uses the same syntax as you saw in the section on masks in Chapter 8, which is the also the same as *border-image* from Chapter 9:



Figure 12-9: A simple reflection below the element

```
E { -webkit-box-reflect: direction offset source slice repeat; }
```

As with *border-image*, *source* is the URL of an image file (you can also use the gradient syntax from Chapter 11 here), *slice* is an optional series of length values used to define the area of the image that will be used, and *repeat* is a keyword value that sets how the image is repeated to fill the dimensions of the reflected element; this value is also optional, and the default value of *stretch* will be used if *repeat* isn't present.

Let me show you two very simple examples that use only the source value:

```
h2.transform-1 { -webkit-box-reflect:  
  below 0 -webkit-gradient(linear,50% 0,50% 100%,from(transparent),to(white));  
}  
h2.transform-2 { -webkit-box-reflect: url('cloud.png'); }
```

You can see the output of this in Figure 12-10.



Figure 12-10: Using a gradient (left) and image (right) as a mask

In the first example, I used a gradient fill vertically from transparent to white, which provides the shiny reflection effect. In the second, I used the image *cloud.png* to act as a mask on the reflection.

Note that the reflection behaves in the same way as the other transformation effects in this chapter: It occupies no position in the layout flow of the page but sits in a layer above the main body content and, therefore, will overlay subsequent elements.

Summary

This chapter has introduced the most complex property so far: the `matrix` function. Although one might be tempted to say that if you want to perform complex operations, you have to deal with complex syntax, I think the other functions in this chapter do a good job of translating that complexity into something simple. Just remember, easier options are always available if you get fed up with `matrix`.

This chapter has also introduced some of the most revolutionary properties so far—just a few years ago the notion that you could rotate, skew, and scale elements was just a pipe dream, but today you can do just that. And if you think what you’ve seen in this chapter is impressive, wait until the next one—I’m going to show how you can introduce subtle (or not so subtle) animation effects, without using JavaScript.

2D Transformations: Browser Support

	WebKit	Firefox	Opera	IE
2D Transformations	Yes (with prefix)	Yes (with prefix)	Yes (with prefix)	No (expected in IE9, with prefix)
<code>box-reflect</code>	Yes (with prefix)	No	No	No

13

TRANSITIONS AND ANIMATIONS



We think of web pages as having three layers: content (HTML), presentation (CSS), and behavior (JavaScript), and it's generally understood that these layers should all be kept absolutely separate—we don't use presentational or behavioral rules in the content layer (in other words, no CSS or JavaScript inline in the markup). This separation is not quite as clear cut as it seems, however—for a start, CSS has always had some behavioral aspects (the `:hover` pseudo-class being a prime example).

This line between presentation and behavior was blurred even further when the WebKit developers introduced two new behavioral modules that have been adopted as CSS3 components: Transitions and Animations. These allow for the animation of element properties, adding movement to otherwise static pages even when JavaScript is not available.

Some have argued—and perhaps still do—about whether the Transition and Animation modules should be included in CSS, as they fall firmly in the behavioral layer. But as the W3C has decided to offer them for discussion as an “official” part of CSS3, we no longer need to debate the ethics of them—we can just have fun with them, instead!

The difference between Transitions and Animations is that the former is *implicit* and the latter is *declared*. That means Transitions only take effect when the property they are applied to changes value, whereas Animations are explicitly executed when applied to an element.

I'll start this chapter with a look at Transitions, as they are the simpler of the two modules; however, both have a lot of syntax in common, so much of what you learn from one can be directly applied to the other.

Transitions

There are no in-between states in CSS2: When the value of a property changes, the change is abrupt. Consider an element with a width of 100px, which changes to a width of 200px when you hover your mouse over it. You'll notice the element does not progress smoothly between the two states; the element jumps between them. CSS3 provides options to change this with the introduction of the Transitions Module (<http://www.w3.org/TR/css3-transitions/>). In CSS, a *transition* is an animation that moves a property between two states.

As I mentioned in the introduction to this chapter, transitions are an *implicit* animation, which means they are triggered only when a new value is set for a CSS property. For a transition to occur, four conditions must be in place: an initial value, an end value, the transition itself, and a trigger.

Here's an example of those four conditions in a very simple transition:

```
div {
  background-color: black;
  transition: background-color 2s;
}
div:hover { background-color: silver; }
```

The `div` element provides the initial value (`background-color: black`) and the transition (`background-color 2s`). Don't worry about the syntax just yet; I'll explain everything in due course. The trigger is the `:hover` pseudo-class, which also provides the end value (`silver`) for the `background-color` property.

So here we have a `div` element with a black background that, when the mouse is passed over it, transitions smoothly to silver. All transitions act in reverse when the trigger is no longer active, so when the mouse is moved off of the `div`, the background smoothly transitions back to black.

Now that you have a general overview of how it works, I'll explore each of the transition properties in turn. Transitions are implemented in WebKit browsers, Opera (from version 10.6), and in pre-release test builds of Firefox (and are currently planned for implementation in version 4). All of the following properties should be prefixed with `-moz-`, `-o-`, and `-webkit-` in your pages; as always, I will refer to the nonprefixed properties in the examples.

Property

The first new property, `transition-property`, specifies which property (or properties) of an element will be animated. Here's the syntax:

```
E { transition-property: keyword; }
```

An acceptable value for *keyword* would be either the keywords `all` or `none`, or a valid CSS property. The default value is `all`, which means every valid property will be animated. I stress *valid* CSS property because not every property can be transitioned; the specification has a full list of the ones that can at <http://www.w3.org/TR/css3-transitions/#properties-from-css->.

Here's an example of `transition-property`:

```
h1 {  
    font-size: 150%;  
    transition-property: font-size;  
}
```

This code sets an initial value of 150% on the `font-size` property and declares this is the property that will be transitioned when the (not yet specified) trigger is activated. Note that I will add properties to this example throughout the rest of this section before showing the completed example in action at the end.

Duration

The next property is `transition-duration`, which defines the length of time that the transition takes to complete. The syntax for this is:

```
E { transition-duration: time; }
```

The *time* value is a number with a unit of *ms* (*milliseconds*) or *s* (*seconds*). Since 1,000 milliseconds equals 1 second, a value of 1.25s is the same as 1250ms. The default value is 0 (zero), meaning this property is the only one required to create a transition. A transition can occur if you declare a `transition-duration` without a `transition-property` (as that defaults to `all`, so all valid properties will animate) but not vice versa.

To make the example transition from the first section happen over a period of two seconds, you would add this code:

```
h1 {  
    font-size: 150%;  
    transition-property: font-size;  
    transition-duration: 2s;  
}
```

Note that although you can supply negative values here, they will be interpreted as 0.

Timing Function

To control the manner in which an element transitions between states we use the `transition-timing-function` property. This allows for variations in speed along the duration of the transition, which gives you control over the animation's pace. This property has two different value types: a keyword or the cubic-bezier function. I'll discuss cubic-bezier in detail later in this section, as it's a little complex; to begin, I'll focus on the keywords.

Timing Function Keywords

The syntax of the `transition-timing-function` property when used with a keyword is quite straightforward:

```
E { transition-timing-function: keyword; }
```

The possible values for *keyword* are `ease`, `linear`, `ease-in`, `ease-out`, and `ease-in-out`. The default value is `ease`, which starts slowly, accelerates quickly, and slows down again at the end. The `linear` value progresses steadily from the start of the transition to the end, with no variation in speed. With the `ease-in` value, the animation begins slowly and then speeds up toward the end, and the `ease-out` value acts in reverse. Finally, `ease-in-out` starts slowly, speeds up through the middle, and then slows down again at the end, similar to—but less dramatic than—the `ease` value.

With that explained, let's add a simple timing function to the example transition:

```
h1 {  
  font-size: 150%;  
  transition-property: font-size;  
  transition-duration: 2s;  
  transition-timing-function: ease-out;  
}
```

The Cubic Bézier Curve

One of the possible values for the `transition-timing-function` is the cubic-bezier function. In case you're not familiar with cubic Bézier curves—and indeed, why would you be?—allow me to explain. First, here's the syntax:

```
E { transition-timing-function: cubic-bezier(x1, y1, x2, y2); }
```

A cubic Bézier curve is plotted over four points on a grid that goes from 0 to 1 along both axes. The four points are known as p_0 , p_1 , p_2 , and p_3 . They define curvature and are plotted with pairs of (x, y) coordinates, where the first (p_0) is always at $(0, 0)$ and the last (p_3) is always at $(1, 1)$. The other two points are defined in the function: (x_1, y_1) and (x_2, y_2) . An example, shown in Figure 13-1, illustrates this best.

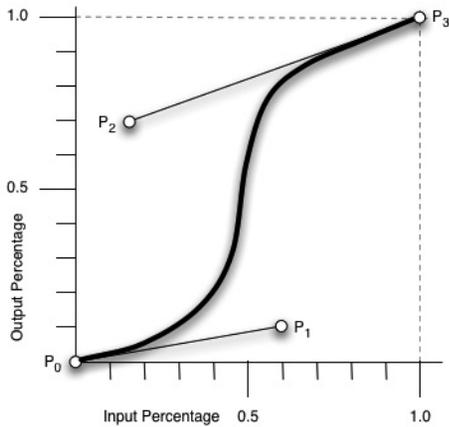


Figure 13-1: An example of a cubic Bézier curve

Figure 13-1 shows the four points mapped onto the grid to create a Bézier curve. The coordinates of each point are shown in Table 13-1:

Table 13-1: The Coordinate Points Used to Plot a Bézier Curve

Point	Coordinates (x, y)
p0	(0, 0)
p1	(0.6, 0.1)
p2	(0.15, 0.8)
p3	(1, 1)

You would use the following CSS to represent this curve (remember, you don't need to define p0 and p3 because they will always have the same values):

```
E { transition-timing-function: cubic-bezier(0.6, 0.1, 0.15, 0.8); }
```

A linear animation progresses in a straight line from (0, 0) to (1, 1), but this example animation follows the progression of the curve toward the final point over the set duration. If you imagine the duration to be 1 second, you can see the speed gradually increases at the start, between 0 and (roughly) 0.5 seconds, then increases sharply to about 0.7 seconds, and then assumes a slower rate until the end of the animation.

All of the transition-timing-function keywords described earlier are produced with cubic Bézier curves. Table 13-2 shows each of the keywords and their corresponding values for the cubic-bezier function.

As with the transformation matrices I introduced in the previous chapter, cubic-bezier functions can be quite daunting if you're not used to math. But don't worry—you can always use the keyword values, which will be more than sufficient in most cases.

Table 13-2: Comparing transition-timing-function Keywords with Their Equivalent Cubic Bézier Values

Keyword Value	Cubic Bézier Value
ease	0.25, 0.1, 0.25, 1
linear	0, 0, 1, 1
ease-in	0.42, 0, 1, 1
ease-out	0, 0, 0.58, 1
ease-in-out	0.42, 0, 0.58, 1

If you want to experiment with cubic Bézier curves there's a tool online at <http://www.netzgesta.de/dev/cubic-bezier-timing-function.html> which graphically displays the results of different combinations of coordinates.

Delay

The final property in the transition-* family is transition-delay, which sets the time when the transition starts. Here's the syntax:

```
E { transition-delay: time; }
```

As with transition-duration, the *time* value is a number with a unit of either ms or s. The default value is 0 (zero), meaning the transition happens as soon as the trigger is . . . well, triggered. Any other positive value starts the transition after the specified length of time has passed.

For example, if you wanted to set a delay of a quarter of a second at the start of the example transition, here's the code you would use:

```
h1 {  
  font-size: 150%;  
  transition-property: font-size;  
  transition-duration: 2s;  
  transition-timing-function: ease-out;  
  transition-delay: 250ms;  
}
```

You can also use negative values for transition-delay, which has an interesting effect: The transition begins immediately but skips ahead by the amount of the negative value. To illustrate what I mean, consider a transition with a duration of four seconds but a delay of negative two seconds:

```
E {  
  transition-duration: 4s;  
  transition-delay: -2s;  
}
```

When triggered, the transition starts immediately, but as if two seconds had already passed (two seconds being the duration minus the delay). In this case, the animation would start halfway through the transition.

Shorthand

Throughout this section, I've been building an example transition property by property. So far, the code looks like this:

```
h1 {
  font-size: 150%;
  transition-property: font-size;
  transition-duration: 2s;
  transition-timing-function: ease-out;
  transition-delay: 250ms;
}
```

This code seems like quite a lot to write for each transition. But, as with all of the other CSS properties that are part of a “family” (background-*, border-*, etc.), the transition-* family has a shorthand. Here's the syntax:

```
E { transition:
  transition-property transition-duration transition-timing-function transition-delay;
}
```

One important thing to be aware of here is that there are two time values: transition-duration and transition-delay, which must be declared in that order. If only one is declared, the syntax presumes it is transition-duration, and transition-delay will be set at the default (or inherited) value.

If you were to use the values from the example transition with the shorthand property, the result would be:

```
h1 {
  font-size: 150%;
  transition: font-size 2s ease-out 250ms;
}
```

which is, obviously, a lot less code to write.

The Complete Transition Example

Now that I've explained all of the component properties, let's take a look at the example transition in action. The full code is shown here:

```
h1 {
  font-size: 150%;
  transition: font-size 2s ease-out 250ms;
}
h1:hover { font-size: 600%; }
```

Figure 13-2 shows what happens when I pass my mouse over the h1 element.



Figure 13-2: Three stages of an animated transition on the font-size property

Obviously, I can't show the full animation on the printed page, but the illustration shows three stages of the transition: the initial, pre-transition stage (left) with a font-size of 150 percent; an intermediate, mid-transition stage (center), which is just under two seconds into the animation when the font-size has increased; and the final, post-transition stage (right) where the font-size is 600 percent.

As I've already mentioned, transitions act in reverse when the condition that acts as a trigger is no longer being met, so when you remove your mouse from over the h1 element, you can read this example from right to left to see what will happen.

Multiple Transitions

You can easily add multiple transitions to an element by providing a list of comma-separated values to the individual or shorthand properties. That being the case, both of the following code examples are valid:

```
E {
  transition-property: border-width, height, padding;
  transition-duration: 4s, 500ms, 4s;
}
E { transition: border-width 4s, height 500ms, padding 4s; }
```

Note that if a property has fewer values than the others, that list of values will be looped. With that in mind, you could rewrite this code example slightly:

```
E {
  transition-property: border-width, height, padding;
  transition-duration: 4s, 500ms;
}
```

Here, the transition-property property has three values, whereas the transition-duration property has only two. This means the third value of the former (padding) is matched with the first value of the latter (4s), matching what was supplied in the first example.

Here's a practical example:

```
.widget {
  background-color: black;
  left: 10px;
  position: absolute;
```

```
top: 90px;
transition: background-color 4s linear, left 4s ease-in-out, top 4s ease-in-out;
}
div:hover .widget {
background-color: silver;
left: 130px;
top: 10px;
}
```

Here, I've used the transition shorthand to apply three transitions. The first transition changes the background-color from black to silver in a linear timing function, and the next two change the left and top properties with ease-in-out timing functions. All the transitions take place over four seconds. You can see this illustrated in Figure 13-3.

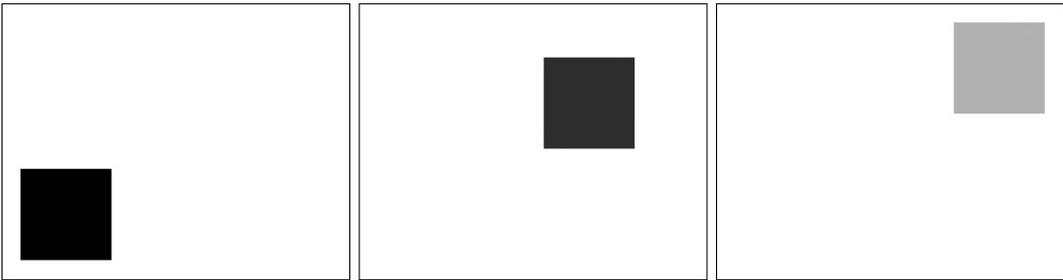


Figure 13-3: Three stages of an animated transition on the background-color, left, and top properties

Figure 13-3 shows three stages of the transition: The first stage (left) shows the element pre-transition, with a black background and positioned at the bottom left of its parent element; the next stage (center) is mid-transition, as the element is changing color and moving to the top right of its parent; and the final stage (right) shows the element post-transition, with a silver background and in its final position.

Triggers

In my examples, I've used the `:hover` pseudo-class as the trigger, but other options are available. As well as all of the other element state pseudo-classes (`:active`, `:target`), you can also combine transitions with JavaScript. For example, you could set up your CSS like this:

```
h2 {
background-color: black;
transition: background-color 2s;
}
h2.transition { background-color: silver; }
```

Then use a script to add the class and trigger the transition:

```
document.querySelector('h2').className = 'transition';
```

When the function runs, the h2 element has the class added to it, and the element's background color changes smoothly from black to silver.

More Complex Animations

Transitions are good but naturally limited; they are only applied when a property value changes. The CSS3 Animations Module (<http://www.w3.org/TR/css3-animations/>) goes beyond what is possible with Transitions, allowing animations to be applied directly to elements with a syntax that is more flexible and permits more granular control. Animations and transitions have quite a lot of syntax in common, but the process for creating animations is very different: First, you define the properties and timings, and then you add the animation controls to the elements that will be animated.

The Animations Module is currently only implemented in WebKit browsers, although Firefox developers have stated that they are considering it for implementation “in the future.”

Key Frames

The first step in creating animations is to define your *key frames*. A key frame is a point that defines the start and end of a transition. The most simple animation will have two key frames—one at the start and one at the end—whereas more complex ones will have multiple key frames in between. A CSS transition is essentially an animation with only two key frames.

In CSS, key frames are declared in the @keyframes rule, which has the following syntax:

```
@keyframes 'name' {  
  keyframe {  
    property : value;  
  }  
}
```

NOTE *As I mentioned, only WebKit browsers currently implement CSS Animations. Remember to use the WebKit prefix for this rule, which is @-webkit-keyframes.*

The first value for the @keyframes rule is *name*; this unique identifier will be used to call the animation, which I'll discuss later. You can use pretty much any value here, although I suggest using a word or term that is relevant to the animation it describes—your stylesheets will be much easier to follow if you do.

The next value, *keyframe*, sets the position along the duration of the animation that the key frame will occur. The possible values are percentages or one of the keywords *from* or *to* (which are analogous to 0 percent and 100 percent, respectively). You must specify at least two key frames (*from/0%* and *to/100%*), but you can have as many as you like as long as each has a unique position in the animation.

Within each key frame is a CSS declaration or series of declarations that are applied to a selected element at the specified stage of the animation. Let me clarify with an example. The following code describes a simple animation with three key frames:

```
@keyframes 'expand' {  
❶ from { border-width: 10px; }  
❷ 50% { border-width: 1px; }  
❸ to {  
❹   border-width: 1px;  
     height: 120px;  
     width: 150px;  
   }  
}
```

At the beginning of the animation (❶), the selected element has a border that is 10px wide; halfway through the animation (❷), the border is reduced to a width of 1px; and at the end of the animation (❸), the border is 1px wide, and the height and width are 120px and 150px, respectively. Between each of the key frames, the elements are animated gradually, so between the start and 50 percent mark, the border is animated to smoothly change width from 10px to 1px.

Note that inheritance operates on individual key frames, so if you want a change to persist between frames, you need to specify it in each frame. If I hadn't specified `border-width` again in the `to` key frame (❹), it would default back to the inherited value of the element the animation was applied to, meaning the animation could be quite different.

Animation Properties

Once you've defined the key frames, the next step is to add the controls to the elements you want to animate. As I mentioned in the introduction to this section, many of the animation properties share syntax with their counterparts in the `transition-*` family, so you should already be pretty familiar with them.

NOTE *As with the `@keyframes` rule, the `animation-*` properties are currently only implemented in WebKit and so should be prefixed with `-webkit-` when you use them in your pages.*

Name

The `animation-name` property is used to refer to an animation that's been defined with the `@keyframes` rule, and as such the syntax is quite straightforward:

```
E { animation-name: name; }
```

You can see it requires only a single value, which is the name of an already defined animation. To call the animation created in the previous section, you would use:

```
div { animation-name: border-changer; }
```

The only other permissible value is *none*, which prevents any animations from occurring on this element.

Duration

The duration of an animation is set by the `animation-duration` property, which is functionally identical to the `transition-duration` property introduced earlier in this chapter:

```
E { animation-duration: time; }
```

As with `transition-duration`, the *time* value is a number with a unit of either ms or s, or a 0 (zero), which prevents the animation from running. (Negative values are also implemented as 0.) To extend the example animation so it's six seconds in duration, you add this line:

```
div { animation-duration: 6s; }
```

Timing Function

Another property that should be familiar is `animation-timing-function`, which again is functionally identical to its counterpart `transition-timing-function`:

```
E { animation-timing-function: keyword OR cubic-bezier(x1, y1, x2, y2); }
```

Permitted values are `ease`, `linear`, `ease-in`, `ease-out`, and `ease-in-out`. The `cubic-bezier` function is also allowed. I explain these values in “Timing Function” on page 166.

Therefore, if you want to make your animation “ease in” at the start, you add this code:

```
div { animation-timing-function: ease-in; }
```

Delay

Yet another property that has (kind of) already been introduced is `animation-delay`, which is identical in function to its counterpart, the `transition-delay` property:

```
E { animation-delay: time; }
```

Like `animation-duration` (and the two corresponding `transition-*` properties), the *time* value is a number with a unit of either ms or s, which sets the delay before an animation begins. A 0 (zero) means no delay. As with `transition-duration`, negative values cause the animation to “skip” by that amount.

To delay the example animation by two seconds, you include this code:

```
div { animation-delay: 2s; }
```

Iteration Count

Unlike a transition, which only happens once (or twice, if you want to count the reverse), an animation can be repeated any number of times. The number of repetitions is set by the `animation-iteration-count` property, which has this syntax:

```
E { animation-iteration-count: count; }
```

The *count* value in this syntax is either a whole number or the keyword *infinite*. A number value sets how many times the animation repeats. The default value is 1 (one), meaning the animation plays from start to end once and then stops. The *infinite* value means the animation loops indefinitely, or at least until another condition is met that changes this value. A value of 0 (zero) or any negative number prevents the animation from playing.

To continue with the example I've been building throughout this section, if you want the animation to repeat 10 times, you need to add the following declaration:

```
div { animation-iteration-count: 10; }
```

Direction

Animations play from start to finish, but they can also play in reverse (like transitions do). You can set whether your animation always plays in one direction or alternates playing forward and backward. To do this, you use the `animation-direction` property:

```
E { animation-direction: keyword; }
```

The *keyword* value has two options: *normal* or *alternate*. The default is *normal*, which always plays the animation forward: The animation plays through from start to finish, and then, if it is set to repeat, it plays again from the start. If the *alternate* value is used, the animation plays from start to finish and then plays in reverse before starting over again. If you consider each iteration of the animation as a "cycle," the odd-numbered cycles play forward and the even-numbered play backward.

To complete the example animation, let's set the animation to alternate forward and backward:

```
div { animation-direction: alternate; }
```

Shorthand

Throughout this section, I've been assembling an example animation one property at a time. Here's how all the properties combined appear:

```
div {  
  animation-name: border-changer;  
  animation-duration: 6s;  
  animation-timing-function: ease-in;  
  animation-delay: 2s;  
  animation-iteration-count: 10;  
  animation-direction: alternate;  
}
```

That's a lot of different properties to declare for each animation. Once again, however, you can take advantage of a shorthand property; it's called `animation`, and here's the syntax:

```
E { animation: animation-name animation-duration animation-timing-function  
animation-delay animation-iteration-count animation-direction; }
```

As with the transition shorthand, `animation-duration` and `animation-delay` must be specified in order. If either is left out, the syntax presumes it is `animation-delay`, which receives a default value of 0.

To make the example somewhat more compact, you can use this shorthand code:

```
div { animation: 'border-changer' 6s ease-in 2s 10 alternate; }
```

Play State

One animation property isn't included in the shorthand. The `animation-play-state` property sets whether an animation is active. Here's the syntax:

```
E { animation-play-state: keyword; }
```

The *keyword* value has two options: `running` means the animation is playing, and `paused` means it isn't. You can use this property to perform a play/pause action, like so:

```
div { animation: 'border-changer' 6s infinite alternate; }  
div:hover { animation-play-state: paused; }
```

In this example, the animation loops continuously until the mouse passes over it, at which point the animation pauses; when the mouse is moved off the element, the animation continues.

WARNING *A note in the specification states that the `animation-play-state` property may be removed in the future, but for now it is part of the language.*

The Complete Animations Example

Let's take a look at the full example animation in action. Here's the CSS:

```
@keyframes 'expand' {  
❶ 0% { border-width: 4px; }  
❷ 50% { border-width: 12px; }  
❸ 100% {  
    border-width: 4px;  
    height: 130px;  
    width: 150px;  
  }  
}  
div {  
  border: 4px solid black;  
  height: 100px;  
  width: 100px;  
  box-sizing: border-box;  
  animation: 'expand' 6s ease 0 infinite alternate;  
}
```

The result is shown in Figure 13-4.

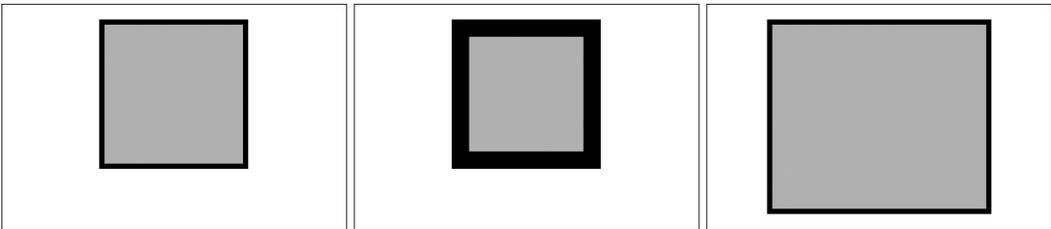


Figure 13-4: Three stages of an animation affecting the border-width, height, and width properties

Unfortunately, I can't show animation in this book, so I'll settle for describing it. Figure 13-4 shows the state of the element at the three key frames of the animation: The key frame at 0 percent (❶) shows the element with a border-width of 4px and the inherited height and width (100px each); in the 50 percent key frame (❷), the border-width is increased to 12px; and in the final, 100 percent key frame (❸), the border-width returns to 4px and the dimensions have changed to 150px by 130px.

Multiple Animations

You can add multiple animations to an element using a comma-separated list. This method works for each of the subproperties and the shorthand property, so both of these examples are valid:

```
E {  
  animation-name: first-anim, second-anim;  
  animation-duration: 6s, 1250ms;  
  animation-delay: 0, 750ms;  
}  
E { animation: first-anim 6s, second-anim 1250ms 750ms; }
```

Here, lists of values are looped through to ensure that all properties have the same number of values applied, in exactly the same way as described in “Multiple Transitions” on page 170.

Summary

Adding a behavioral layer to CSS was a contentious move, but I think the syntax used by the writers of these modules is pretty graceful and adds a lot of flexibility without being overwhelmingly complicated.

Developers expressed concern that transitions and animations would be misused and that we’d end up with a lot of garish, unusable websites. Although a genuine concern, one really can’t do anything to stop people from doing that now with existing CSS properties. (And indeed, many do!) But used sparingly and appropriately, this powerful new tool can add vibrancy to web pages.

In the next chapter, we’ll look at the final new module in what’s loosely termed the “CSS Effects” group: 3D Transformations.

Transitions and Animations: Browser Support

	WebKit	Firefox	Opera	IE
Transitions	Yes (with prefix)	No (expected in Firefox 4 with prefix)	Yes (with prefix)	No
Animations	Yes (with prefix)	No	No	No

14

3D TRANSFORMATIONS



Everything we've discussed in CSS up until now has involved two dimensions; every element has height and width but no depth, and all of the calculations have involved only the x -axis and y -axis. But, with the introduction of the z -axis in the CSS 3D Transforms Module, CSS3 introduces a really revolutionary way of transforming an element in the third dimension (you can learn more at <http://www.w3.org/TR/css3-3d-transforms/>).

Moving objects around in three dimensions (3D) requires quite a lot of computer processing power, so to see elements in true 3D, you need a browser that has hardware acceleration for graphics; currently only Safari 4.03 and above on Mac OS X 10.6 and iPhone OS 2 and above offer this. Other browsers plan on including hardware acceleration in future releases. With that being the case, all of the properties used in this chapter should be prefixed with `-webkit-`. However, as in other chapters, I'll leave it out of the examples for clarity.

The 3D Transforms Module was originally proposed by the WebKit team but has been accepted by the W3C to the recommendation process. It currently has Working Draft status and is very likely to change somewhat in the future; couple that with the fact that it only works on a small subset of possible hardware/software configurations and you might want to think twice before making transformations the crucial center point of a production website.

3D Elements in CSS

Three-dimensional objects in CSS are based on the Cartesian coordinate system, which is illustrated in Figure 14-1. You can read about it at Wikipedia (http://en.wikipedia.org/wiki/Cartesian_coordinate_system). We discussed the two-dimensional version of this system in Chapter 6.

NOTE *If you have experience using three-dimensional computer graphics programs, you should be familiar with the calculations and terminology used in this chapter. If not, don't worry; I'll do my best to explain it all as I go along.*

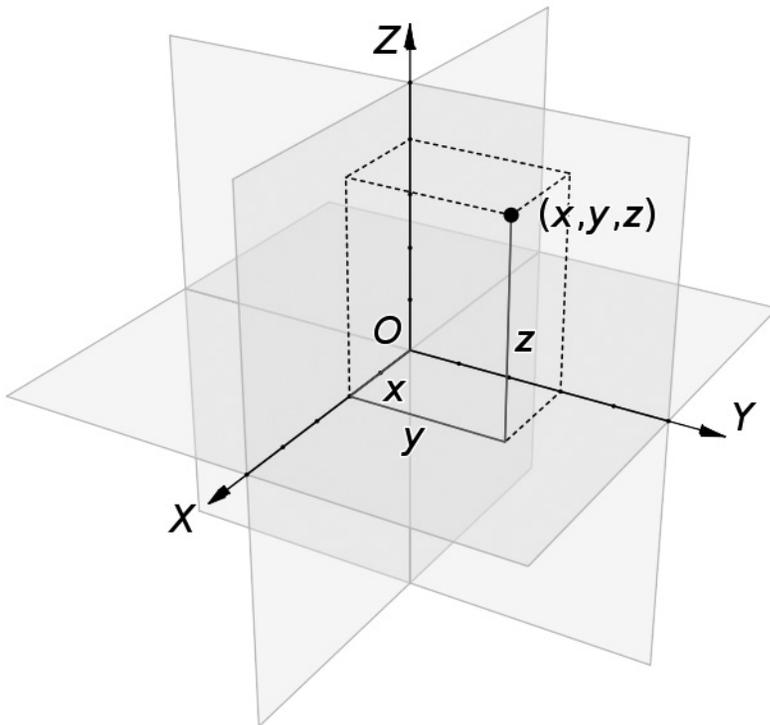


Figure 14-1: The Cartesian coordinate system, with the axes x , y , and z ¹

¹ This image is from Wikimedia Commons (http://commons.wikimedia.org/wiki/File:Coord_system_CA_0.svg).

In CSS, the *z*-axis is based on the viewer: If you think of the *x*-axis as left to right and the *y*-axis as up and down, then think of the *z*-axis as toward and away. When you move an element along the *z*-axis by a positive value, you move it toward yourself; likewise, moving it by a negative value moves it away from you. You can change this somewhat by using different perspectives, which I'll explain in due course.

Illustrating three-dimensional transformation concepts on the printed page is quite hard, so I strongly suggest you take a look at the examples on the website that accompanies this book (<http://www.thebookofcss3.com/>); I've added transition rules to the examples so they animate when you interact with them, showing more clearly the three-dimensional effects that are difficult to convey in a two-dimensional book. These examples should really help with understanding some of the techniques used in this chapter. I also recommend the test page put together by the team at Westciv (<http://www.westciv.com/tools/3Dtransforms/>). This page allows you to try out different combinations of transformation values to see the effect they have on elements.

Before I begin introducing the new and extended transformation properties, a quick note about the examples I'm using in this chapter. Although each example may use different classes to apply unique style rules, all of the examples use the same basic markup:

```
<div class="threed">
  <div>
    <h2>The Book Of</h2>
    <h1>CSS3</h1>
  </div>
</div>
```

Unless otherwise noted, the transformation functions in this chapter are applied to the `div`, which is a child of `.threed`. To more clearly demonstrate some of the 3D effects, I've already rotated the parent element around the *x*-axis and *y*-axis, using the `transform` properties introduced in Chapter 12. Here is the code I've used (don't worry about what it means yet; I will explain it throughout the course of this chapter):

```
.threed { transform: rotateX(15deg) rotateY(45deg); }
```

You can see the result of this code in Figure 14-2. If, in the examples, I refer to a "default" or "untransformed" element, I am referring to this reference element with these transformations applied and no others.

Again, I encourage you to visit the website that accompanies this book (<http://www.thebookofcss3.com/>) to take a look at the example files.



Figure 14-2: A reference element used in many examples in this chapter

Transform Style

The first new property is very simple but very important; if you don't change it from the default value, you won't be able to view your transformations in three dimensions. The property is called `transform-style`, and here's the syntax:

```
E { transform-style: keyword; }
```

The *keyword* value can be either `flat` (the default) or `preserve-3d`. Explaining the difference is easier if I start with an example: First, I'll create two elements that are identical except for the value given to `transform-style`:

```
.div-1 { transform-style: flat; }  
.div-2 { transform-style: preserve-3d; }
```

I also applied some 3D transformations that aren't important for this example, but I'll explain those later in this chapter. The result is shown in Figure 14-3.

The difference is quite stark. The first example has the value of `flat`, and so the three-dimensional transformation is only applied on a two-dimensional plane, distorting the element but not providing any real sense of depth. The second example, by contrast, has a `transform-style` value of `preserve-3d`, transforming the element in three dimensions and giving a true sense of depth. All of the examples in the rest of this chapter have this value, and you'll need this value to make your own three-dimensional page elements.

The value given to `transform-style` affects an element's children, not the element itself. One caveat, however: If an element has an *overflow* value of `hidden`, its children can't be displayed in 3D, so that element behaves as if the `transform-style` value were `flat`.



Figure 14-3: Comparing the effects of different values for the `transform-style` property

The Transformation Functions

In Chapter 12, I introduced the `transform` property and its associated functions. Three-dimensional transformations use the same `transform` property and many of the same functions but also extend some of those functions and add some entirely new ones. I'll explain each of the transformation functions in turn and note whether they are entirely new or extend the existing 2D Transformation properties that have already been discussed.

Rotation Around an Axis

I'll begin explaining the 3D transformation functions as I did the 2D functions—with rotation. Two-dimensional space has only one axis to rotate around, so the rotate function requires only a single value. But when you're dealing with three dimensions, you've got three axes to rotate around and, therefore, three properties to control this. Here they are:

```
E {  
  transform: rotateX(angle);  
  transform: rotateY(angle);  
  transform: rotateZ(angle);  
}
```

Like the rotate function, each of the functions accepts a single angle value. I'll use the deg (degrees) unit in my examples, so negative values are permitted. I'll demonstrate how each of these works by showing a rotation around each axis using the following code:

```
❶ .trans-x { transform: rotateX(-60deg); }  
❷ .trans-y { transform: rotateY(22.5deg); }  
❸ .trans-z { transform: rotateZ(22.5deg); }
```

You can see the results in Figure 14-4.

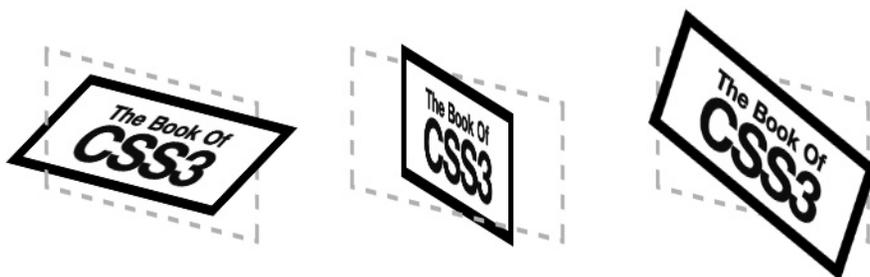


Figure 14-4: Rotation around each of the three axes

The first example (❶) shows an element rotated 60 degrees around the *x*-axis. To understand what's happening here, imagine a line running horizontally across the center the element; the half of the element above the line is inclined 60 degrees toward you, and the half below the line is inclined 60 degrees away. The next example (❷) has a 22.5 degree rotation applied but this time on the *y*-axis. Imagine a vertical line running down the center of the box; the half of the element to the left of the line is inclined 22.5 degrees toward you and the half on the right, 22.5 degrees away. The final example (❸) shows the same rotation, 22.5 degrees, but this time around the *z*-axis, which has the same effect as the two-dimensional rotate function.

If you want to rotate an element around more than one axis, you can apply multiple functions to an element:

```
E { transform: rotateX(angle) rotateY(angle) rotateZ(angle); }
```

Another new function—`rotate3d`—also allows you to rotate an element around multiple axes, however. Here’s the syntax:

```
E { transform: rotate3d(x,y,z,angle); }
```

The *angle* value is straightforward, but the *x*, *y*, and *z* values are not quite so simple. Each takes a number value, which is used to calculate a direction vector (the full explanation of which is beyond the scope of this book; visit <http://www.tutorvista.com/math/3d-cartesian-coordinates/> for an overview of the topic). The origin of the vector is the point where all the axes meet, represented by the values 0,0,0. A direction vector sets a point in 3D space; the *x,y,z* values provided to the `rotate3d` function plot a point at that number of pixels along each axis. Imagine a line between the origin and that point—that’s the line around which the rotation happens by the angle specified.

This subject is quite complex, so I’ll explain by showing a few simple examples using the following code:

-
- ❶ `.trans-x { transform: rotate3d(1,1,0,-45deg); }`
 - ❷ `.trans-y { transform: rotate3d(1,0,1,-22.5deg); }`
 - ❸ `.trans-z { transform: rotate3d(0,10,10,15deg); }`
-

You can see the output in Figure 14-5.

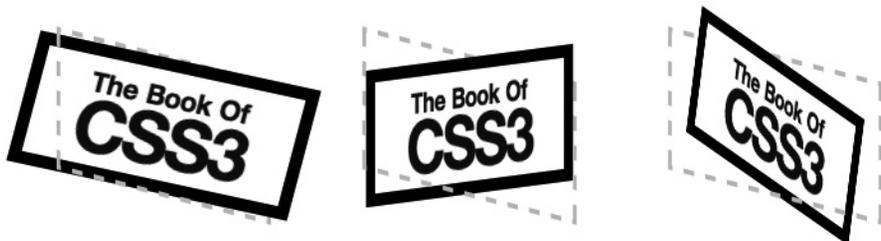


Figure 14-5: Rotation using directional vectors with the `rotate3d` function

The first (left) example (❶) has the values 1,1,0, meaning the imaginary line goes to a point 1px along the *x*-axis and *y*-axis. (In fact, the “line” continues past that point in the same direction; the values 10,10,0 or 1000,1000,0 would produce the same results.) The element is rotated 45 degrees around that line, as shown in the example on the left in Figure 14-5. The second (middle) example (❷) has the values 1,0,1, creating a point 1px along the *x*-axis and *y*-axis and rotating the element by 22.5 degrees around that line. The final example (❸) has the values 0,10,10, so the element rotates 15 degrees around a line between the origin and a point 10px along the *y*-axis and *z*-axis,

as shown on the right in Figure 14-5. Remember that any two equal values would have the same effect.

You probably won't use this function very often when coding pure CSS transformations, as the calculations involved are quite complex. But when combined with JavaScript, the flexibility of this approach could really come into its own.

Translation Along the Axis

The `translateX` and `translateY` functions (and their shorthand, `translate`) are used to move an element along its axis by a specified length in two-dimensions, but the move into a third dimension requires a new function: `translateZ`. The syntax is identical to its sibling properties:

```
E { transform: translateZ(length); }
```

The *length* value is any number with a unit of length. For example, to move a `div` element 30px along the *z*-axis (toward the viewer), you use

```
div { transform: translateZ(30px); }
```

Before I demonstrate this, however, allow me to introduce the new shorthand function, `translate3d`. This shorthand function allows you to set all three values. The syntax is a logical extension of the `translate` function:

```
E { transform: translate3d(translateX,translateY,translateZ); }
```

Each of the values is equivalent to the named function, so each accepts a numerical value, positive or negative, with a CSS length unit.

Now that you've met the new functions, let's see them at work. In the following example, I show a mix of different `translate` functions using this code:

```
❶ .trans-xy { transform: translateX(10px) translateY(10px); }  
❷ .trans-z { transform: translateZ(40px); }  
❸ .trans-xyz { transform: translate3d(10px,0,-20px); }
```

You can see the results in Figure 14-6.



Figure 14-6: Showing translation along different axes

The first (left) example (❶) has a value of 10px for the `translateX` and `translateY` functions, which means it is moved by that amount along the *x*-axis and *y*-axis, which places it below and to the right of its original position. The next (❷) has a value of 40px on the `translateZ` function, moving it by that amount along the *z*-axis and making it appear “in front” of the original, as the middle example shows. In the final example (❸), the `translate3d` function is used to move the element 10px along the *x*-axis and 20px along the *z*-axis, while retaining the same position on the *y*-axis, which puts it below and behind its original position, as in the example on the right in Figure 14-6.

Scaling

I also introduced the `scale` function, along with the subfunctions `scaleX` and `scaleY`, in Chapter 12. The move to three dimensions adds a new subfunction, `scaleZ`, which has this syntax:

```
E { transform: scaleZ(number); }
```

As with its siblings, the *number* value provides a factor to scale the element by, so a value of 2 would double the element’s size along the *z*-axis. The resulting behavior is probably not what you’d expect, however—the element itself has no depth; it has only height and width, so an increase in `scaleZ` by itself doesn’t seem to change the element. What the increase actually does is act as a multiplier to any value that’s supplied to `translateZ`. For example, consider this code:

```
div { transform: scaleZ(3) translateZ(10px); }
```

The `scaleZ` function’s value of 3 would multiply the `translateZ` function’s value of 10px, so the element would appear 30px ($3 \times 10\text{px}$) along the *z*-axis.

In addition to `scaleZ`, a new shorthand function, `scale3d`, has also been added. Here is its syntax:

```
E { transform: scale3d(scaleX,scaleY,scaleZ); }
```

As should be pretty clear, this shorthand simply accepts a number for each of the values, acting as a scaling factor on the pertinent axis. Here are a few examples to show you 3D scaling in action:

-
- ❶ `.trans-xy { transform: scaleX(1.5) scaleY(1.5); }`
 - ❷ `.trans-z { transform: scaleZ(3) translateZ(10px); }`
 - ❸ `.trans-xyz { transform: scale3d(1.25,1.25,4) translateZ(10px); }`
-

You can see the results in Figure 14-7.



Figure 14-7: Scaling on different and multiple axes

In the first (left) example (❶), I've applied values of 1.5 to both `scaleX` and `scaleY`, so the element is half again its original size on the two-dimensional axes. The second (middle) example (❷) shows an element with a `translateZ` value of 10px and a `scaleZ` value of 2.5; as I mentioned, `scaleZ` acts as a multiplier of `translateZ`, so the element appears 25px along the *z*-axis. You can see the result in the middle example in Figure 14-7. In the final example (❸), I used the `scale3d` function to set values of 1.25 on the *x*-axis and *y*-axis and 4 on the *z*-axis. The resulting element, shown on the right in Figure 14-7, is 25 percent larger on the two-dimensional axes, and the `scaleZ` value multiplies the `translateZ` value of 10px to move the element 40px along the *z*-axis.

The Transformation Matrix

One of the more esoteric aspects of 2D Transformations is the `matrix` function, which I introduced in Chapter 12. This function allows complex transformations to be applied using six values (based around a grid pattern) and some trigonometric calculations. You can also apply three-dimensional transformations using a matrix with the `matrix3d` function. But if you thought the 2D matrix was hard to grasp, you might want to skip this one—`matrix3d` has a whopping 16 values! Here's the syntax:

```
E { transform: matrix3d(  
    m01,m02,m03,m04,  
    m05,m06,m07,m08,  
    m09,m10,m11,m12,  
    m13,m14,m15,m16  
); }
```

NOTE *The line breaks are shown here for clarity; you don't need to use them in practice.*

Each of the *m* values is a number, but I can't even begin to explain what each of them does! I would suggest you read an introduction to the subject (http://gplwiki.org/index.php/3D:Matrix_Math) and decide if this is something you want to learn more about. Remember that all of the `matrix3d` effects can be achieved with different functions; this function is here if you want fine control or an easy function to interact with using JavaScript.

I'll provide some simple examples using this code to demonstrate the functionality:

-
- ❶ `.trans-1 { transform: matrix3d(1,0,0,0,0,1,0,0,0,0,1,0,10,10,10,1); }`
 - ❷ `.trans-2 { transform: matrix3d(1.5,0,0,0,0,1.5,0,0,0,0,2,0,0,0,10,1); }`
 - ❸ `.trans-3 { transform: matrix3d(0.96,-0.26,0,0,0.26,0.96,0,0,0,0,1,0,-10,0,20,1); }`
-

You can see the results in Figure 14-8.



Figure 14-8: Transformations applied with the `matrix3d` function

The first (left) example (❶) shows the element moved 10px along each axis with the equivalent of the `translate3d` function—the `m13`, `m14`, and `m15` values in the matrix operate as `translateX`, `translateY`, and `translateZ`, respectively. In the second example (❷), I scaled the image by a factor of 1.5 on the *x*-axis and *y*-axis (the `m1` and `m6` values) and by a factor of 2 on the *z*-axis (the `m11` value), which multiplies the `translateZ` value (`m15`) to move the element 20px along the *z*-axis, as shown in the middle example in Figure 14-8. The final example (❸) requires a scientific calculator for some trigonometric functions, as I've rotated the element by 15 degrees on the *z*-axis. To create the rotation, you need to give a value of $\cos(15)$ —which is 0.96—to `m1` and `m6` and then $\sin(15)$ —that is, 0.26—to `m5` and negative $\sin(15)$ to `m2`. I also translated the element by 10px on the *x*-axis with the value in `m13`. You can view the result on the right in Figure 14-8.

As I'm sure you can see already, this function is very powerful—and very complex. Whether you want to learn the full potential of `matrix3d` depends largely on you and your projects, but I feel it's beyond the scope of this book. Don't forget that you can perform all of these transformations with the individual functions; you'll end up with more code, but it will be easier to understand—not only for yourself but also for anyone tasked with maintaining the website after you!

Perspective

The final function is perspective, a brand-new function that creates an artificial viewpoint from where you view the 3D object, providing the illusion of depth. Here's the syntax:

```
E { transform: perspective(depth); }
```

The value *depth* is an integer that represents a length (in pixels) or the default of none. This length sets a “viewpoint” at that distance along the *z*-axis away from the element’s origin (*z* = 0). If the element is translated along its *z*-axis, the distance of the viewpoint from the element dictates how large the element will appear to be.

Consider an element with a `translateX` value of 10px. If the perspective value is 50, the element appears exceptionally large; if the perspective value is 1,000, the element appears to be its original size. An element only appears smaller than its original size if you move it negatively along the *z*-axis and view it from a low perspective.

perspective is probably easier to show than to describe. I’ll provide some examples of different values for the perspective function to show you the viewpoint changes. Here’s the code I’ll use:

```
❶ .trans-1 { transform: rotateX(-90deg) rotateY(-15deg) perspective(20) translateZ(10px); }
❷ .trans-2 { transform: rotateX(-90deg) rotateY(-15deg) perspective(50) translateZ(10px); }
❸ .trans-3 { transform: rotateX(-90deg) rotateY(-15deg) perspective(1000) translateZ(10px); }
```

Before I show the results, I’d like to make a slight digression. You may be wondering why the code contains so much repetition; why couldn’t I have done something like this instead?

```
div { transform: rotateX(-90deg) rotateY(-15deg) translateZ(10px); }
.trans-1 { transform: perspective(20); }
.trans-2 { transform: perspective(50); }
.trans-3 { transform: perspective(1000); }
```

The reason is that if you don’t specify a function, its value is presumed to be the default, so the values I set in the functions on the `h1` element are effectively overwritten by (the absence of) the functions in the subsequent styles. For example, by not restating `rotateX(-90deg)` in any of the further declarations, the assumption is it defaults back to `rotateX(0)`.

Anyway, back to the example. You can see the results in Figure 14-9.

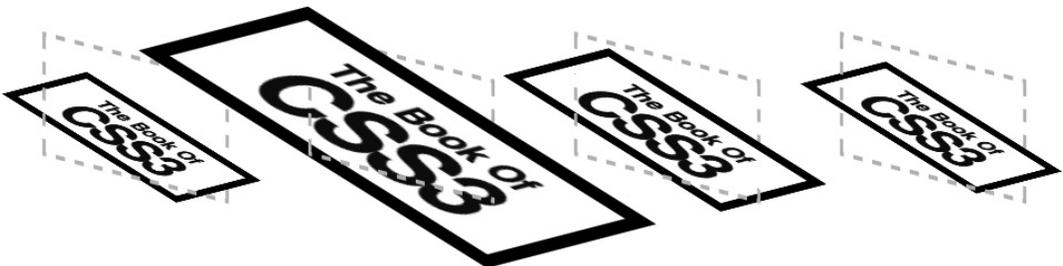


Figure 14-9: The effects of different perspective function values

The first example in Figure 14-9 is an untransformed reference, different than the one used up to now. In the second example (❶), you see the same element from a perspective value of 20; that means that you are effectively viewing it 20px along the *z*-axis, and as the element itself is already translated

10px along that axis, it is half the distance between the viewpoint and the origin and appears to be double its original size. In the third example (❷), the perspective value has increased to 50, placing the viewpoint a little farther away from the element. So although it is smaller than the previous example, as you can see in the figure, the element is still larger than the reference. The final (far right) example (❸) shows the element from a perspective distance of 1,000, and the element appears to be about the same size as the reference example in Figure 14-9. From higher values like 1,000, perspective often has little noticeable effect.

The perspective and perspective-origin Properties

I've just covered the perspective transformation function, but a perspective property is also available. The syntax is pretty straightforward:

```
E { perspective: depth; }
```

This property operates in the same way as the perspective function: The *depth* value is a number that sets the distance from the element's origin, $z = 0$. In fact, the only difference between the function and the property is that the value supplied to the property applies only to its child elements, not to itself.

The companion property of perspective is perspective-origin. This property sets the origin point of the element from where the perspective will be viewed, changing the angle at which the element is viewed. Here's the syntax:

```
E { perspective-origin: x-position y-position; }
```

The *x-position* value can be any one of the keywords left, right, or center; and the *y-position* value can be top, bottom, or center. Percentage or length values can also be used. You should be familiar with these values from other properties, such as background-position or transform-origin.

The default is center center (or 50% 50%), so you are viewing the line of perspective as if it started at the absolute center of the element. Changing the values of perspective-origin changes the origin of that line of perspective.

That may sound a little brain-bending, but once again, showing it is easier than explaining it. In the next example, I'll show the same transformed element from different perspective origins. Here's the code:

```
.three { perspective: 200; }  
.three div { transform: rotateX(45deg) rotateY(-15deg) rotateZ(90deg) translateZ(20px); }  
❶ .trans-1 { perspective-origin: left center; }  
❷ .trans-2 { perspective-origin: 128px 56px; }  
❸ .trans-3 { perspective-origin: 75% 25%; }
```

The examples are illustrated in Figure 14-10.

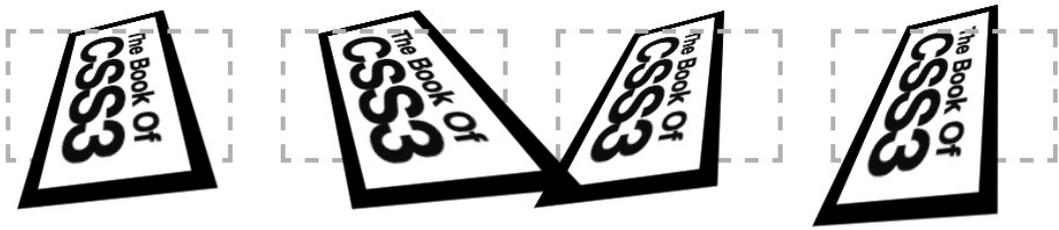


Figure 14-10: Different values for the `perspective-origin` property

A reference example is shown first; this reference is a transformed element viewed from the default origin of center center. The second example (❶) is the same element but with `perspective-origin` at the center of the left-hand side of the element. You can clearly see that in the second example in Figure 14-10. The angle you’re viewing the element from has changed; the viewpoint appears to be slightly to the left of it. In the third example of the figure (❷), I’ve used length values of 128px 56px, which changes the origin to the bottom of the right-hand side. This time the viewpoint appears to be slightly to the right and looking up. The last (far right) example (❸) is viewed from a point 75 percent along the *x*-axis and 25 percent along the *y*-axis; this example is similar to the reference example, but the viewpoint has shifted slightly to be just to the right of, and slightly above, the element.

The Transformation Origin

An element origin is the point at which the *x*-axis, *y*-axis, and *z*-axis meet. By default, this point is the exact center of an element, but you can change this using the `transform-origin` property, which I introduced in Chapter 12. Obviously that property was written for two-dimensional transformations, but you can also use it on three-dimensional ones by extending the syntax:

```
E { transform-origin: x,y,z; }
```

The first two values, *x* and *y*, act the same as in the `transform-origin` property for 2D elements; that is, they accept values as either a keyword (left, right, center for *x*, and top, bottom, center for *y*), a length unit, or a percentage. The default value is center center, or 50% 50%. The third value, *z*, is a length value, which sets the distance along the *z*-axis that the transformation will happen. This can seem quite counterintuitive as it seems to act in reverse; if a negative value is given, the transformation origin is behind the element, which makes it appear in front of its parent; likewise, a positive value places the origin in front of the element, making the element appear behind its parent.

I’ll illustrate this with a demonstration showing three elements identical in all values but `transform-origin`. Here’s the relevant code for these examples:

```
❶ .trans-1 div { transform-origin: 0 28px 0; }
❷ .trans-2 div { transform-origin: 100% 100% 10px; }
❸ .trans-3 div { transform-origin: center bottom -20px; }
```

You can see the output in Figure 14-11.



Figure 14-11: Different `transform-origin` values on a transformed element

The first example is, once again, a reference element. The next example (❶) shows the same element with the `transform-origin` value set to `0 28px`, or the center of the left-hand side (the element is 56px high), and in the original (unchanged) position on the *z*-axis. The third example (❷) has values of `100% 100% 10px`, so the transformation origin is set to the bottom-right corner of the element, which appears to be moved 10px along the *z*-axis away from the viewer—due, as I explained, to the origin now being in front of the element. In the final (far right) example (❸), the origin of the transformation is set to be the center bottom of the element, which moves it 20px along the *z*-axis—that is, toward the viewer (as it appears).

The original WebKit proposal also has three individual subproperties:

```
E {  
  transform-origin-x: value;  
  transform-origin-y: value;  
  transform-origin-z: value;  
}
```

These accept the same values as the subvalues of the `transform-origin` property just discussed: either the positional keywords (`left`, `right`, `center`, `top`, `bottom`), a percentage, or a length value. I could have declared the third example (❸) from those shown previously like this:

```
.trans-3 div {  
  transform-origin-y: bottom;  
  transform-origin-z: -20px;  
}
```

Although these subproperties do currently work in Safari, they don't appear in the W3C module, so they may have been dropped and won't appear in future implementations by different browsers.

Showing or Hiding the Backface

Often you'll encounter a situation where the element has been rotated so it faces away from you and you are seeing the "back" of it (known as the *backface*). By default, the element behaves as if it were transparent, so you will see the reverse of what appears on the front. You can change this by using the `backface-visibility` property, which has this syntax:

```
E { backface-visibility: state; }
```

The *state* value is one of two keywords: `hidden` or `visible`. The default is `visible`, which behaves in the way I just described; the alternative, `hidden`, shows nothing. These work in the same way as the `visibility` property, which you should be familiar with from CSS2.

I'll provide a quick example of the difference between the two states by showing two elements that are identical except for their `backface-visibility` values. Here's the code:

```
.threed div {  
  backface-visibility: visible;  
  -webkit-transform: rotateY(90deg);  
}  
div.trans-3d { backface-visibility: hidden; }
```

You can see the results in Figure 14-12.



Figure 14-12: Demonstrating the `backface-visibility` property

Both example elements are rotated around the *y*-axis so they are facing away from you. The example on the left shows the element with a value of `visible` on the `backface-visibility` property, so you can clearly see the back of the element. In the example on the right, you can see . . . well . . . nothing. The `backface-visibility` property has a value of `hidden`, so nothing is displayed—no border, no background-color. Nothing. To confirm that the element is actually there, visit the website that accompanies this book and take a look at the animated code examples so you can better see how they work.

Summary

The introduction of the third dimension takes CSS into areas that are rich with potential but immensely daunting. Performing simple transformations is easy now, but for anything beyond simple, you'll need strong math skills—at least until someone invents tools that can calculate all of this for you.

This module will almost certainly change before it is ready for release, so implementation in other browsers may be some distance down the line. Three-dimensional transformations are about as cutting-edge as CSS can be, so I suggest you have fun making demos and learning the syntax and the possibilities, but don't consider it for your client sites in the near future.

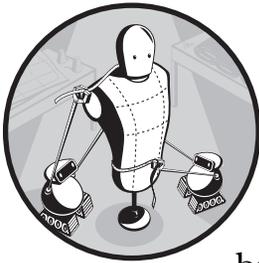
In the next chapters, I'll move on to more of the bleeding edge of CSS and look at a new subject: alternatives for laying out your pages.

3D Transformations: Browser Support

	WebKit	Firefox	Opera	IE
3D Transformations	Yes (with prefix)	No	No	No

15

FLEXIBLE BOX LAYOUT



That web pages look as good as they do is a minor miracle (comparatively speaking, of course!). Since the move away from table-based layouts to pure CSS, developers have often had to create complex page structures using simple tools such as `float`, `margin`, and `position`—tools that perhaps were not even intended for that purpose when they were created. As CSS matures and browsers get more powerful, a whole new range of alternative toolsets have been proposed, and over the next three chapters, I'll discuss each of these in turn. Note that we're moving beyond the cutting edge and onto the bleeding edge—some of these toolsets are not yet ready for daily use.

I'll start with the proposal that's closest to being ready. The Flexible Box Layout Module (<http://www.w3.org/TR/css3-flexbox/>, from here on known as *Flex Box*) is based on a syntax used by Mozilla for many of its software products, including Firefox. After being proposed as a standard, Flex Box was implemented in the KHTML layout engine, which is at the core of WebKit, and as a result is well implemented in browsers based upon that engine.

The Firefox implementation is now quite old, however, and hasn't kept up to date with changes in the specification. Although its reimplementations has been flagged as a priority for future releases, its current version has some quirks that make it problematic to use—I'll provide more detail about this, where relevant, throughout the chapter.

Some properties in this module have also been implemented in the pre-release versions of IE9, although nothing has yet been officially announced by Microsoft. Whether Flex Box will make it into the final release remains to be seen. The properties that have been implemented will use the `-ms-` prefix.

So what is the Flex Box layout method? Flex Box makes elements more fluid, allowing them to resize or change position in relation to their parent or sibling elements without your needing to specify positioning or perform complex calculations. This flexibility is ideal for modern web design, which has to deal with a wide variety of screen sizes across different devices.

Triggering the Flexible Box Layout

In order to initiate the Flex Box layout mode, you must specify a containing element, whether the document body or child element within it. You do this by using the existing `display` property with a new value:

```
E { display: box; }
```

When this declaration is set on an element, all of its immediate child elements are subject to Flex Box layout rules. Be aware that this differs from the example declarations you'll see in the rest of this chapter. Those all require the browser-specific prefix on the property; here you must set it on the value. For practical purposes, you must declare this:

```
E {  
  display: -moz-box;  
  display: -ms-box;  
  display: -webkit-box;  
}
```

With that advisory out of the way, let's take a look at the effect this property has. In this first example, I'll show a parent element with three children and apply the `box` value to the parent. Here's the code:

```
#flexbox-holder {  
  display: box;  
  width: 600px;  
}  
div.flex { width: 200px; }
```

You can see the result in Figure 15-1.

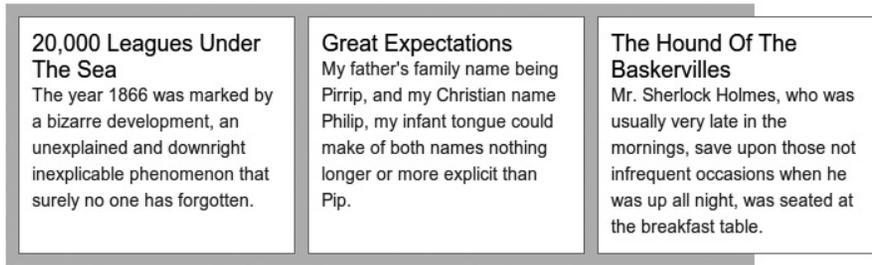


Figure 15-1: The effect of using the box value on the display property

The first thing you notice is that, without specifying any float or position values, the child elements are displayed horizontally within the parent; I’ll explain why this happens later in this chapter in “Changing Orientation” on page 204.

The other thing to note is that the child elements have overflowed the parent. Because the parent has a width of 600px and each child is 200px wide plus margin, padding, and border, their combined box widths exceed the width of the parent and overflow the container. This behavior is expected, but, in general, it’s not really what you want. To fix this, you need the new properties in the box-* family, which I’ll introduce throughout the rest of this chapter—after a brief digression about browser differences.

The box Value in Firefox

In the introduction, I mentioned that the Firefox implementation is somewhat problematic. This first example clearly highlights one of those problems. The example shown in Figure 15-1 demonstrates how the code displays in WebKit browsers. Have a look at the same code rendered in Firefox in Figure 15-2.

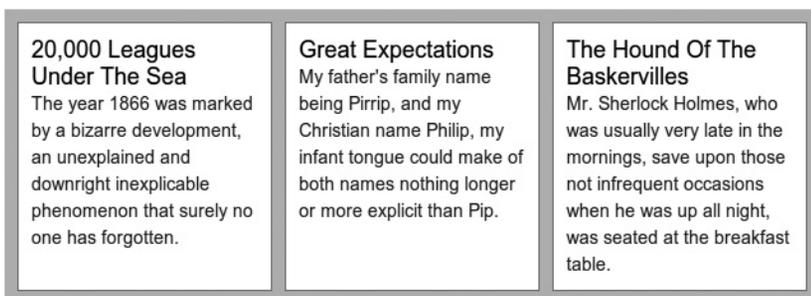


Figure 15-2: The same code from Figure 15-1 but viewed in Firefox

Here, you can see the parent element has expanded to 640px. In the Firefox implementation, if the parent element has a width value that is auto or less than the children’s width combined, that value is ignored and the parent is resized to accommodate all of its children. This is not what we want; if we’ve set a width value for the parent element, we want that width to be respected.

I don't believe either WebKit or Firefox have done this *incorrectly* as such. The module isn't clear about which of these methods is right. But I would say that the WebKit implementation respects the values we assign to elements and so is more predictable and thus easier to work with. Throughout the rest of this chapter, I'll show examples as they appear in WebKit, making a note of where Firefox's implementation differs and, if possible, how to work around it.

Inline Boxes

Other than the effect it has on its children, an element with a display value of `box` acts in the same way as an element with a value of `block`, generating a line break before and after itself. But just as `block` has a counterpart value of `inline-block`, so `box` has a counterpart value of `inline-box`:

```
E { display: inline-box; }
```

As you may be able to deduce, this value triggers the Flex Box layout but makes the element run inline in the document flow, without creating a line break. Firefox's `-moz-box` implementation is actually incorrect and behaves like `-moz-inline-box`, so I can illustrate the difference using this code:

```
div {  
  display: -moz-box;  
  display: -webkit-box;  
}
```

You can see the result in Figure 15-3. The two blocks on the left show how they appear in Firefox—they're inline with each other, so they display horizontally. The two blocks on the right illustrate how they appear in WebKit—they are correctly displayed as block elements so display vertically.

Unfortunately, I haven't found a way to change this behavior in Firefox, so if you want your elements to display in a block flow using that browser, consider using clearing elements:

```
br.clear { clear: both; }
```

I won't use `inline-box` for any of the examples in the rest of this chapter, but now you know that the possibility to do this exists.



Figure 15-3: Two elements with the display value of `box` displayed incorrectly (similar to `inline-block`) in Firefox (left) and correctly (similar to `block`) in WebKit (right).

Making the Boxes Flexible

In Figure 15-1, the child elements overflowed their parent. Now let's see how to make them fit by using Flex Box and a new family of properties, which I'll refer to as the `box-*` properties. The first of the family, `box-flex`, is the key to the Flex Box layout method. Here's the syntax:

```
E { box-flex: number; }
```

According to the spec, the *number* value can be either a whole number or a decimal fraction. At this time, however, Firefox only supports whole numbers, so I'll stick with whole numbers for the following examples. This value acts as a ratio when resizing child elements within their parent—I'll explain that in more detail as I go along. The default value is 0, which results in the layout you saw in Figure 15-1. Any value other than 0 will be used in the calculation to distribute the child elements into the width (or, as you'll see later, the height) of their parent. Showing this is much easier than explaining it, so let's take a look at an example. This example is similar to the one shown in Figure 15-1, but I added the `box-flex` property to the child elements:

```
#flexbox-holder {  
  display: box;  
  width: 600px;  
}  
div.flex {  
  box-flex: 1;  
  width: 200px;  
}
```

Figure 15-4 shows the result as seen in WebKit browsers.

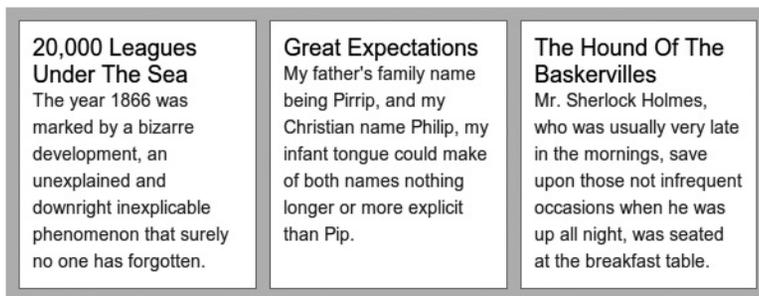


Figure 15-4: Child boxes resized dynamically to fit the parent

All three child boxes fit comfortably inside the parent; their width has been reduced to 165px so their combined values equal the width of the parent. How did this happen? As I mentioned, the value of `box-flex` actually represents a ratio to distribute the boxes within their parent. The value of 1 is not important; what's important is the fact that all the elements have the same value.

In this example, the parent element has a width of 600px; the child elements have a width of 200px, 20px of horizontal padding, and a 2px horizontal border; and the first and third elements also have 20px horizontal margin. By setting their `box-flex` value to any other value than 0, you trigger them to be flexible; although the border, margin, and padding values are always respected, their width values will be dynamically altered so they fit inside the parent.

In Figure 15-4, the children are wider than the parent, so the width is reduced in accordance with the `box-flex` value of each child. To calculate the amount of reduction, you first subtract the combined border, margin, and padding values (6px, 40px, and 60px, respectively) from the width of the parent (600px), which leaves 494px to divide between the child elements. Each has a `box-flex` value of 1, which means 494px is divided equally between the three child elements, giving each box a (rounded up) value of 165px.

`box-flex` also works the other way around: As well as reducing width to fill a parent, `box-flex` increases the width of the child elements if their combined value is less than their parent's width. Take a look at this example:

```
#flexbox-holder {
  display: box;
  width: 600px;
}
div.flex { width: 150px; }
```

Here, the combined width of the children is less than that of the parent. You can see the empty space in Figure 15-5.

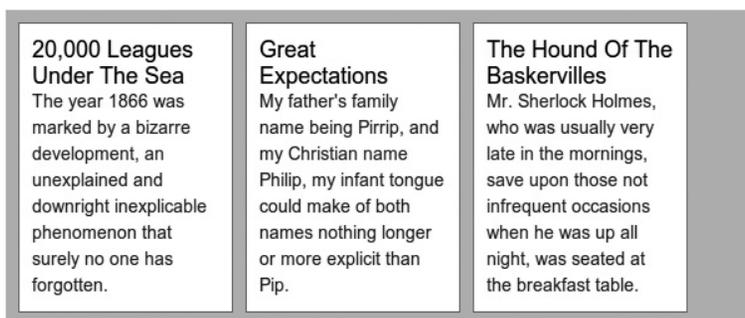


Figure 15-5: Child elements whose combined width is less than that of the parent

Now let's apply the `box-flex` value as before:

```
#flexbox-holder {
  display: box;
  width: 600px;
}
div.flex {
  box-flex: 1;
  width: 150px;
}
```

The same calculation used previously applies here: The parent element has a width of 600px, and the combined border, margin, and padding values of the children are 106px, leaving a difference of 494px, which is divided three ways to produce a width value of 165px—only this time the boxes' width increases instead of decreases. To see the result, look again at Figure 15-4—the output of this example is identical to the last one!

Unequal Ratios

A ratio value of 1:1:1 is the same as a ratio value of 3:3:3 or 5.5:5.5:5.5 or 100:100:100, so if all child elements have the same `box-flex` value, it doesn't actually matter what that value is. But what happens when the values are not equal?

This next code example is similar to the one in the previous section, but the width value of the child elements is reduced and the element with the class name `flex-two` has an increased `box-flex` value:

```
#flexbox-holder {
  display: box;
  width: 600px;
}
div.flex {
  box-flex: 1;
  width: 100px;
}
div.flex-two { box-flex: 3; }
```

The same calculations run on these elements, but this time the 494px difference is distributed between the children using the ratio 1:3:1. Therefore, the width of the element with a `box-flex` value of 3 is increased by 3px for each time that the other two elements are increased by a width of 1px. You can see the result of this in Figure 15-6.

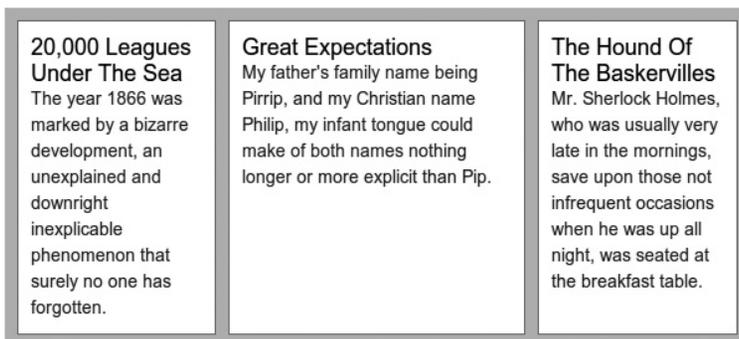


Figure 15-6: The result of using different `box-flex` values

Recall that in Figures 15-4 and 15-5 each element had a width of 165px (494px divided by 3). In Figure 15-6, the two elements with a `box-flex` value of 1 have a width value of 139px, whereas the element with a `box-flex` value of 3 has a width of 217px—all due to the changed distribution ration of 1:3:1.

Zero Values and Firefox Layouts

Setting a value of 0 for `box-flex` (or not actively specifying a value) means the element stays inflexible—that is, it retains its original dimensions. Or, at least, that’s what it’s *supposed* to do. Unfortunately, 0 has somewhat of a strange side-effect in Firefox in that it makes an element act as if it uses the Quirks Mode box model (see <http://www.quirksmode.org/css/quirksmode.html> if you need a reminder)—the element’s specified width value applies to the entire box, including its border and padding. Take a look at this code:

```
#flexbox-holder {  
    display: box;  
    width: 600px;  
}  
div.flex {  
    box-flex: 0;  
    width: 150px;  
}
```

I used this code in Figure 15-5, so the result should be the same as what’s shown there. But the output in Firefox is somewhat different, as you can see in Figure 15-7.

A notably larger amount of space remains at the right of the parent because each of the children has a total box width of 150px, which is made up of a 2px border, 20px padding, and a 128px width. Now, I can’t see anywhere in the specification that says this shouldn’t be the case, but the result doesn’t seem at all logical to me.

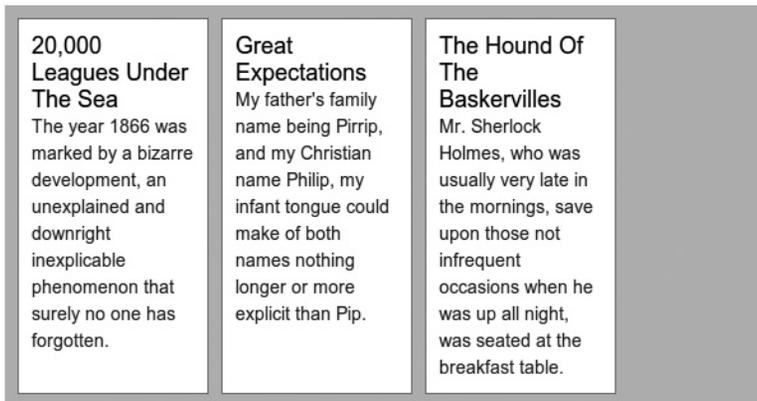


Figure 15-7: Child elements with a `box-flex` value of 0 displayed in Firefox

Unfortunately, I’ve been unable to find a solution to this; you simply have to factor it in when planning your website layouts until the Firefox developers resolve it.

Grouping Flexible Boxes

As well as individual boxes being resized according to their `box-flex` values, you can also create groups of boxes that will be resized jointly using the `box-flex-group` property. This property has the following syntax:

```
E { box-flex-group: number; }
```

In the case of this property, the *number* value is a single integer that creates a numbered group; all elements with the same value are considered part of the same group. The default value is 1, so unless otherwise specified, all elements belong to this group. When the elements are resized to fit their parent, all elements in the same group are given the same size. Groups with lower numbers are given priority.

Before I give some examples, a quick aside about browser support: Firefox and the latest preview version of IE9 have no implementation of the property at all, and WebKit's doesn't work according to the specification. In WebKit, all elements in the group with the lowest `box-flex-group` value are considered flexible, and their individual `box-flex` values apply; all elements in other numbered groups are not flexible, meaning they act as if they have a `box-flex` value of 0.

Keep this in mind as I step through a simple side-by-side demonstration. I have two flexible box parent elements with three children each; the first child of each parent has a `box-flex` value of 2, whereas all the other children have a `box-flex` value of 1. All of the other properties have the same values except for the `box-flex-group` property.

Here's the code:

```
div[class^='box-holder'] { display: box; }
div[class^='flex'] {
  width: 60px;
  box-flex: 1;
}
div.flex-one { box-flex: 2; }
❶ .box-holder-one .flex-one, .box-holder-one .flex-two { box-flex-group: 2; }
❷ .box-holder-one .flex- three { box-flex-group: 3; }
❸ .box-holder-two .flex-one, .box-holder-two .flex-two { box-flex-group: 3; }
❹ .box-holder-two .flex- three { box-flex-group: 2; }
```

And here's the relevant markup:

```
<div class="box-holder-one">
  <div class="flex-one">...</div>
  <div class="flex-two">...</div>
  <div class="flex-three">...</div>
</div>
<div class="box-holder-two">
  <div class="flex-one">...</div>
  <div class="flex-two">...</div>
  <div class="flex-three">...</div>
</div>
```

You can see the results in Figure 15-8.



Figure 15-8: Two examples using different *box-flex-group* values

In the top example, the first and second boxes (❶) have a *box-flex-group* value of 2, and the third (❷) has a value of 3. The group with the lower value has priority, so the third element is considered inflexible and retains its original size. The first and second elements are flexible and so are resized dynamically with the space distributed 2:1 in favor of the first box.

The elements in the bottom example have the values reversed; now the first and second elements (❸) are not flexible, so the *box-flex* value is ignored, whereas the third (❹), which is in the group with the lower *box-flex-group* value, is flexible and fills the remaining width.

Changing Orientation

When you begin to use this layout method, the boxes with a set *box-flex* value automatically fill the horizontal length—the width—of their parents, as you can see in all of the examples used in this chapter so far. But you can change this behavior using the next new property: *box-orient*. Here's the syntax:

```
E { box-orient: keyword; }
```

keyword has four possible values: *block-axis*, *inline-axis* (the default), *horizontal*, or *vertical*. The first two values, *block-axis* and *inline-axis*, depend on the writing mode of the web page; as this book is using English, which is a left-to-right language, *block-axis* and *inline-axis* are interchangeable with *vertical* and *horizontal*, respectively. I'll use those values throughout this section. Here's a quick demonstration of the difference between the two:

```
div[class^='box-holder'] { display: box; }  
.box-holder-one { box-orient: vertical; }  
.box-holder-two { box-orient: horizontal; }  
div[class^='flex'] { box-flex: 1; }
```

I applied this to the same markup used in Figure 15-8, and you can see the result in Figure 15-9.

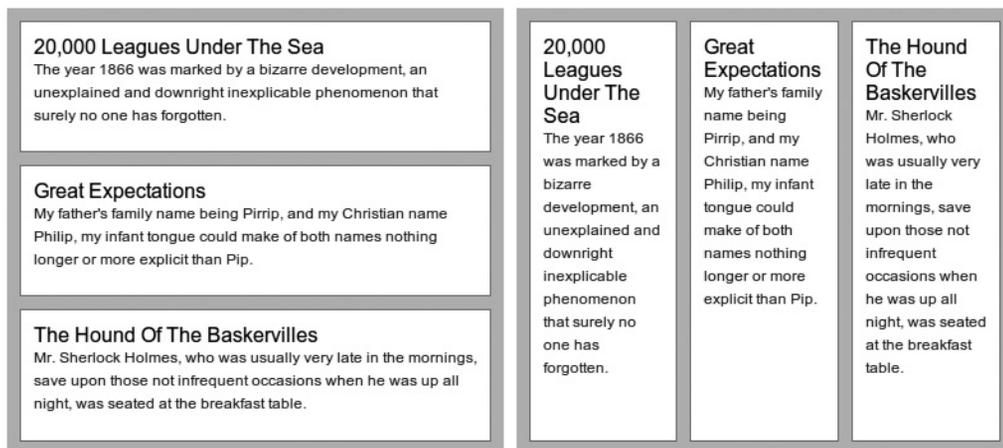


Figure 15-9: Comparing values for the `box-orient` property: vertical (left) and horizontal (right)

The three elements on the left have a `box-orient` value of `vertical` so are resized to fill the parent's height, whereas the three on the right have a value of `horizontal` so they fill the parent's width.

Changing the Order of Flexible Boxes

Flex Box has another big advantage: You can easily change the order in which the boxes display on the page, regardless of their order in the DOM, without using JavaScript or positioning tricks. You can perform this reordering in two ways: by reversing the order or by organizing elements into ordinal groups.

Reversing the Order

By default, flexible box elements display in the order in which they're coded in the document—but you can overrule this behavior using the `box-direction` property. `box-direction` is applied to the parent element and has the following syntax:

```
E { box-direction: keyword; }
```

The `keyword` value can be either `normal`, which is the initial or default value, or `reverse`. Given a value of `reverse`, all of the flexible box elements display in the opposite order to their position in the document: The last element displays first and vice versa. To see this in action, let's duplicate the markup from one of the examples shown in Figure 15-8 but use a different value for `box-direction`.

Here's the CSS:

```
div[class^='box-holder'] {  
    display: box;  
    box-orient: vertical;  
}  
.box-holder-one { box-direction: normal; }  
.box-holder-two { box-direction: reverse; }  
div[class^='flex'] { box-flex: 1; }
```

You can see the result in Figure 15-10.

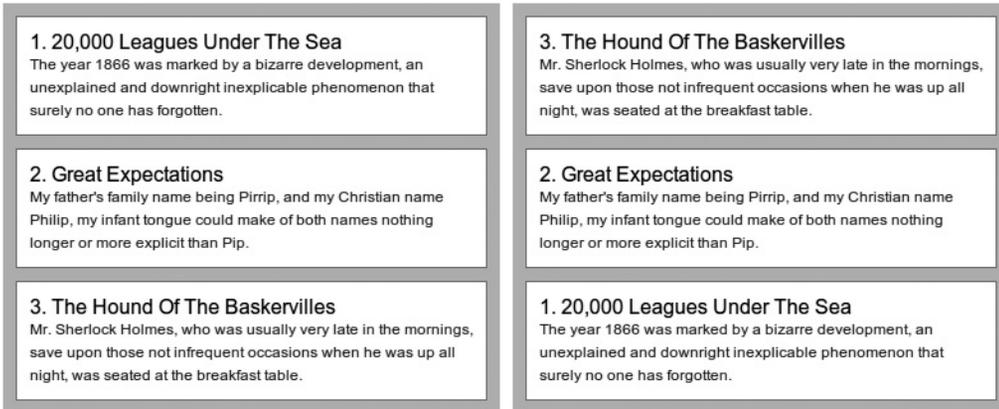


Figure 15-10: Comparing values for the `box-direction` property: *normal* (left) and *reverse* (right)

I added numbers to the titles to more clearly show the difference in ordering. The three boxes in the left column have a `box-direction` value of `normal` and so display in the order in which they appear in the markup. But the boxes in the right column have a value of `reverse`, so they appear in reverse order: The third box is displayed first, the second box remains in place, and the first box is displayed last.

Further Control over Ordering

If you want even finer control over the order of box elements, you can use the `box-ordinal-group` property, which is applied to the child elements and has this syntax:

```
E { box-ordinal-group: number; }
```

The *number* value is a positive integer with a default value of 1. The number provided as a value sets the order that the box elements are displayed in, from low to high, regardless of their position in the markup. For example, an

element with a `box-ordinal-group` value of 2 displays before an element with a value of 3. If multiple elements are given the same value, they form a group; elements within that group display in the order in which they appear in the markup but still before or after elements with a different number value.

To show you how this works in practice, let's return to the markup from the example in Figure 15-8 but this time show the effects of different values for `box-ordinal-group`. Here's the CSS:

```
div[class^='box-holder'] {  
    display: box;  
    box-orient: vertical;  
}  
❶ .box-holder-one .flex-one { box-ordinal-group: 2; }  
❷ .box-holder-one .flex-two { box-ordinal-group: 3; }  
❸ .box-holder-one .flex-three { box-ordinal-group: 1; }  
❹ .box-holder-two .flex-one { box-ordinal-group: 2; }  
❺ .box-holder-two .flex-two, .box-holder-two .flex-three { box-ordinal-group: 1; }  
div[class^='flex'] { box-flex: 1; }
```

You can see the results illustrated in Figure 15-11.

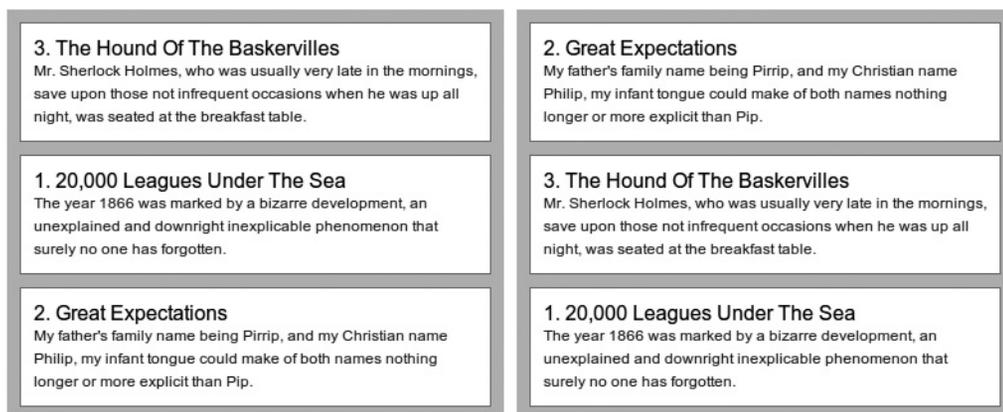


Figure 15-11: Two examples using different values for the `box-ordinal-group` property

In the examples in the left column, I set a unique value for `box-ordinal-group` on each element (❶,❷,❸), which displays them in the order assigned: 3, 1, 2. In the right column, I gave the first element (❹) a value of 2 and the second and third elements (❺) a value of 1. Therefore, the first element displays after the second and third elements. The second and third elements display in the order in which they appear in the document. The result is that the elements display in the order 2, 3, 1.

Alignment

Depending on the properties assigned to your box elements, you can quite often end up with boxes that don't stretch to fill the entire width or height of the parent and, as a result, empty spaces. Whether that empty space appears horizontally or vertically depends on the parent's `box-orient` value, but where that space appears in relation to the children can be set on the parent by using the `box-align` property. Here's the syntax:

```
E { box-align: keyword; }
```

keyword has a number of possible values. The default is `stretch`, which gives you the behavior you've already seen in this chapter, increasing the element's height or width to fill its parent. The `start` keyword places the element at the left of its parent if `box-orient` is set to `horizontal` or at the top if set to `vertical`; similarly, the `end` keyword places the element to the right if `horizontal` and to the bottom if `vertical`. If the value is `center`, the element is given equal space on both sides either horizontally or vertically, again depending on the `box-orient` value.

NOTE *The `box-align` property works on the opposite axis from its orientation. If `box-orient` is set to `horizontal`, `box-align` will distribute the space on the vertical axis and vice versa.*

I'll demonstrate some of the possibilities in the next example, using the following code:

```
div[class^='box-holder'] { display: box; }
❶ div.box-holder-one { box-align: center; }
❷ div.box-holder-two { box-align: start; }
❸ div.box-holder-three, div.box-holder-four { box-orient: vertical; }
❹ div.box-holder-three { box-align: end; }
❺ div.box-holder-four { box-align: center; }
div.flex { box-flex: 1; }
```

The results are illustrated in Figure 15-12.

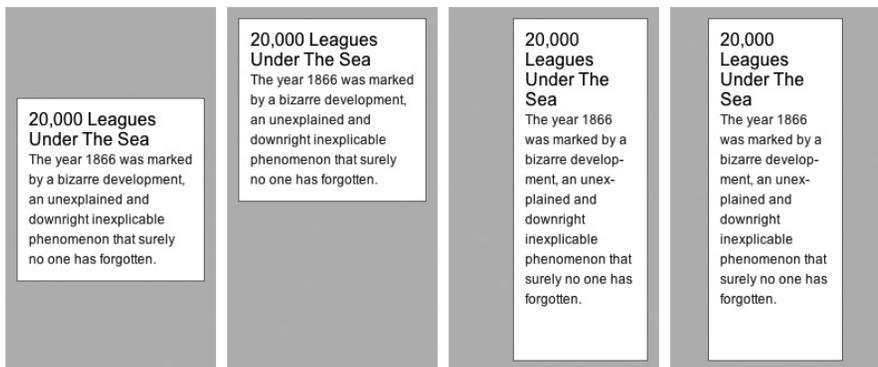


Figure 15-12: Different values for the `box-align` property

Since we haven't specified a `box-orient` value for the first (far left) example (❶), it uses the default value of `horizontal`, and the `box-align` value of `center` gives the element equal space above and below and positions it in the vertical center of its parent. The second (left) example (❷) is also oriented horizontally but has a `box-align` value of `start`, placing it at the top edge of its parent. For the next two examples the `box-orient` property is set to `vertical` (❸); the third (right) example (❹) has a `box-align` value of `end`, moving the element to the right edge of its parent, while the final (far right) example (❺) also has a value of `vertical` for `box-orient` but a value of `center` for `box-align`, positioning it horizontally within its parent.

One further value is mentioned in the specification, which is `baseline`. What `baseline` *should* do is align all the child elements by their baseline and then align the element with the greatest height to the top of the parent. However, try as I might I can't get this to display as intended—`baseline` behaves exactly the same as the `start` keyword value. Either this value has not been correctly implemented or I've misunderstood it entirely!

Same-Axis Alignment

I mentioned in the last section that the `box-align` property works on the opposite axis to its orientation, but what do you do if you want to distribute space on the same axis? For this, you have the `box-pack` property. Again, you also apply it to the parent element. Here's the syntax:

```
E { box-pack: keyword; }
```

The *keyword* values are very similar to those for `box-align`: `center` distributes the space on either side of the child elements (either horizontally or vertically depending on the `box-orient` value) and `start` and `end` add space either after or before the elements (respectively). One new keyword value is `justify`, which is used only when two or more child box elements are present. `justify` aligns the first element to the start of the parent and the last to the end and adds all the extra space in between them.

I'll illustrate some possible combinations using this code:

```
div[class^='box-holder'] { display: box; }
❶ div.box-holder-one { box-pack: center; }
❷ div.box-holder-two, div.box-holder-three { box-pack: end; }
❸ div.box-holder-three, div.box-holder-four { box-orient: vertical; }
❹ div.box-holder-four { box-pack: center; }
div.flex { box-flex: 0; }
```

You can see the results in Figure 15-13.

The first and second examples have the default `box-orient` value of `horizontal`, and the third and fourth examples have the value of `vertical`. In the first example (❶), the `box-pack` property has the value `center`, so the horizontally oriented element has the extra space distributed on either side of it on the horizontal axis. The second (❷) and third (❸) examples both have a

box-pack value of end, but each is oriented to a different axis: For the second example (❷), the space is to the child's left; and in the third example (❸), the space is above the child element. The final example (❹) has two child elements, and the parent has a box-pack value of justify, so the first child is at the top of the parent and the second at the bottom.

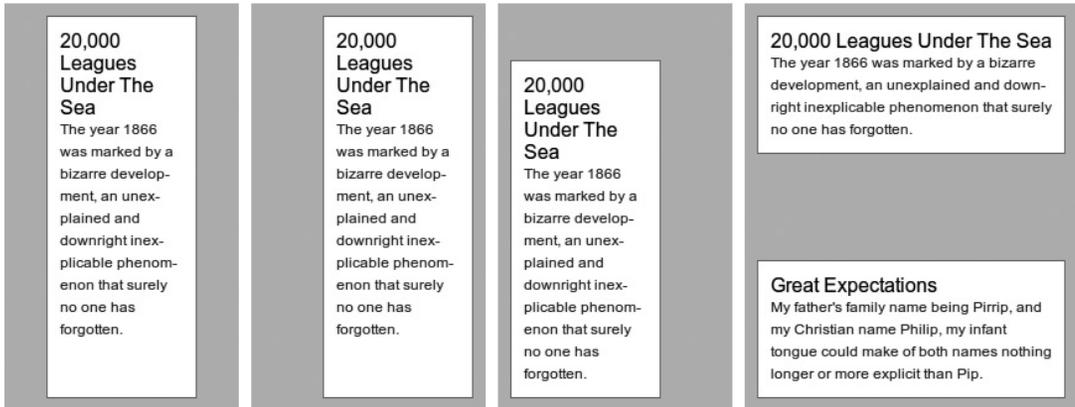


Figure 15-13: Different values for the box-pack property

This property can also be used to perform a function that has been difficult to do for far too long: centering an element inside a parent on both axes. Until now, you had to perform various tricks to do this, such as combining positioning with negative margins:

```
div {
  height: 100px;
  left: 50px;
  margin: -50px 0 0 -50px;
  position: absolute;
  top: 50%;
  width: 100px;
}
```

Using the Flex Box layout, you can do this much more easily, without needing to specify heights or calculate margins:

```
div.box-holder {
  display: box;
  box-align: center;
  box-pack: center;
}
div.flex { box-flex: 0; }
```

The element has the value of center for both `box-align` and `box-pack`, meaning all the extra space around it is distributed equally on both axes, centering the child both horizontally and vertically. You can see this in action in Figure 15-14.

Multiple Rows or Columns

The final new property is not currently implemented in Firefox, and although this property is listed as being implemented in WebKit, rigorous testing seems to show that it doesn't work in this browser at all. As the property could be fixed or implemented by the time you read this, I'll talk through it anyway.

The property is `box-lines`, and it deals with the potential issue of a row (in horizontal orientation) or column (in vertical orientation) of flexible children exceeding the dimensions of the parent. Here's the syntax:

```
E { box-lines: keyword; }
```

The *keyword* value can be single or multiple. The default value is single and declares that only one row or column can continue outside the bounds of the parent—subject to the value of the `overflow` property. The alternative value is multiple. If only one element exceeds its parents' dimensions, this value is ignored and all the elements are resized to fit; however, if multiple elements exceed the parent dimensions, they will be moved to a subsequent row (or column) if the dimensions allow it.

As I mentioned, `box-lines` currently remains unimplemented. How this will work in practice remains to be seen.

Cross-Browser Flex Box with JavaScript

Although Flex Box is only currently implemented in Firefox and WebKit, you can get it to work across all browsers by using a neat little piece of JavaScript called Flexie (<http://www.flexiejs.com/>). Installing Flexie is very easy. You need to use it in conjunction with another JavaScript library (any one will do; in this example, I'm using jQuery, which you can get at <http://www.jquery.com/>). Download the files and link to them in your document like this:

```
<script src="jquery.js"></script>  
<script src="flexie.js"></script>
```

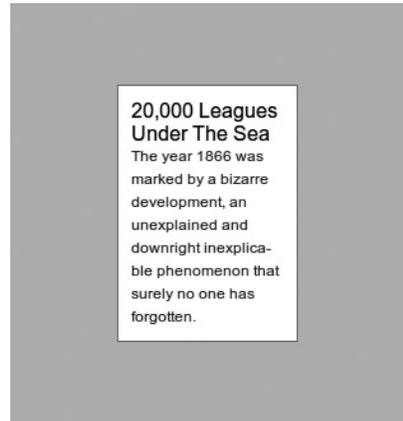


Figure 15-14: An element centered on both axes using `box-align` and `box-pack`

To get the JavaScript to work with your code, you just add the nonprefixed properties after the prefixed ones in your CSS:

```
E {  
  -moz-box-flex: 1;  
  -ms-box-flex: 1;  
  -webkit-box-flex: 1;  
  box-flex: 1;  
}
```

The script checks to see whether your browser supports Flex Box natively and replicates its effects if it doesn't. The developer has also put together a very useful test page (<http://www.flexiejs.com/playground/>) where you can alter various Flex Box values and see the results updated on the fly.

Stop the Presses: New Syntax

As I was finalizing this chapter, the W3C's CSS Working Group announced that the next draft of the specification will feature substantial alterations. These alterations include the following:

- The value for `display` will change from `box` to `flexbox`. All properties listed in this chapter that begin with `box-*` will now be changed to `flex-*`, so `box-align` becomes `flex-align`, `box-pack` becomes `flex-pack`, and so on.
- `box-direction` and `box-orient` will be combined into a single property: `flex-direction`.
- `box-flex` will be split into two properties: `flex-grow` and `flex-shrink`.
- The `box-ordinal-group` will be renamed either `flex-index` or `flex-order`.

At the time of writing, these changes have not yet made it into the draft specification, much less into any browser, so all of the properties included in this chapter will continue to operate as explained. If you want to see the new syntax, it currently resides at <http://dev.w3.org/csswg/css3-flexbox/>.

Summary

The Flexible Box Layout Module currently has Working Draft status, and its implementation remains patchy. Many inconsistencies need to be resolved—even in WebKit browsers, which currently lead Firefox in terms of these properties. But I think Flex Box is a quite elegant and logical proposal that provides a solution to several common problems in website layouts. Once the implementation issues are resolved, I can see Flex Box being adopted pretty quickly. I know the Firefox team is keen to bring its execution of this module into line with the WC3 proposal, so you could see this module moving ahead fairly quickly in the near future.

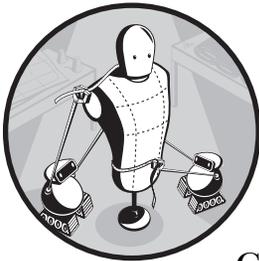
Implementation in IE9 remains limited at the time of writing this book, but I'm hopeful that the final release version of the browser will offer full support, which would be a big step toward being able to use the Flexible Box Layout and freeing ourselves from the tyranny of floats!

Flexible Box Layout: Browser Support

	WebKit	Firefox	Opera	IE
Flexible Box Layout	Yes (with prefix)	Yes (with prefix)	No	No (possibly in IE9, with prefix)

16

TEMPLATE LAYOUT



The next new layout proposal—the Template Layout Module—was authored by Bert Bos, one of the authors of the original CSS specification and, therefore, someone who should really know what he’s talking about! The Template Layout Module (<http://www.w3.org/TR/css3-layout/>) allows for the declaration of columns and rows, similar to the HTML table system.

This module is somewhat unique in that, although not currently implemented in any browsers, you can still use it! Thanks to developer Alexis Deveria, you can use a very clever JavaScript function that simulates the effects of the Template Layout Module by interpreting the CSS properties and values. I encourage you to visit the website (<http://code.google.com/p/css-template-layout/>) to read more about this fantastic tool.

The specification still has Working Draft status (last updated in April 2010), and none of the major browser makers seem to be planning for implementation in the near future, so the module will not be in widespread use anytime soon. But with the help of the JavaScript tool, you can practice using the Template Layout Module and, if you so desire, provide some helpful feedback that might move it toward implementation in the long term.

Setting Up the JavaScript

As I mentioned in the introduction, to emulate the Template Layout Module, you need to use Alexis Deveria's JavaScript tool. This tool uses the jQuery library (<http://www.jquery.com/>), which I'm sure many of you are familiar with. In the head of your page, you use script tags to link to first jQuery and then to Alexis's CSS Template script:

```
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.5/jquery.min.js"></script>
<script src="http://css-template-layout.googlecode.com/files/jquery.tpl_layout1.1.6.js"></script>
```

NOTE *The version numbers in this code—1.x for jQuery and 1.1.6 for the Template Layout—are the most up to date as I write (February 2011), but you may need to update the version numbers by the time you read this.*

After linking to the script, you run the JavaScript functions when the page is ready. You do this with jQuery's `ready()` event, which calls the `setTemplateLayout` function:

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

This code is what I'll use for my examples as all of my CSS will be embedded in the page. If you want to use an external stylesheet, however, you must supply a path to the document as an option of the function:

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

Many further options are available; I advise you to read the supplied documentation (<http://code.google.com/p/css-template-layout/>) to see what else is possible with this fine script. Now that the setup is complete, let's move on and see the new module in action.

Using position and display to Create Rows

The Template Layout method requires that you set at least two different properties for it to work, so I'll talk through both of these before showing the results. The two properties already exist in CSS2.1, but I'll give them new values.

The first property is `position`, which you should remember has the values `absolute`, `fixed`, `relative`, or `static`. But in this new method `position` is used a little differently. To show you how it works, I'll call on the same page elements

that I used for the demonstrations in Chapter 15: a parent element and three children. I usually show the CSS involved before the markup that you apply it to, but in this case, I'll make an exception. Here's the markup first:

```
<div id="tpl-holder">
  <div class="tpl-one">
    <h3>20,000 Leagues Under The Sea</h3>
    <p>...</p>
  </div>
  <div class="tpl-two">
    <h3>Great Expectations</h3>
    <p>...</p>
  </div>
  <div class="tpl-three">
    <h3>The Hound Of The Baskervilles</h3>
    <p>...</p>
  </div>
</div>
```

Here you have three child div elements, each with the class name `tpl-*`. To create a layout with the Template method, I assign each of them a letter of the alphabet as a value for the position property:

```
.tpl-one { position: a; }
.tpl-two { position: b; }
.tpl-three { position: c; }
```

The letters themselves are arbitrary—you can use any letter of the alphabet—although only a single letter is allowed for each element. (A keyword is also available; I'll return to it later in this section.)

So far, so meaningless. What do these letters represent? For that, you need to take a look at the other key property: `display`. Again, you should be familiar with this property; it currently permits values like `block`, `inline`, and `list-item`, and is used to set an element's box type. But in the Template method, you use it to set the order of the elements, which I assigned the letters to previously.

The value for the `display` property is a string of letters inside quotation marks; the string represents a horizontal row of elements—I'll refer to them as *row strings* throughout the rest of this chapter. For example, here are the three elements I introduced earlier laid out in a row in alphabetical order:

```
#tpl-holder { display: "abc"; }
.tpl-one { position: a; }
.tpl-two { position: b; }
.tpl-three { position: c; }
```

NOTE *The specification says only that the letters should be declared in a string, so either single or double quotation marks can be used; but the JavaScript I'm using to emulate these properties accepts only double quotation marks, so I'll use those throughout this chapter.*

To see the result of this code, take a look at Figure 16-1. Here, you can see the elements are displayed in the order of the characters I assigned to them, *ABC*, in a row—that is, horizontally—within the parent, without using any floats or other layout methods. As no width property value was set on any of the three elements, they are distributed into three equal columns whose combined width (including padding, border, and margin) equals the width of their parent.

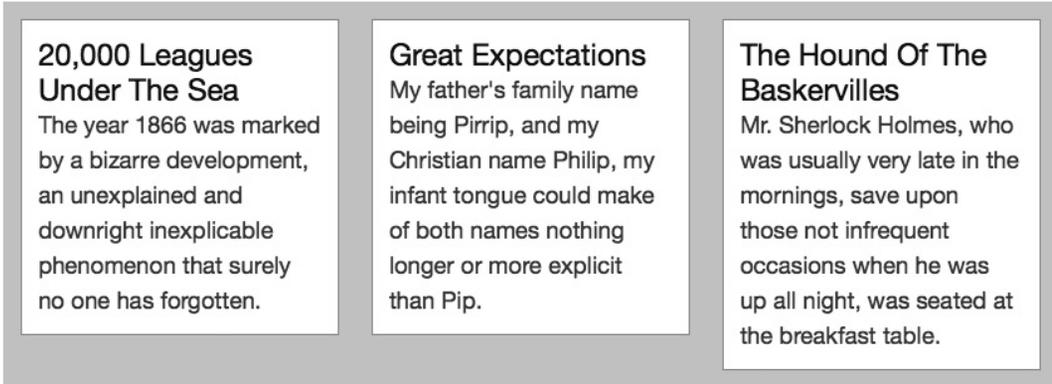


Figure 16-1: A row of elements ordered using the Template Layout method

You can just as easily display the child elements in a different order by changing the order of the letters in the string. Consider this:

```
#tpl-holder { display: "bca"; }
```

Compare the result in Figure 16-1 with the new result in Figure 16-2. You can see that the boxes have been reordered by changing the letter order in the string to *BCA*; again, the distribution of the widths is equal to the width of the parent.

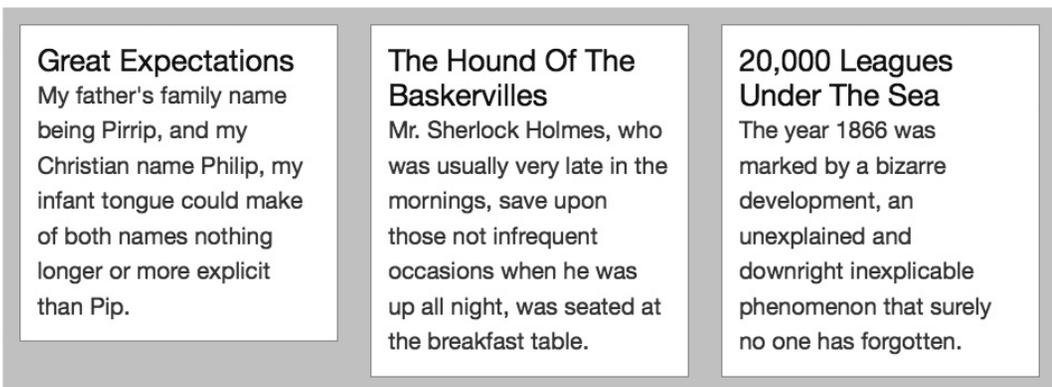


Figure 16-2: The example elements from Figure 16-1 reordered by changing the *display* property

Before moving on, I want to return to the new keyword value permitted on the `position` property, which I mentioned previously. The value is `same`, and it does what it sounds like it should do: sets the position of an element to be the same as the last specified position of a sibling element. Consider, for example, the following code:

```
.div-one { position: a; }  
.div-two { position: b; }  
.div-three { position: same; }
```

The elements `.div-two` and `.div-three` would both have the position value of `b`; this would be exactly the same as if you used this code:

```
.div-one { position: a; }  
.div-two, .div-three { position: b; }
```

Multiple Rows

So far what I've shown is fine for laying out elements horizontally in a single row, but what happens if you want to have more than one row? Doing this is actually very easy. You can have as many row-string values on the `display` property as you like; you just need to list the strings with a space separating each one. To see what I mean, take a look at this example:

```
#tpl-holder { display: "ab" "cc"; }
```

This code sets two rows; in the first are the elements with the position values `a` and `b`, each occupying half of the width; in the second row, I repeat `c` so it appears twice, matching the two elements in the row above, meaning it occupies the full width. You can see how this appears in Figure 16-3.

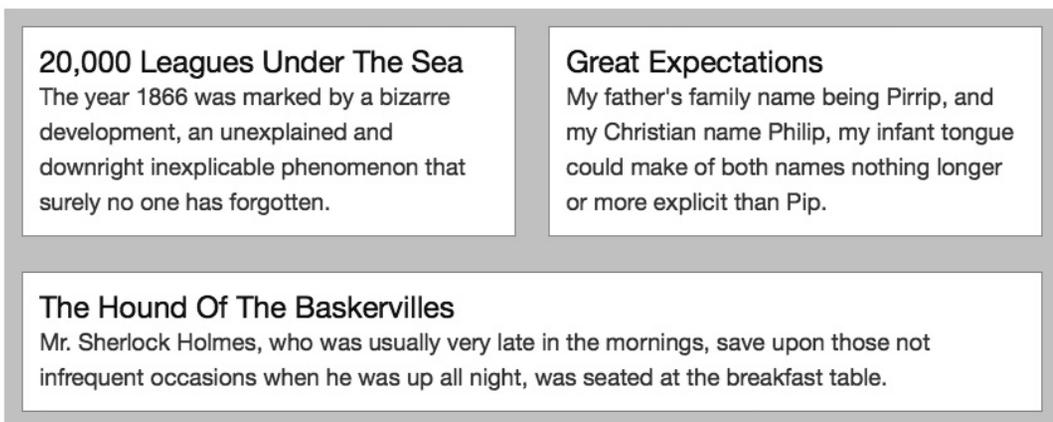


Figure 16-3: Elements laid out in two rows

The elements in positions a and b are in the first row and have a width of half that of the parent, whereas the element in position c occupies the full width of the second row.

There are two important rules to remember about row strings: The elements within each string are by default distributed proportionally, and each string must contain an equal number of characters. These two rules mean you can change the ratio of distribution by repeating letters in a string, like this:

```
#tpl-holder { display: "abb" "ccc"; }
```

Take a look at the output of this code in Figure 16-4.

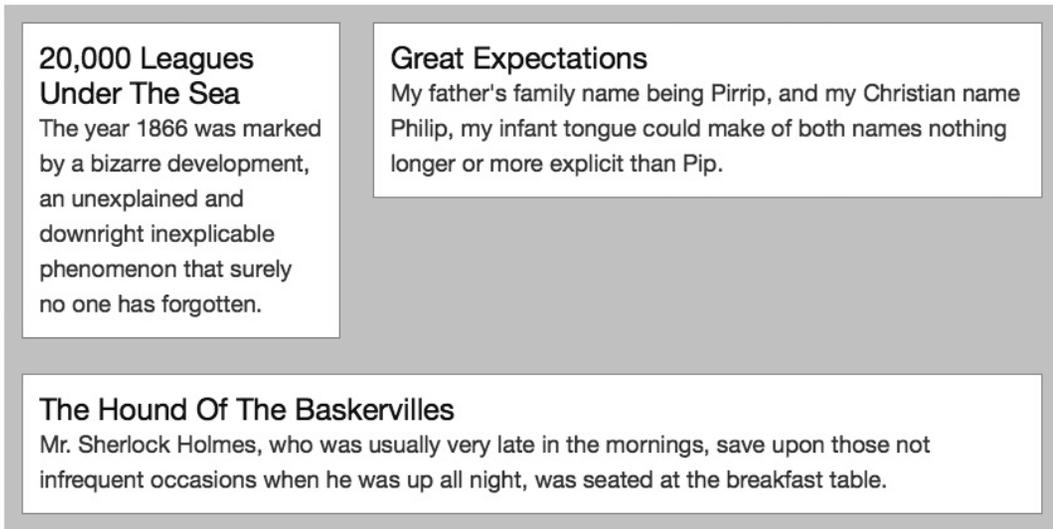


Figure 16-4: Changing the distribution of the elements by repeating characters in the `display` property

Each row now has three letters; therefore, three elements will be distributed equally in each row. But the first row has one a and two b values, so the element in position b expands to fill two-thirds of the width. In the second row, the element in position c again occupies the full width—this row also has three characters, matching the count of characters in the previous row.

Slots and the `::slot()` Pseudo-element

You can also have elements that span multiple rows, but before I show how this works, I need to introduce the concept of *slots*. The letters in a row string create an area known as a slot. All of the examples used in this chapter so far have three slots—positions a, b, and c—although the slots vary in size depending on how many places they occupy in the row strings. In the example shown in Figure 16-4, the element in position b is twice the width of the element in position a, and the element in position c is three times the width of the element in position a; however, each occupies only a single slot.

NOTE *The height of each row is calculated automatically based on the height of the elements within it—although you can adjust this manually, as you’ll see later in this chapter.*

If I remove the border from the elements in Figure 16-4 and change the colors somewhat, you’ll be able to see the slots more clearly. I’ll do this with the help of the new `::slot()` pseudo-element, which allows you to apply a (limited) range of properties directly to each slot. I’ve highlighted `::slot()` in the relevant code shown here:

```
#tpl-holder { display: "abb" "ccc"; }
.tpl-one { position: a; }
.tpl-two { position: b; }
.tpl-three { position: c; }
#tpl-holder::slot(b) { background-color: #555; }
#tpl-holder::slot(c) { background-color: #DDD; }
.tpl-two * { color: white; }
```

You can see the results illustrated in Figure 16-5.



Figure 16-5: Exposing the layout slots with the `::slot()` pseudo-element

As you can see, the element in slot b has a dark gray (#555) background-color applied to it, whereas the element in slot c has a lighter gray (#DDD) background. Below that, note that I use the element’s class name rather than the slot to apply the white text color. You might be wondering why I didn’t use this instead:

```
#tpl-holder::slot(b) * { color: white; }
```

The slot is only a construct, however; it doesn’t appear in the document tree, and isn’t actually the parent of the elements inside it. Don’t worry if you don’t understand that—all you need to know is that the `::slot()` pseudo-element can’t be used as part of a selector except to select itself.

Now that you understand slots, I can more easily demonstrate how elements can span multiple rows. The explanation is really quite simple: You use the same letter in the same position in the rows that you want the element to span. Here's a straightforward illustration:

```
#tpl-holder { display: "ab" "cb"; }
```

As you can see, the `b` is the second character in both rows; if you want to see that laid out more clearly, let's format the code in a different way:

```
#tpl-holder { display:  
  "ab"  
  "cb"  
  ; }
```

That's a perfectly valid way to write your code, by the way, so if you start designing more complex layouts, you might want to consider it. Anyway, let's see the result of this code in Figure 16-6.



Figure 16-6: A slot spanning multiple rows

The elements in positions `a` and `c` occupy half of the width of their respective rows, whereas the element in position `b` not only occupies half of the width but also spans the two rows. Spanning rows is a technique you should be familiar with from using HTML tables, and the technique works in a similar way here.

You probably noticed that the content in the element in position `b` is aligned vertically to the middle of the slot. I did this by using another of the few properties I mentioned were permitted with the `::slot()` pseudo-element:

```
#tpl-holder::slot(b) {  
  background-color: #555;  
  vertical-align: middle;  
}
```

All of the background-* properties are permitted, as well as vertical-align and overflow. Some have suggested adding box-shadow and direction to that short list, but that change hasn't been made to the module at the time of writing.

Creating Empty Slots

In addition to letters, a few special characters are permitted in the row strings. The first of these is the period (.), which is used as a spacer to create an empty slot, as in this example:

```
#tpl-holder { display: "c.b" "aab"; }
```

You can see this demonstrated in Figure 16-7.

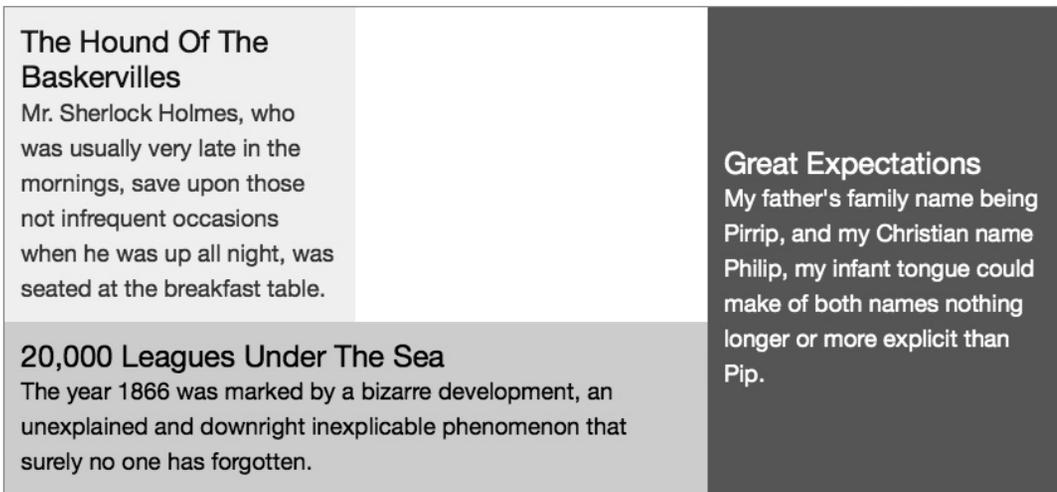


Figure 16-7: The period character creates spacer slots.

This example has two rows and is divided into three columns (each column is a third of the width of the parent). In the code, the second character in the first row string is a period, so an empty slot is created in the middle column of the first row, as shown.

Setting Height and Width on Rows and Columns

Until now I've allowed the layout algorithm to distribute the heights and widths of all of the slots equally, but the Template Layout method also gives you fine control over your layouts, permitting you to set the row height and column width explicitly. Both are easy to set, but I'll begin with the former as row height is the easier of the two to set.

By default, a row's height is set by the slots within it, and a slot is as high as the combined height of the elements contained inside it. But to overrule that and set a fixed height, you have only to specify a length value after the row string that you want to apply the height to, using a forward slash (/) to separate the two:

```
E { display: "abc" / length; }
```

Here you can see a real-world example, with the result shown in Figure 16-8:

```
#tpl-holder { display: "cb" / 200px "ab"; }
```

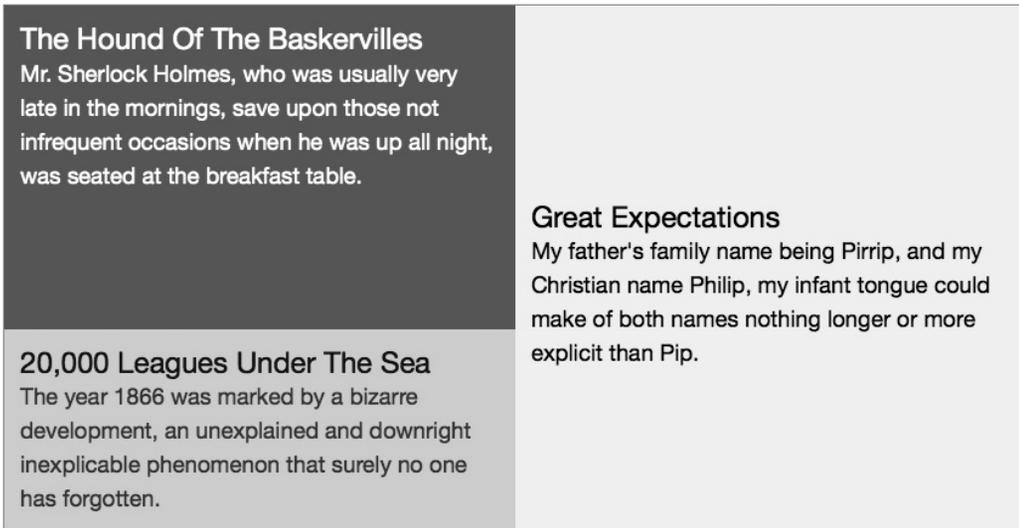


Figure 16-8: The height of the first row was set manually.

This layout has two rows and two columns—the element in position b spans both rows. After the string that lays out the first row ("cb"), you have a forward slash and the value 200px; this sets the first row to be 200px high. If this value isn't supplied, this row would be the same height as the one below it (in position c).

Setting the width of a column is only slightly more complicated. At the end of the display property, you specify a length value for each column without the forward slash:

```
E { display: "abc" length length length; }
```

The number of length values given must equal the number of columns, but if you don't want to set the value of a column explicitly you can use an asterisk (*) instead. Here's how that looks in practice:

```
#tpl-holder { display: "cb" "ab" * 33%; }
```

You can see the results illustrated in Figure 16-9.

<p>The Hound Of The Baskervilles Mr. Sherlock Holmes, who was usually very late in the mornings, save upon those not infrequent occasions when he was up all night, was seated at the breakfast table.</p>	<p>Great Expectations My father's family name being Pirrip, and my Christian name Philip, my infant tongue could make of both names nothing longer or more explicit than Pip.</p>
<p>20,000 Leagues Under The Sea The year 1866 was marked by a bizarre development, an unexplained and downright inexplicable phenomenon that surely no one has forgotten.</p>	

Figure 16-9: The width of second column was set manually.

This same markup was used for Figure 16-8, but instead of specifying the row height, here I specified the column width. After the row strings, you see two values: an asterisk and 33 percent. As there are two columns, the first uses an automated value and the second a width of 33 percent of the parent—if I hadn't specified any width, the column width would have defaulted to 50 percent.

Width Keyword Values

As well as length and percentage values, a few keywords are available for setting column widths. The first is `max-content`, which specifies the column should be at most only as wide as the widest content that it contains; its opposite is `min-content`, which indicates the column should be at least as wide as the widest content within it, but it can be wider.

The `minmax` function accepts two values—a minimum and maximum—which provide a range within which the column width may be set. You can use either length values or the `max-content`/`min-content` keywords. Here's an example:

```
E { display: "abc" * minmax(min-content, 500px); }
```

This code sets the column to be at least as wide as its content, up to a limit of 500px.

The final keyword value is `fit-content`, which is shorthand for `minmax(min-content, max-content)`—that is, make the column exactly as wide as the widest content it contains.

Setting Both Row Height and Column Width

Specifying both row height and column width in a declaration can begin to look a little confusing. What you need to remember is if you're setting a width on the last row, the first value after the forward slash is the row height and the remaining values are the column widths:

```
#tpl-holder { display: "cb" "ab" / 200px * 33%; }
```

If that's confusing, remember you can reformat your code to make it easier to see:

```
#tpl-holder { display:  
"cb"  
"ab" / 200px  
* 33%  
;}
```

This layout makes the code a little more straightforward to read, but how you lay out your code is for you to decide. However the code is laid out, the result is the same, which you can see in Figure 16-10.

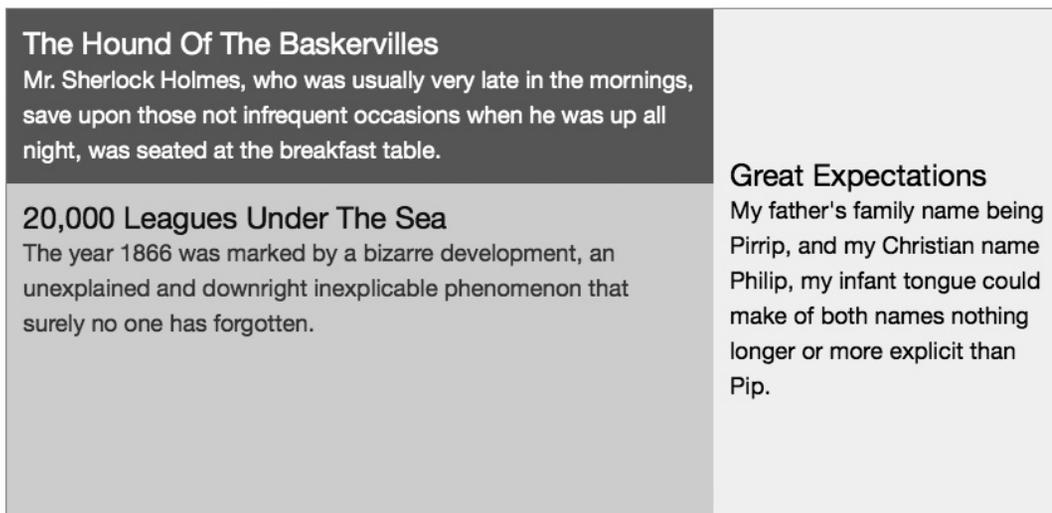


Figure 16-10: The dimensions for second row and second column were set manually.

The example in Figure 16-10 has two rows of two columns: The second row has a height of 200px, the second column has a width of 33 percent, and the first column is dynamically sized to fit the remaining width of the parent.

Default Content: The @ Sign

I mentioned earlier in this chapter, when introducing the period (.) used for spacing, that certain characters can be used in the row string. The other permitted character is the at sign (@), which is used to represent the default content in an element, by which I mean any child element that has not been assigned a slot. Here's how it looks:

```
E { display: "@"; }
```

This code simply displays the content in its natural position, which isn't perhaps very useful. But consider a more practical example, as in this next demonstration, which utilizes the example markup I've used throughout this chapter:

```
#tpl-holder { display: "a" "@"; }  
.tpl-three { position: a; }
```

You can see how this works in Figure 16-11.

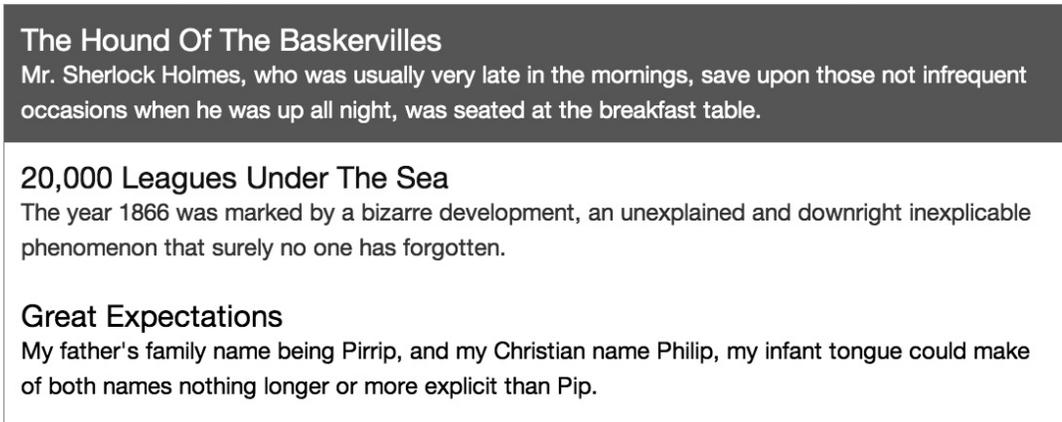


Figure 16-11: The element in position *c* has been moved above the rest of the content by using the *at* sign (@).

Here the layout is composed of two rows. The element in the first row is the element with the position value of *a*, which is the last child element in the example markup; the second row string contains the @ character, which represents everything that is not specified, so any content not inside the element in position *a* is placed in this row.

By changing the display value, I can easily rearrange the elements:

```
#tpl-holder { display: "@a" * 33%; }  
.tpl-three { position: a; }
```

Figure 16-12 shows this result.

Here, I have only a single row, and the element in position *a*, with a width of 33 percent, now sits in a column to the right of the rest of the content in the element.

WARNING *In Figure 16-12, the default content doesn't wrap and is hidden under the column on the right. I can't tell whether this error is in the specification or in the JavaScript implementation, and until browsers begin to implement this module, we probably won't know.*

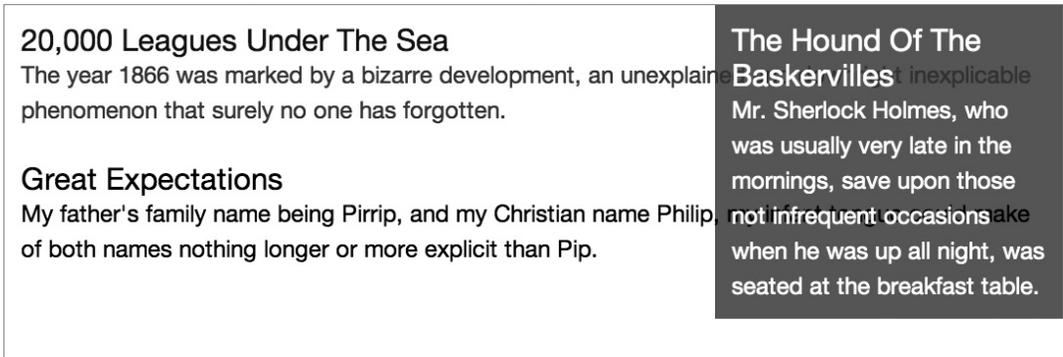


Figure 16-12: The @ sign is used again to position the element to the right of the default content.

You can only have one @ slot per layout (although the slot can span multiple rows or columns), and slots can only be rectangular in shape. The following uses are, therefore, not permitted:

```
E { display: "@.@"; }
E { display: "@." ".@"; }
E { display: "@." "@@"; }
```

In both the first and second cases, two @ slots would be required, and in the third, the @ slot would form an L shape; all of these examples are invalid.

Summary

Considering this layout method is fairly new to me, this chapter was surprisingly easy to write—not something that I’m attributing to any talent on my part but to how easy this syntax is to grasp. Every developer should be familiar with columns, rows, and spans from using HTML tables, and this syntax cleverly leverages that knowledge to become quite intuitive.

Huge thanks must go to Alexis Deveria for writing the script that made this whole chapter possible. I have no doubt his script will be referred to often when browser makers are implementing the Template Layout Module natively.

As well as the demonstrations in this chapter, you can find further examples of the possibilities available with the Template Layout Module at the sites of Alexis Deveria (<http://a.deveria.com/csstpl/>) and Neal Grosskopf (<http://www.nealgrosskopf.com/tech/thread.asp?pid=46>). Both are well worth your time if you’re experimenting.

Template Layout: Browser Support

	WebKit	Firefox	Opera	IE
Template Layout	No*	No*	No*	No*

* Not implemented natively but works using JavaScript.

17

THE FUTURE OF CSS



Almost all of the properties featured in this book so far have been implemented in at least one browser and are likely to be implemented in more in the near future as the modules mature. But you must remember: CSS3 is still in a state of flux in many areas, with a constant flow of new properties being defined and old properties being updated.

In this final chapter, we'll look at properties that have limited implementation or no implementation at all, at modules that have been written but remain in indefinite status, and at proposals for new modules that have yet to be ratified. Although you most likely can't use these properties in the next year, you may be able to use them in the years that follow.

The properties in this chapter give you the ability to perform mathematical operations and define constants and variables, letting CSS dip its toes into the waters of programming languages; they borrow from print design to create layout grids; they greatly extend the possibilities of background images—and more. Don't take for granted that you'll see any of these new

properties in your browser any time soon, if at all. A lot of negotiating and hard work needs to be done before these make it beyond the theoretical stage. But if anything in this chapter excites or enrages you, make it known publicly; the W3C pays attention to discussions by website authors—especially if constructive criticism is involved.

That said, let's take a look at the (possible) future of CSS.

Mathematical Operations

I mentioned briefly the CSS3 Values and Units Module (<http://www.w3.org/TR/css3-values/>) when introducing the `appearance` property in Chapter 10. Although the module is currently undergoing revision and so not even at Working Draft status yet, a few of its interesting new functions deserve to be highlighted. These functions are big steps forward, taking CSS beyond being a descriptive presentational language and moving it toward being a programming (or, at least, computational) language. As such, anyone with experience using JavaScript or PHP (among others) should have no trouble grasping the concepts I'm about to introduce.

Calculation Functions

In CSS2, all length values are declared and fixed, which can cause problems if you want to create layouts that mix length units. Consider, for example, an element with a width of 60 percent and `margin-right` of 10px. To float another element to its right, you would have to know the width of the parent element—to get the width of the element in question—and then calculate that width minus 10px to get the width value of the new floating element. Add to that any borders or padding the elements may have, and figuring placement out becomes tricky and time-consuming at best and, in some cases, impossible.

CSS3 introduces the `calc()` function, which aims to take care of calculations such as these, thus allowing for more dynamic, flexible layouts. Its syntax is as follows:

```
E { property: calc(calculation); }
```

I haven't specified a property in the code as `calc()` can be applied to any property that accepts length values: `border-width`, `font-size`, `height`, `margin`, `padding`, to name but a few. The *calculation* value supports simple arithmetic using five basic operators: plus (+), minus (-), multiply (*), divide (/), and mod.

NOTE *Everyone probably understands the first four operators, but mod may need some explaining. A mod value is written a mod b; basically, mod is the value that remains when a has been divided by b a whole number of times. For example: 20 mod 3 equals 2, because 20 can be divided by 3 six times, leaving a remainder value of 2 (3 × 6 = 18; 20 - 18 = 2).*

These operators can be used with length values or numbers, such as in these following examples:

```
E { property: calc(80% + 10px); }
E { property: calc(100% - 5em); }
E { property: calc(25em * 5); }
E { property: calc(75% / 4); }
E { property: calc((75% / 4) - 20px ); }
E { property: calc(150px mod 10em); }
```

In the hypothetical example I used at the beginning of this section, I could use `calc()` to account for the `margin-right` and `padding` of the first element:

```
div.one {
  margin-right: 10px;
  padding: 10px;
  width: 60%;
}
div.two { width: calc(40% - 30px); }
```

You can see the result in Figure 17-1. In the first (left) example, the width of the box is set to 40 percent, but the second box is pushed onto a new row by the extra margin and padding on the first box. In the second (right) example, the value provided to `calc()` compensates for the margin and padding, so the two boxes remain on the same horizontal line. Without `calc()` I would, as I mentioned, have to know the exact width of the parent in order to make the same adjustment—somewhat defeating the object of using flexible widths for layout.

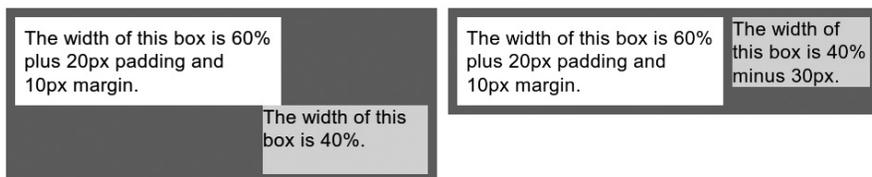


Figure 17-1: In the second example, I set the width of the right column with the `calc()` function.

You can do more than simple calculations, however, as the following example shows. Take a look at this code:

```
.calc {
  border: 2px solid black;
  margin: 10px;
  padding: 1em;
  width: 100%;
}
#calc { width: calc(100% - (2em + 24px)); }
```

The result is shown in Figure 17-2. Here, you see two identical elements, each given 100 percent width. The extra width given by the border, padding, and margin makes the example on the left too wide for its containing element, and the overflow is cropped on the right side. In the second example, however, I used the `calc()` function to subtract those values from the width, so it fits within its parent element—and what’s really useful is that I was able to mix three different length units (percentage, `em`, and `px`) in doing so.

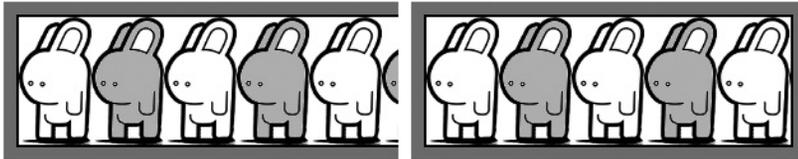


Figure 17-2: I used the `calc()` function to set the width of the example on the right so the child doesn’t overflow its parent element.

IE9 will be the first browser to implement `calc()` fully, although Firefox 4 Beta has an implementation using the `-moz-` prefix:

```
E { property: -moz-calc(calculation); }
```

Two more calculation functions can be used on elements with length values: `max()` and `min()`. Here’s the syntax:

```
E { property: max(value,value); }  
E { property: min(value,value); }
```

These functions compare the values supplied to them and apply the one that fits the stated criteria; that is, if I supplied the same two values to each, the `max()` function would apply the largest, and the `min()` function would apply the smallest. Let’s look at an example:

```
E { property: max(150%,20px); }  
E { property: min(150%,20px); }
```

These functions act as shorthand for the logical statement: “Apply the value of 150 percent or 20px, whichever is the larger/smaller.” So if you presume that the 150 percent value is equivalent to 15px, the `max()` function applies the value of 20px to the property—20px being greater than 15px—and thus, the `min()` function applies the 150 percent (15px) value.

The `max()` and `min()` calculation functions currently exist only in the Editor’s Draft of the Values and Units Module. Firefox nightly builds had implemented these functions with the `-moz-` prefix, but they seem to have been removed from later builds.

Cycle

Another common function in programming languages is *cycling*, which is a repetition process that works through a range of values. Given values *a, b, c*, the cycle would start at *a*, move to *b*, and then *c*, and then back to *a* to begin the cycle again. Not currently in the Values and Units Module, although apparently planned for inclusion, is a method for cycling in CSS. The syntax is quite simple:

```
E { property: cycle(values); }
```

The *values* are a comma-separated list of values that are permitted with the specified property—so you couldn't apply a color value to the font-size property, for example; these values are then cycled through, in order, under certain conditions. Let's look at a practical example. First here's a CSS snippet:

```
p.italic { font-style: italic; }  
em { font-style: cycle(italic, normal); }
```

And I'll apply this snippet to this example markup:

```
<p>The next word will be <em>emphatic</em>.</p>  
<p class="italic">The next word will be <em>emphatic</em>.</p>
```

In the first sentence, which can be considered “normal” circumstances, the word contained in the `em` element has a value of `italic` applied to its font-style property; the word is applying the first value from the `cycle()` notation. In the second sentence, however, the word in the `em` element has inherited the `italic` value from its parent already, so the next value in the cycle, `normal`, would be applied instead.

As I mentioned, `cycle` isn't currently in the specification but is expected to be included in the near future. Although not implemented in any browser at the time of writing, `cycle` is listed for inclusion in a future version of Firefox.

The Grid Positioning Module

In addition to the two proposed new layout methods featured previously in Chapters 15 and 16, a third method, the Grid Positioning Module (<http://www.w3.org/TR/css3-grid/>) has also been proposed—although this module hasn't currently been implemented in any browsers. The module's major advantage is that you can use it to extend other modules or stand alone. The Grid Positioning Module is made up of three core concepts: *implicit and explicit grids*, the *grid unit* (*gr*), and *extended floats*.

Implicit and Explicit Grids

Some HTML and CSS properties create natural grid structures. The most obvious example is the `table` element (or elements laid out with the `display: table` family of CSS declarations), which provides regular numbers of rows and columns. But the multi-column layout (introduced in Chapter 7) also creates a grid structure—a single row with many columns (and the gaps between columns, which are set with the `column-gap` property, also counting as columns). In the language of the Grid Positioning Module, these grids are referred to as *implicit grids*.

If elements in a page layout do not form an implicit grid, then you can impose a grid using two new proposed properties:

```
E {  
  grid-columns: length;  
  grid-rows: length;  
}
```

Grids are defined by the lines that divide them, and both of these two properties accept a list of space-separated values—either length units, percentages, or fractions (more on this soon)—which set the position of the lines that define the grid’s columns and rows. Here’s an example:

```
div {  
  grid-columns: 20% 60%;  
  grid-rows: 2em 4em;  
}
```

This example creates a 3-by-3 grid, where the column lines are placed at 20 percent and 60 percent of the grid’s width—so the first column is 20 percent wide, the second and third 40 percent wide. The row lines are placed 2em and 4em from the top of the parent—so the first and second rows are both 2em high, and the third is the remaining height of the parent. Assuming the div is 100px wide by 100px high, and that 1em is equal to 10px, you would see the grid shown in Figure 17-3.

To aid in the layout of more complex grids, the `repeat()` notation is available, which can be used to repeat values to fill the length of the parent. Here’s an example:

```
div { grid-columns: 25% repeat(1em 3em); }
```

This code creates one column that is 25 percent of its parent’s width; the remaining width is then filled with alternating columns of 1em and 3em widths.

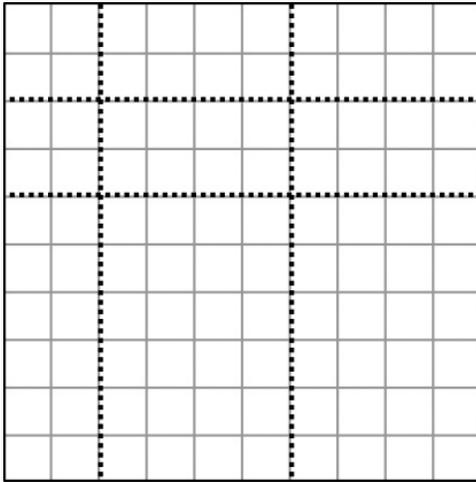


Figure 17-3: Illustration of grid lines making rows and columns

As I mentioned, you can also use the *fraction* unit. Fraction is a number followed by the unit `fr`—for example, `1fr`, `3fr`, or `4.5fr`—and is used as a ratio to divide nonallocated lengths (similar to the Flexible Box Layout, which I covered in Chapter 15). To see what I mean, take a look at this example:

```
div { grid-columns: 40px 2fr 3fr 80%; }
```

This code creates a grid with four columns. The first grid line is 40px from the left, and the last is 20 percent from the right; the third grid line is a point in between them, where the remaining width is divided by the ratio 2:3. Suppose the element is 200px wide; the first and last columns would be 40px wide (20 percent of 200px equals 40px). This leaves 120px of space divided by the ratio 2:3, meaning the second column is 48px and the third is 72px. You can see how this would appear in Figure 17-4.

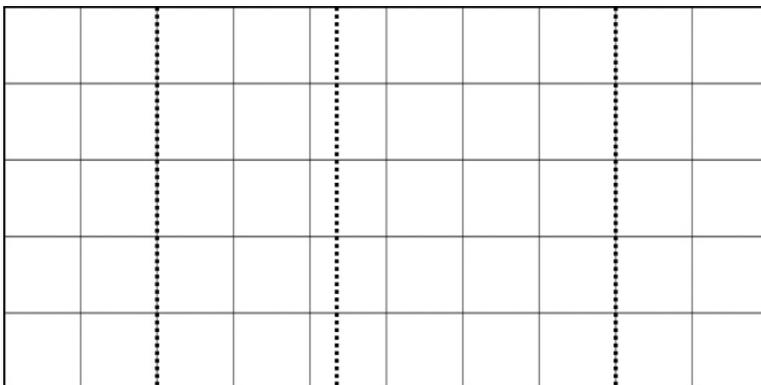


Figure 17-4: A grid created with fraction values

A grid that's formed by using these properties is, in the syntax of this module, known as an *explicit grid*.

The Grid Unit (*gr*)

Once you have your grid, whether implicit or explicit, you can begin to place elements in it. For this, you need to use the new *grid* length unit, abbreviated to *gr*. You can use *gr* on any block element property that accepts length units as values, and its value is a number that is used to measure a grid unit (or cell, if you prefer). So if you want an element to span three columns, you would use this code:

```
E { width: 3gr; }
```

You can also use the *gr* unit to make an element span rows and as an off-set for positioning, as shown here:

```
E {  
  height: 2gr;  
  width: 2gr;  
  position: absolute;  
  left: 1gr;  
  top: 1gr;  
}
```

This example has an element that spans two rows and two columns and is absolutely positioned one column from the left and one row from the top, as illustrated in Figure 17-5.

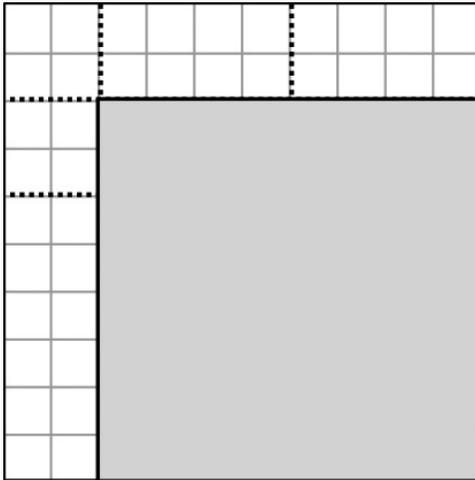


Figure 17-5: Illustration of positioning an element with the *gr* unit

Extended Floats

The final component of the Grid Positioning Module lifts a concept from the CSS GCPM (*Generated Content for Paged Media*) Module (<http://www.w3.org/TR/css3-gcpm/>), which is a bit of a mouthful. The concept it lifts is that of an extended *float* property, which accepts a wider range of values, allowing you to place an element in a grid and have content flow around it.

NOTE *I haven't covered the GCPM Module at all in this book because it deals specifically with printed or print-like matter.*

As you know, *float* currently only permits the values *left*, *right*, or *none*. But in the Grid Positioning Module, you can assign further positional values and combine those with the *gr* length unit to give you more granular control over placing an element. Consider this example:

```
E {  
  float: left bottom;  
  width: 2gr;  
}
```

Here the element would float at the bottom and to the left of the column in which it is specified, and it would span two columns.

Predicting whether this module will ever be implemented is hard. This module was authored by Microsoft staff so it has a good pedigree, but they haven't updated it since 2008 and so seems to face an uncertain future.

Extending the Possibilities of Images

CSS3 introduces a number of new methods to combat issues that have up till now been resolved by using images (rounded corners, drop shadows, and so on). Images are still an integral part of using CSS, however—from decorative backgrounds to the CSS sprites method and the like. With this being the case, that you have only really one way to set an image on a property seems a little limiting:

```
E { property: url('image-path'); }
```

The method is functional but not very flexible. A heavily in-development new specification, the Image Values and Replaced Content Module (<http://www.w3.org/TR/css3-images/>), henceforth known as the Images Module for brevity, aims to change that by extending what you can do with graphics in CSS.

Image Fallback

The first major stumbling block comes if a specified image isn't found or the file type of the image is not supported—you don't have a fallback option available to display an alternative. With the `background-image` property, you do have the option to provide multiple values, but this property, which has its drawbacks, isn't really intended to act as a fallback provider. For example, if the first-choice element has a transparent background, the second-choice element shows through from beneath the first.

The proposed solution is the new `image()` notation, which has the following syntax:

```
E { property: image('image-path' resolution or color); }
```

The first value, *image-path*, is the same as used in the `url()` notation: a path to an image file. After that, you have an optional *resolution* value and an optional keyword or with a *color* value that can be used as the background color if the image path provided is invalid. Let's see a practical example:

```
div { background-image: image('image.png' 150dpi or #F00); }
```

This code means “show the image *image.png* at a resolution of 150dpi (dots per inch), or set the background-color to #F00.” You can add further image options by listing them, comma-separated, after the image and resolution values. This solution works in a similar way to `background-image` (with multiple values) but will only display the first valid image, ignoring any subsequent items in the list.

Here's a further example showing multiple image paths:

```
div { background-image: image('image.svg', 'image.png' 150dpi, 'image.gif' or #F00); }
```

Here, I've extended the previous example so its logical statement is now “show *image.svg* as a first choice, or *image.png* at 150dpi as a second choice, or *image.gif* as a third choice, or set the background-color to #F00 if none of those images are available.”

Image Slices

Back in Chapter 8, I showed you Firefox's suggested `image-rect()` notation to define an area of an image for clipping. The CSS3 Images Module has an alternative suggestion that extends the `url()` notation. Here's the syntax:

```
E { property: url('image-path#xywh=x-co-ord,y-co-ord,width,height'); }
```

As before, the *image-path* value is the path to a valid image but followed by *#xywh* and four integer values: The first pair of those four values defines an *x*- and *y*-coordinate on the image from which the top-left corner of a rectangle will be created, and the second pair defines the width and height of that rectangle. The rectangle, of course, is the area of the image that will be displayed.

Consider this example:

```
div { background-image: url('image.png#xywh=20,10,50,100'); }
```

Here, a `div` element has the image file `image.png` set as its `background-image`. But instead of the whole image, only an area that is 50px wide and 100px high and that starts 20px from the left and 10px from the top of the image file is shown.

To further demonstrate, let's look back to Chapter 8 at the first example in Figure 8-7. In this example, I clipped an image using the `image-rect()` syntax:

```
.white { background-image: -moz-image-rect(url('bunny_sprite.png'),0,57,100,0); }
```

Here's how I would do this in the new suggested syntax:

```
.white { background-image: url('bunny_sprite.png#xywh=0,0,57,100'); }
```

I'm actually using the same coordinate values in both syntaxes, just in a different order. Neither of these syntaxes has been decided upon yet (as far as I know). My own feeling is that `image-rect()` seems more in keeping with existing CSS syntax, but the alternative suggestion is undeniably more compact.

Image Sprites

The Images Module also mentions a dedicated Image Sprites syntax, which I briefly referenced in Chapter 8. Two candidates have been proposed, and no decision has yet been made as to which will be implemented. The first proposal is for a new `@sprite` rule that defines a grid matrix that can be referred to by using a unique *id* and matrix coordinates. Here's the syntax:

```
@sprite id {  
  sprite-image: image-path;  
  sprite-offsets-x: int;  
  sprite-offsets-y: int;  
}
```

In the first line, I create a unique *id* for the sprite, which I'll use to refer to it in later rules, and in the second line, I specify the image file. The properties in the final two lines accept a space-separated list of integers, which refer to coordinates on the image and create a grid matrix.

Here's a simple example:

```
@sprite example {  
  sprite-image: image.png;  
  sprite-offsets-x: 0 10 20;  
  sprite-offsets-y: 0 10 20;  
}
```

The unique sprite identifier is `example`, and the image I'm using is `image.png`, which, for the purposes of this example, I assume is 30px wide and high. I then set three *x*-coordinates at 0, 10px, and 20px and three *y*-coordinates at the same values, which means the `example` image is divided into a matrix of nine equally sized (10px by 10px) cells, as shown in Figure 17-6.

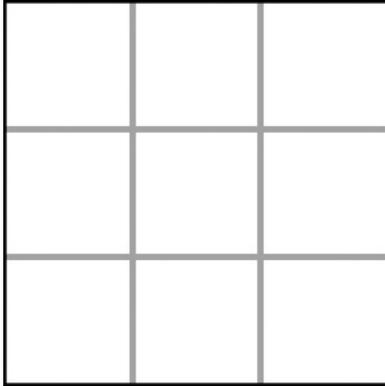


Figure 17-6: Illustration of a 3-by-3 grid matrix

The next step is to refer to the sprite *id* and matrix coordinate on the element I want to apply the sprite to:

```
div { background-image: sprite(example, 2, 2); }
```

On this `div`, I use the `sprite()` notation to refer to the sprite with the identifier of `example`, and use the portion of the image that's at the matrix coordinates 2,2. Each column and row is numbered sequentially from 0 (zero), so the first column in the first row has the coordinates 0,0. That being the case, the cell at 2,2 in the `example` matrix is at the bottom right, as illustrated in Figure 17-7. This portion of the image is displayed.

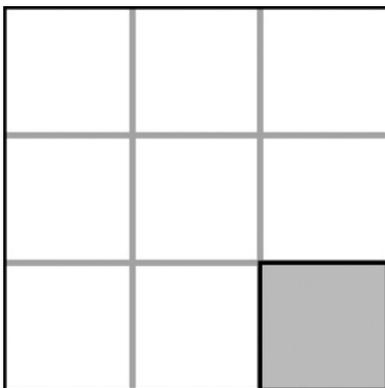


Figure 17-7: The cell located at matrix coordinates 2,2

The second proposal under consideration is somewhat less well defined but again uses an at-rule—this time the `@define` notation (which is also proposed in the CSS Variables syntax I'll introduce later in this chapter). In this proposal, all of the sprites are assigned a unique variable name using this syntax:

```
@define image-vars{
  id { rect(top,right,bottom,left) url('image-path')}
}
```

Here, you use the `@define` rule to call your image variables. Then you give each variable a unique *id* and define the area of the image to use with the `rect()` notation. The four integers represent values for the top, right, bottom, and left of the image area, all calculated in pixels from the top-left corner, in exactly the same way as the CSS2 `clip()` property (and the CSS3 `image-rect()` property shown in Chapter 8).

After that, the `url()` notation defines which image to use for this sprite. As many sprites as required can be created by repeating the whole notation in a comma-separated list:

```
@define image-vars{
  id { rect(top,right,bottom,left) url('image-path')},
  id { rect(top,right,bottom,left) url('image-path')}
}
```

Here's a simple real-world example:

```
@define image-vars{
  example { rect(10,20,20,10) url('image.png')}
}
```

This code creates a variable with the identifier `example` and clips an image that is 10px wide and 10px high, offset by 10px from the left and top of the image `image.png`—exactly the same as the example I used to illustrate the first proposal. I then refer to the *id* on the property that I want the sprite to be applied to:

```
div { background-image: var(example); }
```

The first proposal has the advantage of being more precise and less repetitive and is more suitable for slicing up a single large image. The second is more repetitive but has the advantage of reusing a lot of existing syntax and is perhaps more suitable if many different images need to be sliced. As to which will become the official proposal—I have no idea. Perhaps neither!

Grouping Selectors

Although CSS2 syntax provides for most eventualities, one of its drawbacks is the amount of repetition that can occur in stylesheets, especially for larger or more complicated web pages. For example, having a page with three different

types of list elements—ordered (ol), unordered (ul), and the new HTML5 menu element—each with nested lists inside them is not improbable. Then, as a further complication, those lists can be inside different container elements, some of which you want to select and some that you don't. After all that, to select those nested list items with CSS2, you'd have to use something like this:

```
article ul ul li, div ul ul li, ol ul li, menu ul li { property: value; }
```

And I think that's a pretty conservative example. I've seen (and used) a lot more complicated code than that. To help eliminate this kind of repetition, Firefox has recently implemented a new selector, called the Grouping Selector, that acts to collect duplicated patterns. Here's the syntax:

```
:-moz-any(selectors) { property: value; }
```

The *selectors* value is a comma-separated list of simple selectors—that is, no combinators or pseudo-elements—that apply if any of the selection criteria are met. I can perhaps better explain this by showing how to apply it to the first example in this section:

```
:-moz-any(article ul,div ul,ol,menu) ul li { property: value; }
```

Here you can see I've used `:-moz-any()` to provide a group of alternatives; basically I'm saying "select any list item that is the child of an unordered list, which is the child of any one of these elements." I save only a few characters here but avoid a lot of repetition (repeating `ul li` at the end of each of the four selector rules).

The Grouping Selector is currently only a proposal and still has unresolved issues relating to specificity. As such, Grouping Selector hasn't been suggested for any of the CSS3 modules yet, but you should be able to experiment with it in Firefox 4.

Constants and Variables

In Chapter 10 I introduced `currentColor`, which has the distinction of being the first variable in CSS. Just in case you've no programming background, *variables* and *constants* are strings of characters that represent data and whose value can be changed (a *variable*) or is fixed (a *constant*). By way of illustration, consider this fairly typical set of declarations:

```
E {  
  background-color: #F00;  
  border: 1px solid #F00;  
  color: #F00;  
}
```

This same color value could be used in many stylesheets, but if you ever needed to change it, you would have to update every instance of it—at the very least, a find-and-replace task. But the advantage of variables and constants is that you can define a character string to represent that value—let’s say you use the string `mainColor` and assign it the value of `#F00`—and use that string instead:

```
E {
  background-color: mainColor;
  border: 1px solid mainColor;
  color: mainColor;
}
```

Now if you want to change the color, you just update the value of `mainColor`, and the new color is applied automatically to all of the elements you set it on. That’s the theory.

The notion of defining a syntax for variables or constants in CSS has been floated many times, but until recently had yet to be taken up by the CSS Working Group. Although some firm opposition seems to be coming from within the CSSWG, two competing proposals have been put forth. The first was created by Daniel Glazman, co-chairman of the CSS Working Group, and Dave Hyatt, a leading light at WebKit, so this proposal has a pretty good pedigree. Their concept for CSS Variables (<http://www.disruptive-innovations.com/zoo/cssvariables/>) is notionally similar to the example I used in the introduction to this section and is persuasively simple. The first step is to use the new `@variables` at-rule to define string and value pairs:

```
@variables {
  exampleColor: #F00;
  exampleLength: 10em;
}
```

These strings can then be used as values in the `var()` notation, applying the value to relevant properties:

```
E {
  background-color: var(exampleColor);
  width: var(exampleLength);
}
```

The second proposal, CSS Constants (<http://fantasai.inkedblade.net/style/specs/constants/>), was created by another CSS Working Group member and is more flexible, allowing for three different types of constants: *values*, *style-sets*, and *selectors*. The first of those, *values*, works in a similar way to the CSS Variables proposal I just described, setting values in the `@define` at-rule:

```
@define values {
  exampleColor: #F00;
  exampleLength: 10em;
}
```

In this proposal, constants are then called using the backtick character (`):

```
E {  
  background-color: `exampleColor`;  
  width: `exampleLength`;  
}
```

Where CSS Constants go further than CSS Variables, however, is that you can use them to define whole sets of styles, as in the next code example:

```
@define style-sets {  
  exampleSet {  
    background-color: #F00;  
    width: 10em;  
  }  
}
```

You then apply this whole style-set to a property by simply using the constant string, again with the backtick:

```
E { `exampleSet` }
```

This proposal also allows you to assign groups of selectors to a constant:

```
@define selectors {  
  exampleSelector : ul ul li;  
}
```

Once again, you refer to the constant (with a backtick) in the selector chain:

```
`exampleSelector` E { <property>: <value>; }
```

As well as the differing syntaxes, the two proposals are distinguished from one another in terms of their scope: The values in CSS Variables are automatically inherited by other stylesheets, which are called using the `@import` rule, whereas those in CSS Constants would not be inherited by default, although you would have the option to allow it if required.

Both proposals have their attractions—the simplicity of Variables, the flexibility of Constants—and in an ideal world, I'd like to see a mixture of the two. I don't think selector constants are truly necessary, but values and style-sets are both great ideas. Here's hoping that the CSS Working Group comes to a decision about this.

WebKit CSS Extensions

As this book was going to press, the WebKit team announced that it is going to implement many new experimental features in its browser over the coming months. These are based on some of the extra features provided by programmer-created CSS extensions that run on the server, such as SASS (<http://www.sass-lang.com/>).

These features have not (at the time of writing) been proposed to the W3C—indeed, WebKit has not produced any written specifications yet—and are, therefore, some way from being implemented. In addition, these new features are not backward-compatible, so you may have to wait many years before they gain enough traction to be used on a day-to-day basis.

CSS Variables

I discussed two existing CSS Variables proposals in the previous section, but the WebKit proposal is yet another alternative. This proposal is similar to the first proposal I showed. First, you define the variables and values with the `@var` rule:

```
@var $exampleColor #F00;
```

You then put this value in your declarations by using the defined name with a string character (\$) before it:

```
E { background-color: $exampleColor; }
```

In this example element, *E* is given a background-color value of #F00.

Extending Variables Using Mixins

The syntax just introduced is useful for creating simple variables, but the proposal goes further with the introduction of *mixins*, which are blocks of reusable code. To use these, you first declare your blocks of properties with the `@mixin` rule:

```
@mixin exampleBlock {  
  background-color: #F00;  
  font-size: 150%;  
}
```

Then you insert the blocks into your other rules by using the `@mix` directive with the defined mixin name:

```
E {  
  color: #000;  
  @mix exampleBlock;  
}
```

Even better, you can pass parameters into mixins so they act like JavaScript functions, meaning you could extend your mixin like this:

```
@mixin exampleBlock($exampleColor #F00) {
  background-color: $exampleColor;
  font-size: 150%;
}
```

The defined name of the mixin now has a variable name in parentheses after it (`$exampleColor`) with a default value following it (`#F00`). The `background-color` property has this variable as a value. If no other parameter is provided, this property uses the default value. A parameter can be provided, however, when the `@mix` directive is used:

```
E {
  color: #000;
  @mixin exampleBlock(#00F);
}
```

Here, the mixin is called and a value of `#00F` passed to it as a parameter, replacing the default provided in the original `@mixin`. Mixins make CSS into a much more programmatic language, sacrificing some of its simplicity for extensibility.

CSS Modules

As programmers know, once you begin to introduce variable names you begin to run into problems of scope—that is, how variables defined in a function can affect other functions if given the same name. CSS variables and mixins all have global scope: Once defined anywhere, they apply everywhere. Not a problem if you're building a site yourself, but when many different developers are put into the mix, the chance of a variable name being defined twice and causing conflicts increases exponentially.

CSS Modules have been proposed to avoid this. You define a module with the `@module` rule and declare variables and mixins within it:

```
@module exampleModule {
  @var $exampleColor #F00;
  @mixin exampleBlock {
    background-color: #F00;
    font-size: 150%;
  }
}
```

Now to use a variable or mixin from this module, you either prefix it with the module name:

```
E {
  border-bottom-color: $exampleModule.exampleColor;
  @mix exampleModule.exampleBlock;
}
```

or use the @use directive:

```
E {
  @use exampleModule;
  border-bottom-color: $exampleColor;
  @mix exampleBlock;
}
```

This way, if you have two different variables called *\$exampleColor* in two different modules, only the one in the *exampleModule* module will be used, avoiding scope conflicts.

Nested Rules

The final new planned feature is the ability to nest rules. This feature is aimed at avoiding repetition in your code and is performed by using the ampersand character (&) before a nested selector:

```
article header h1 {
  font-size: 200%;
  & a {
    color: #F00;
    & :hover {
      color: #000;
    }
  }
}
```

At present, you write the same rules like this:

```
article header h1 { font-size: 200%; }
article header h1 a { color: #F00; }
article header h1 a:hover { color: #000; }
```

In just this simple code, you have a lot of repetition; imagine how much more repetition occurs on a large website with multiple stylesheets. Nested rules help to remove the repetition and make styles easier to maintain.

Haptic Feedback

As we move into the era of portable computing, the power of touch is becoming more important. Some touchscreen devices now provide *haptic feedback*—that is, they use force and vibration to emulate the sensation of touching physical objects. A new idea from Nokia, CSS Haptics (<http://www.starlight-webkit.org/CSS/css3-haptics.html>) proposes enabling haptic technology through CSS.

This proposal involves two new properties: `haptic-tap-type` and `haptic-tap-strength`. The first sets the feel of an element being tapped, and the second sets the strength of the feedback. For example, you might have something like this:

```
E {  
  haptic-tap-type: latched-button-down;  
  haptic-tap-strength: strong;  
}
```

This code provides the feel of a button that remains depressed after you push it with strong feedback. You could also provide the same values with the `haptic-tap` shorthand:

```
E { haptic-tap: latched-button-down strong; }
```

This proposal is brand new, so no decision has been made as to its future. However, haptic feedback is an example of what you might see as we move into a bold new age of personal computing using CSS.

Summary

Everything I've covered in this chapter faces an uncertain future—the implementation of CSS depends not on any kind of mandate from the W3C but from the willingness of browser makers to adopt it. Modules or individual properties remain unimplemented for many reasons, not all of them obvious: business decisions, resources, or just plain politics could all stand in the way of any of these new features becoming a standard.

But CSS is clearly moving beyond its humble beginnings as a way to provide simple decoration to text documents and toward a future where it becomes almost a language in itself, capable of adapting to the many devices that you will use to access the Web in the future.

I haven't covered everything in this book—I could never hope to—but I think I've covered enough to at least make you curious to find out what the next stages in the evolution of CSS will be. I urge you to stay connected to the conversation that surrounds styling elements for the Web, to download preview releases of browsers and create your own experiments, and to let the browser makers and W3C know the results of your experiments. CSS3 has (mostly) been molded around the desires of web developers, and your opinion and feedback is vital.

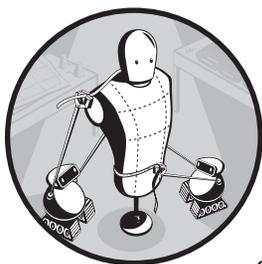
What changes are made to CSS3 as the specification reaches maturity will be interesting to see—and whether a future revision of this book varies wildly from the one you’ve just read. But if the fluid and uncertain nature of the final CSS3 spec doesn’t leave you excited about the possibilities of its future, consider this: Work has already begun on the first module of CSS4.

Future CSS: Browser Support

	WebKit	Firefox	Opera	IE
calc()	No	No (expected in Firefox 4; with prefix)	No	No (expected in IE9)
max()	No	No	No	No
min()	No	No	No	No
cycle()	No	No (expected in “a future version” of Firefox)	No	No
Grid Positioning	No	No	No	No
image()	No	No	No	No
Image Slices	No	No	No	No
Image Sprites	No	No	No	No
Grouping Selector	No	No (expected in Firefox 4; with prefix)	No	No
Constants and Variables	No	No	No	No
WebKit CSS extensions	No	No	No	No
Haptic Feedback	No	No	No	No

A

CSS3 SUPPORT IN CURRENT MAJOR BROWSERS



This appendix collects the browser support tables that are shown at the end of each chapter, providing an at-a-glance summary of the implementation of the CSS3 properties and rules featured in this book.

This listing is somewhat complicated by two factors: The first is that, as I've mentioned repeatedly, CSS3 is still in a state of flux, and some properties are still very subject to change (*box-shadow* was dropped and reinstated in the time I spent writing this book, for example); the second is that new versions of browsers are released regularly and consistently, and each release sees a host of new implementations.

In the following tables, I indicate implementation status in the four major browser types: WebKit, Firefox, Opera, and Internet Explorer. The most up-to-date versions of those browsers I had available to me when I began writing this book were:

WebKit Safari 4.04 and Chrome 6.0

Firefox Firefox 3.6 and Beta releases of Firefox 4.0

Opera Opera 10.5

Internet Explorer Internet Explorer 8

During the period I was writing this book, Safari 5, Opera 11, and Chrome 10.0 were officially released, and Betas of Internet Explorer 9 (IE9) were also made available. Where possible, I updated the support tables with any new or changed properties that were implemented in those browsers, but I can't be certain that they are 100 percent accurate—especially where IE9 is concerned.

I plan to keep updated versions of these tables on the website that accompanies this book (<http://www.thebookofcss3.com/>), so check there regularly, particularly if you want to find out what's in store from Microsoft—IE9 is looking like an impressive new browser that could catapult certain aspects of CSS3 into the mainstream.

Media Queries (Chapter 2)

	WebKit	Firefox	Opera	IE
Media Queries	Yes	Yes	Yes	No (expected in IE9)

Selectors (Chapter 3)

	WebKit	Firefox	Opera	IE
Attribute Selectors	Yes	Yes	Yes	Yes
General Sibling Combinator	Yes	Yes	Yes	Yes

Pseudo-classes and Pseudo-elements (Chapter 4)

	WebKit	Firefox	Opera	IE
Structural Pseudo-classes	Yes	Yes	Yes	No (expected in IE9)
:target	Yes	Yes	Yes	No (expected in IE9)
:empty	Yes	Yes	Yes	No (expected in IE9)
:root	Yes	Yes	Yes	No (expected in IE9)
:not	Yes	Yes	Yes	No (expected in IE9)
Pseudo-elements (new syntax)	Yes	Yes	Yes	No (expected in IE9)
UI element states	Yes	Yes	Yes	No (expected in IE9)
:selection	Yes	Yes	Yes	No (expected in IE9)

Web Fonts (Chapter 5)

	WebKit	Firefox	Opera	IE
@font-face	Yes	Yes	Yes	Yes (.eot only; other formats expected in IE9)
font-size-adjust	No	Yes	No	No
font-stretch	No	No	No	No

Text Effects and Typographic Styles (Chapter 6)

	WebKit	Firefox	Opera	IE
text-shadow	Yes	Yes	Yes	No
text-outline	No	No	No	No
text-stroke	Yes	No	No	No
text-align (new values)	Yes	Yes	No	No
text-align-last	No	No	No	Yes
word-wrap	Yes	Yes	Yes	Yes
text-wrap	No	No	No	No
text-rendering	No	Yes	No	No

Multiple Columns (Chapter 7)

	WebKit	Firefox	Opera	IE
column-count	Yes (with prefix)	Yes (with prefix)	No	No
column-width	Yes (with prefix)	Yes (with prefix)	No	No
column-gap	Yes (with prefix)	Yes (with prefix)	No	No
column-rule	Yes (with prefix)	Yes (with prefix)	No	No
columns	Yes (with prefix)	No	No	No
column-span	No	No	No	No
break-*	No	No	No	No

Background Images and Other Decorative Properties (Chapter 8)

	WebKit	Firefox	Opera	IE
Multiple Background Images	Yes	Yes	Yes	No (expected in IE9)
background-size	Yes (with prefix in Safari 4 and earlier)	Yes (with prefix; expected in Firefox 4 without prefix)	Yes	No (expected in IE9)
background-clip	Yes (with prefix in Safari 4 and earlier)	Yes (with prefix; expected in Firefox 4 without prefix)	Yes	No (expected in IE9)
background-origin	Yes (with prefix in Safari 4 and earlier)	Yes (with prefix; expected in Firefox 4 without prefix)	Yes	No (expected in IE9)
background-repeat (new values)	No	No	Yes	No (expected in IE9)
image-rect	No	Yes (with prefix)	No	No
mask-*	Yes (with prefix)	No	No	No

Border and Box Effects (Chapter 9)

	WebKit	Firefox	Opera	IE
border-radius	Yes (with prefix in Safari 4 and earlier)	Yes (with prefix; incorrect syntax on sub-properties; expected in Firefox 4 without prefix)	Yes	No (expected in IE9)
border-image	Yes (with prefix)	Yes (with prefix)	Yes	No
Multiple border-color Values	No	Yes	No	No
box-shadow	Yes (with prefix)	Yes (with prefix)	Yes	No
box-decoration-break	No	No	No	No

Color and Opacity (Chapter 10)

	WebKit	Firefox	Opera	IE
opacity	Yes	Yes	Yes	No (expected in IE9)
RGBA Values	Yes	Yes	Yes	No (expected in IE9)
HSL Values	Yes	Yes	Yes	No (expected in IE9)
HSLA Values	Yes	Yes	Yes	No (expected in IE9)
currentColor Value	Yes	Yes	Yes	No (expected in IE9)
appearance	Yes	Yes	No	No

Gradients (Chapter 11)

	WebKit	Firefox	Opera	IE
Gradients	Yes (with prefix; incorrect syntax; correct syntax in future versions)	Yes (with prefix)	No	No
Repeating Gradients	No	Yes (with prefix)	No	No

2D Transformations (Chapter 12)

	WebKit	Firefox	Opera	IE
2D Transformations	Yes (with prefix)	Yes (with prefix)	Yes (with prefix)	No (expected in IE9 with prefix)
box-reflect	Yes (with prefix)	No	No	No

Transitions and Animations (Chapter 13)

	WebKit	Firefox	Opera	IE
Transitions	Yes (with prefix)	No (expected in Firefox 4 with prefix)	Yes (with prefix)	No
Animations	Yes (with prefix)	No	No	No

3D Transformations (Chapter 14)

	WebKit	Firefox	Opera	IE
3D Transformations	Yes (with prefix)	No	No	No

Flexible Box Layout (Chapter 15)

	WebKit	Firefox	Opera	IE
Flexible Box Layout	Yes (with prefix)	Yes (with prefix)	No	No (possibly in IE9 with prefix)

Template Layout (Chapter 16)

	WebKit	Firefox	Opera	IE
Template Layout	No	No	No	No

The Future of CSS (Chapter 17)

	WebKit	Firefox	Opera	IE
calc()	No	No (expected in Firefox 4 with prefix)	No	No (expected in IE9)
max()	No	No	No	No
min()	No	No	No	No
cycle()	No	No (expected in "a future version" of Firefox)	No	No
Grid Positioning	No	No	No	No
image()	No	No	No	No
Image Slices	No	No	No	No
Image Sprites	No	No	No	No
Grouping Selector	No	No (expected in Firefox 4 with prefix)	No	No
Constants and Variables	No	No	No	No
WebKit CSS extensions	No	No	No	No
Haptic Feedback	No	No	No	No

B

ONLINE RESOURCES



In this appendix, I list some useful resources for finding out more about CSS3 and some tools that help you build websites using the new properties and features to their full potential. A million and one sites offer demonstrations and tricks; throw a stone at Google and you'll hit 500 pages of them. Although some are useful, many are notably less so, so I've tried to steer clear of "30 Awesome Things You Can Do with CSS3!" blog posts and offer, instead, resources that I think are of more practical value.

This list is very much a work in progress, and I aim to keep an updated list on the website that accompanies the book, <http://www.thebookofcss3.com/>. If you know of any resources that you think I should add, get in touch through the website and let me know.

CSS Modules

The W3C's own pages on CSS have a list showing the current status of all of the modules, which should be your first port of call if you want to know what's on the horizon. Each of the modules is shown with its current and

upcoming status—for example, the Backgrounds and Borders module currently has Candidate Recommendation (CR) status and should soon become a Proposed Recommendation (PR)—and all are grouped by priority, giving some indication of how likely they are to be accepted. (For example, the Multi-column Layout Module you learned about in Chapter 7 is High Priority, so you can expect to see it fast-tracked through the process.)

Below the summary list, each module’s journey through the recommendation process is shown, so you can see its history. Each version is archived and listed so you can, if you so wish, take a look at which properties survived and which got culled:

<http://www.w3.org/Style/CSS/current-work>

Browsers

Each of the four main browser engines has a dedicated documentation and resource website, which made writing this book a lot easier than it would have been otherwise. Although I still had to test all of my examples and demonstrations in each different browser, knowing which properties should be supported, which require a prefix, and so on, was extremely useful in planning those examples in the first place.

WebKit

Although the WebKit project itself doesn’t have too much in the way of documentation that’s useful for your purposes (as is, unfortunately, all too common with many open source projects), Apple has a wealth of it at their Safari Developer site:

<http://developer.apple.com/safari/>

The most relevant section is their CSS reference, which lists all of the supported properties, including those with a `-webkit-` prefix:

<http://developer.apple.com/safari/library/documentation/AppleApplications/Reference/SafariCSSRef/>

Firefox

Firefox bucks the open source documentation trend I just mentioned with their fantastic CSS reference on their excellent Mozilla Developer Network. This site explains all of the properties in detail, with examples and illustrations, and I referred to it on many, many occasions while writing this book:

https://developer.mozilla.org/en/CSS_Reference/

They also list all of the `-moz-` prefixed properties on a single page:

https://developer.mozilla.org/en/CSS_Reference/Mozilla_Extensions/

Opera

The layout engine used in Opera is called Presto, and the latest version is 2.7, which powers Opera 11. Opera's documentation page helpfully lists all of the CSS supported in Presto, with exhaustively detailed tables of the exact implementation levels:

<http://www.opera.com/docs/specs/presto27/>

Internet Explorer

The Microsoft Developer Network has a page called CSS Compatibility and Internet Explorer, which lists all of the CSS features implemented in many versions of their browser:

[http://msdn.microsoft.com/en-us/library/cc351024\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/cc351024(VS.85).aspx)

If IE9 has not yet been released as you read this, the IE9 Guide for Developers contains a wealth of information about forthcoming CSS3 support:

<http://msdn.microsoft.com/en-gb/ie/ff468705.aspx>

Browser Support

In addition to the tables I've provided in Appendix A, a number of resources are available that show the level of support for CSS3 in all browsers.

When Can I Use . . .

When Can I Use covers all emerging web technologies (not just CSS3) and shows old, current, and future versions of browsers, listing the availability of features in each. If a feature is available in a particular browser, this site shows which version was the first to offer that feature. And if the feature is not available, the site shows when it may be implemented in the future. When Can I Use is very complete, and you can find it at:

<http://www.caniuse.com/>

Quirks Mode

Quirks Mode is a venerable website, which for many years has been documenting and comparing standards support, bugs, and—as the title suggests—quirks in browsers. They have tables detailing CSS3 implementation that, although not extensive, are nonetheless quite complete:

<http://www.quirksmode.org/css/contents.html>

Find Me By IP

Find Me By IP has a suite of tools to show you which CSS3 (and other technologies') features are supported by the browser you're using. The results are binary: Either your browser supports a feature or it doesn't—no concession is made as to whether the implementation has any quirks or peculiarities. Here's the URL:

<http://www.fmbip.com/#css3>

Feature Detection and Simulation

If you're building a site using CSS3, one of your major considerations must be what happens in browsers that don't have support for it (**cough*IE*cough**). Most of the time you can work around the problem by simply paying attention to the fallback values, but on occasion you will need some extra help.

Perfection Kills

JavaScript is ideal for this purpose; you can use the `Style` object to test if a property is supported and perform actions based on the result of that test. If you want to test on a property-by-property basis, the Perfection Kills blog has a good overview of how to do so:

<http://www.perfectionkills.com/feature-testing-css-properties/>

Modernizr

A better alternative, however, might be to use Modernizr. This lightweight JavaScript library tests the visitor's browser for CSS3 support and adds a class to the `html` element for each property supported. Here's an example:

```
<html class="boxshadow csstransitions opacity">
```

Then you can use targeted CSS rules, for instance:

```
div { border: 2px solid black; }
.boxshadow div {
  border: none;
  box-shadow: 2px 2px black;
}
```

In this example, the `div` element has a 2px black border in browsers that don't support `box-shadow` and a black shadow but no border in browsers that do. If this is appealing, you can download Modernizr at:

<http://www.modernizr.com/>

CSS3 Pie

Taking the opposite approach is CSS3 Pie, which uses proprietary Microsoft features to simulate some CSS3 properties in IE8 and below. Adding an HTML component to your style with the `behavior` property leverages Vector Markup Language (VML), a Microsoft-only graphics feature similar to SVG, which allows you to use `border-image`, `border-radius`, `box-shadow`, and more. Whether you should or not is a different matter; I say that designs should be allowed to degrade gracefully in older browsers, but your mileage may vary. Have a look and make up your own mind:

<http://www.css3pie.com/>

Code-Generation Tools

Remembering all of the small implementation differences in prefixed properties can be quite difficult if you don't have this book on hand, and typing out all of those properties manually can be an onerous task. A few websites have realized this and created tools that take the donkey work out of writing cross-browser code.

CSS3, Please!

CSS3, Please! not only helps you get around the repetition of typing all of the different prefixed properties required for many features but also provides a live preview of the features in question. Updating a value in any property automatically updates it in all of its sibling properties, so you can copy and paste the code when you're satisfied with its output. CSS3, Please! also goes a step further and replicates some CSS3 features with proprietary Microsoft filters, but if I were you, I'd ignore those as they're nonstandard. Here's the URL:

<http://www.css3please.com>

CSS3 Generator

CSS3 Generator performs a similar task but with a few more properties and a quite different interface. It, too, provides Microsoft-only filters and allows only pixel-length values, but this tool is a useful little helper:

<http://www.css3generator.com/>

CSS3 Gradient Generator

Back in Chapter 11, I discussed CSS gradients and spent a long time explaining the difference between the Firefox and WebKit syntaxes. The CSS3 Gradient Generator takes all of the hard work and calculation out of creating gradients

and generates cross-browser code to your specifications. Although this tool doesn't allow for all of the possible options, it should help you out in most cases. Check it out:

<http://gradients.glrzad.com/>

Type Folly

Type Folly is a tool built solely for experimenting with typographic features. Photoshop users should be instantly familiar with its interface, which allows you to create and edit text layers, adding font, text, and 2D transformation properties to each. The tool seems a bit fiddly for my liking, but I'm sure that with practice, you might find it useful for typographical experimentation:

<http://www.typefolly.com/>

Web Fonts

Web fonts are the hot new thing in web design, and a new web font service seems to get launched every week. Some are free; some you have to pay for; some allow you to host the fonts yourself and some are cloud-based; some require JavaScript; and others can be called directly in the font stack. Rather than provide an exhaustive list, I'll cover a few of the more interesting ones.

Typekit

First to market (I think) was Typekit, which is a subscription service. For a yearly fee, you get access to a wide range of fonts, which you organize into kits. Each kit has a unique ID that you refer to with a script element in the head of your page, and you apply the web fonts to your elements with a series of unique class names. Typekit is a pretty elegant solution, although the annual subscription makes it perhaps more suitable to those who run a range of websites:

<http://www.typekit.com/>

Fontdeck

Fontdeck is also subscription-based, but it's a per-font rather than blanket subscription. Their solution is somewhat different than Typekit's: First, you call a unique stylesheet using the link element, and then you name each of your chosen fonts in the font stack as you usually would. This solution has the advantage of not requiring JavaScript. The caveat of this service, however, is that each font is licensed both per-year and per-site, so the price could very quickly start stacking up:

<http://www.fontdeck.com/>

Fonts.com Web Fonts

Another similar service is Fonts.com Web Fonts. This service offers a range of some 9,000 fonts from some of the most famous type foundries, such as Linotype and Monotype, and includes fonts like Helvetica and Frutiger, which are unavailable anywhere else. It uses a (somewhat clumsy) pure JavaScript solution that sits between the implementations of Typekit and Fontdeck in the way it works. They have a range of pricing options, starting from free:

<http://webfonts.fonts.com/>

Google Font API

Recently Google entered the game with the Google Font API, which provides an easy way to embed selected free and open source fonts in your pages. Google uses a very similar method to that of Fontdeck (but without requiring any account or payment): You call a stylesheet containing the @font-face rule—either with `link` or `@import`—and then place the font in your stack:

```
<link href="http://fonts.googleapis.com/css?family=Cardo" rel="stylesheet">
<style>
h1 { font-family: Cardo, serif; }
</style>
```

Only a limited set of fonts are available currently, but the selection is sure to grow over time:

<http://code.google.com/webfonts/>

Web FontFonts

If renting fonts is not your idea of good value, you'll probably want to buy them. I mentioned the new WOFF format in Chapter 5, and one of the first sites to sell the format is FontShop's Web FontFonts. They sell a range of fonts (in both WOFF and EOT formats) that you can host yourself; interestingly, they also have a deal with Typekit that allows you to buy the fonts from FontShop and then host them with Typekit, which means you can embed them in browsers that don't support either of the sold formats. Clever. See it for yourself:

http://www.fontshop.com/fontlist/n/web_fontfonts/

Font Squirrel

I also mentioned Font Squirrel in Chapter 5, but they deserve another mention here because they provide a great range of free fonts that come @font-face ready:

<http://www.fontsquirrel.com/>

Fontspring

Font Squirrel also has a commercial partner, Fontspring, that provides the same service but with paid-for fonts. You can buy single-site or unlimited licenses, and the fonts come from a range of independent foundries:

<http://www.fontspring.com/>

Other Resources

The following sites are more general resources; although not as immediately useful as the tools listed previously, they're definitely ones you'll want to refer to in order to immerse yourself in the rich possibilities of CSS3.

CSS3.info

One of the first blogs to discuss and show the possibilities of CSS3 was CSS3.info, for which I wrote many articles between 2006 and 2009. This blog features the Automated Selectors Test, which was used as a benchmark by browser makers, and the articles and subsequent discussions helped to shape the syntax of some properties (notably, `border-radius`). With increased competition nowadays, CSS3.info is no longer as relevant as it used to be (and I'm not just saying that because I no longer write for them), but it still has a rich archive of tutorials and demonstrations:

<http://www.css3.info/>

CSS3 Watch

CSS3 Watch collects, in their words, “examples of creative, innovative, and unexpected use of CSS3.” I would add to that list “nonsemantic” and “purely decorative,” as many of the creations they demonstrate would probably best be done with other technologies. But as a showcase of CSS3's possibilities, and perhaps as a source of inspiration, I'd say it's quite impressive. Check it out:

<http://www.css3watch.com/>

CSS3 Cheat Sheet

And finally, if you think this is all too much to remember, *Smashing Magazine* has provided a five-page PDF CSS3 Cheat Sheet that lists every single value for every single property in every single module—quite an achievement and incredibly useful, although be aware that some properties have changed since this document was created:

<http://www.smashingmagazine.com/2009/07/13/css-3-cheat-sheet-pdf/>

INDEX

Symbols and Numbers

- & (ampersand character), for nested selector, 247
- * (asterisk), for Arbitrary Substring Attribute Value selector, 28
- @ (at sign), for element default content, 226–228
- ` (backtick character), for constants, 244
- ^ (caret), for Beginning Substring Attribute Value selector, 25
- \$ (dollar sign), for Ending Substring Attribute value selector, 27
- :: (double colon), for CSS3 pseudo-elements, 47
- . (period), as spacer for empty slot, 223
- 2D Transformations Module, 147
 - browser support, 161
- 3D elements, 180–181
- 3D Transforms Module, 179, 180
 - browser support, 194
 - perspective-origin property, 190–191
 - perspective property, 190
 - showing or hiding backface, 193
 - transformation matrix, 187–188
 - transform-origin property, 191–192
 - transform property, 182–190
 - perspective function, 188–190
 - rotate function, 183–185
 - scale function, 186–187
 - translate along axis, 185–186

A

- accessibility, testing colors for, 71
- active animation, setting, 176
- :active pseudo-class, as trigger, 171
- adjacent keyword, for punctuation-trim property, 79
- Adjacent Sibling Combinator, 30
- :after pseudo-element, 47

- alignment
 - of box elements, 208–209
 - of text, 75–76
- all keyword, for transition-property property, 165
- Alpha color channel, 119, 122–129
 - HSL with, 127
- alternate keyword, for animation-direction property, 175
- ampersand character (&), for nested selector, 247
- and operator
 - for chaining queries, 20
 - for Media Query, 12
- Android devices, device-width for, 16
- angle argument, for linear gradient, 133
- Animations Module, 163, 172
 - browser support, 178
 - complete example, 177
 - key frames for, 172–173
 - multiple, 177–178
 - properties, 173–176
 - animation-delay property, 174–175
 - animation-direction property, 175
 - animation-duration property, 174
 - animation-iteration-count property, 175
 - animation-name property, 173–174
 - animation-play-state property, 176
 - animation-timing-function property, 174
 - shorthand, 176
- antialiased keyword, for `-webkit-font-smoothing` property, 78
- anti-aliasing, 78
- appearance property, 129, 230
 - browser support, 130
- Arbitrary Substring Attribute Value selector, 28–29, 83
- artificial font faces, vs. `true`, 53
- aspect ratio, testing browser's, 18–19
- asterisk (*), for Arbitrary Substring Attribute Value selector, 28

- at sign (@), for element default content, 226–228
- attributes, details based on devices, 10
- attribute selectors, 23, 24–29
 - Arbitrary Substring, 28–29
 - Beginning Substring, 25–27
 - browser support, 31
 - chaining, 29
 - Ending Substring, 27–28
- auto keyword
 - for column-fill property, 87
 - for text-rendering property, 77
- axes, 66–67, 185

B

- backface of elements, showing or hiding, 193
- background-* properties
 - backface-visibility property, 193
 - background-clip property, 98–100
 - background-color property, 95
 - ::selection pseudo-element for applying, 47
 - background-image property, 94–95
 - for applying gradients, 141–142
 - background-origin property, 100–101
 - background-position property, 94–95
 - background-repeat property, 94–95, 102–103
 - background-size property, 20, 96–98
 - for ::slot() pseudo-element, 223
- background color of link, changing with target selector, 43
- background images, 94–104
 - avoiding download, 16–17
 - basing on browser window size, 14
 - as borders, 111–114
 - browser support, 106
 - clipping, 103–104
 - multiple, 94–96
 - position of, 98–101
 - scaling, 96–98
- Backgrounds and Borders Module, 94, 108
 - browser support, 117
- backtick character (`), for constants, 244
- balance keyword, for column-fill property, 87
- bandwidth allowances, background image and, 17
- baseline keyword, for box-align property, 209
- Basic User Interface Module, 129
- :before pseudo-element, 47
- Beginning Substring Attribute Value selector, 25–27
- behavior, vs. presentation, 163
- behavior property, 261
- block-axis keyword, for box-orient property, 204
- blur-radius option
 - for box-shadow property, 115
 - for text-shadow property, 69–70, 71
- border-box keyword
 - for background-clip property, 98–99
 - for background-origin property, 100
 - for border-box property, 6
- border-color property, RGBA implementation, 125
- border-image-outset property, 114
- border-image property, 111–114
 - browser support, 117
- border-image-repeat property, 114
- border-image-slice property, 114
- border-image-source property, 114
- border-image-width property, 114
- border-radius property, 108
 - browser support, 111, 117
 - shorthand, 109–111
- borders
 - browser support, 117
 - effects, 107
 - and element width, 6
 - images for, 111–114
 - mask as, 106
 - multicolored, 114–115
 - rounded corners, 108–111
- Bos, Bert, 215
- both value, for resize property, 75
- bottom keyword
 - for border-radius property, 108
 - for perspective-origin property, 190
- box-* properties, 212
- box-align property, 208–209, 211
- box-direction property, 205–206, 212
- box effects, browser support, 117
- box-flex-group property, 203
- box-flex property, 199, 202, 212
- box keyword, for display property, 196
- box-lines property, 211
- box-ordinal-group property, 206–207, 212
- box-orient property, 204, 208, 212
- box-pack property, 209, 211
- box-shadow property, 115–117, 124
 - browser support, 117
- box-sizing property, 4–5, 6, 7
- break-after property, 91
- break-before property, 91
- break-inside property, 91
- break-word value, for word-wrap property, 76

- browsers
 - implementation of CSS2, 2
 - and prefixes, 7–8
 - web resources, 258–259
 - width, 13
- browser support, 251–255
 - for 2D Transformations Module, 161
 - for 3D Transforms Module, 194
 - for Animations Module, 178
 - for Backgrounds and Borders Module, 106, 117
 - for Color Module, 130
 - CSS future possibilities, 249
 - for DOM and attribute selectors, 48
 - for Flexible Box Layout Module, 213
 - for Fonts Module Level 3, 64
 - for Image Values and Replaced Content Module, 146
 - for Media Queries Module, 21
 - for Multi-column Layout Module, 92
 - for Selectors Level 3 Module, 31
 - for Template Layout Module, 228
 - for Text Level 3 Module, 80
 - for Transitions Module, 178
 - web resources, 259–260
- buttons, in WebKit on Ubuntu, 129–130

C

- calc() function, 230–232
- Candidate Recommendation status, 3
- canvas element, 131
- caret (^), for Beginning Substring Attribute Value selector, 25
- Cartesian coordinate system, 66, 180
- case sensitivity, attribute selector and, 26
- centering elements inside parent, 210–211
- center keyword
 - for box-align property, 208
 - for box-pack property, 208
 - for perspective-origin property, 190
- chaining attribute selectors, 29
- checkbox input element, 46
- checked pseudo-class selector, 45–46
- child boxes
 - dynamic resizing to fit parent, 199–201
 - unequal widths, 201
- child elements, opacity property inheritance, 121
- Chrome. *See* WebKit
- Chunk font, 50
- circle keyword
 - for -moz-radial-gradient property, 138
 - for radial gradient, 137
- circular radial gradient, 138

- classes. *See* pseudo-classes
- class names, 34
- class selectors, 23
- clip keyword, of text-overflow property, 74
- clipping background images, 103–104
- closest-side keyword, for -moz-radial-gradient property, 138
- code-generation tools, 261–262
- color
 - Alpha color channel, 119, 122–129
 - background, of link, 43
 - in CSS2, 119
 - currentColor variable, 127–129, 130, 242
 - of drop shadow, 115
 - HSL (Hue, Saturation, Lightness), 125–127
 - matching from operating system, 129–130
 - multiple for borders, 114–115
 - new and extended values, 122
 - RGB (Red, Green, Blue) model, 119
 - RGBA (Red, Green, Blue, Alpha) model, 122
 - and graceful degradation, 124–125
 - of text shadow, 68
 - of text stroke, 72
- Color Module, 119
 - browser support, 130
- color property
 - alpha value, 124
 - ::selection pseudo-element for applying, 47
- color stop
 - adding to linear gradient, 135–136
 - in gradient, 132
- color-stop() function, 137
 - for radial gradients, 140–141
 - for WebKit, 135
- column-count property, 82–83
 - combining with column-width, 87–88
- column-fill property, 87
- column-gap property, 88–89
- column-rule-color property, 89
- column-rule property, 88–89
- column-rule-style property, 89
- column-rule-width property, 89
- columns, 81–92
 - browser support for, 92
 - containing elements within, 90–92
 - elements spanning multiple, 91
 - gaps and rules, 88–89
 - layout methods, 82–88
 - distribution differences in Firefox and WebKit, 86–87
 - dynamic columns, 83–84

- columns (*continued*)
 - lines between, 89
 - readability, 84–86
 - setting height and width, 223–226
- column-span element, 91
- column-width property, 83–84, 87–88
- combinator, 23
- common ligatures, 63
- constants, future possibilities, 242–244
- contain keyword
 - for background-size property, 97–98
 - for `-moz-radial-gradient` property, 138
- content-box keyword
 - for background-clip property, 98–99
 - for background-origin property, 100
 - for border-box property, 6
- coordinates, 66–67
- corners for borders, rounded, 108–111
- cos (cosine) function, 158
- cover keyword
 - for background-size property, 97–98
 - for radial gradient, 138
- CSS, future-proofing experimental, 8
- CSS2.1, 2, 3
- CSS3
 - future possibilities, 229–249
 - constants and variables, 242–244
 - cycling, 233
 - GCPM (Generated Content for Paged Media) module, 237
 - Grid Positioning Module, 233–237
 - Haptics, 248
 - images, 237–241
 - mathematical operations, 230–233
 - modules, 246–247
 - nested rules, 247
 - history, 2
 - modules, 2–3
- CSS3 Cheat Sheet, 264
- CSS3 Generator, 261
- CSS3 Gradient Generator, 261–262
- CSS3.info, 264
- CSS3 Pie, 261
- CSS3, Please!, 261
- CSS3 Watch, 264
- CSS Constants proposal, 243
- CSS Effects, 104
- CSS Pixel unit (px), 19
- CSS Variables, 243
- cubic-bezier function, for transition-timing-function property, 166–168
- currentColor variable, 127–129, 242
 - browser support, 130

- current state, selecting UI elements based on, 45–46
- cycling, 233

D

- datetime attribute, 27
- dConstruct conference, 10, 11
- declaration, value of, 5
- decorative elements, 93. *See also* background images
 - images as borders, 111–114
- default content for element, @ (at sign) for, 226–228
- default element, for 3D, 181
- default origin, of element, 67
- @define rule, 241
- DejaVu Serif font, 78
- Deveria, Alexis, 215, 216, 228
- device-aspect ratio, testing device's, 18–19
- device-height feature, 16
- device-pixel-ratio, 19
- device-width Media Feature, 15–16
- direction vector, calculating, 184
- disabled attribute, of textarea element, 45
- disabled pseudo-class selector, 45–46
- disabling kerning, 62
- discretionary ligatures, 63
- display property, 198, 212
 - for column and row size, 224
 - for creating rows, 217–218
 - row strings on, 219
- div element
 - splitting content into columns, 82
 - width calculation, 5
- document tree, selecting first element, 44
- dollar sign (\$), for Ending Substring Attribute value selector, 27
- DOM selectors, 23
- double colon (::), for CSS3 pseudo-elements, 47
- downloading
 - background image, avoiding, 16–17
 - fonts, forcing, 55
 - drop shadows, 67–71, 115–117
- duration of animation, 174
- dynamic columns, 83–84

E

- ease-in keyword
 - for animation-timing function property, 174
 - for transition-timing-function property, 166

- ease-in-out keyword
 - for animation-timing function property, 174
 - for transition-timing-function property, 166
- ease keyword
 - for animation-timing function property, 174
 - for transition-timing-function property, 166
- ease-out keyword
 - for animation-timing function property, 174
 - for transition-timing-function property, 166
- elements. *See also* pseudo-elements
 - @ (at sign) for default content, 226–228
 - centering inside parent, 210–211
 - containing within columns, 90–92
 - flipping, 155
 - origin, 67, 191
 - positioning with gr unit, 236
 - resizing, 74–75, 154–155
 - scaling, 157
 - spanning multiple columns, 91
 - total width calculation, 4–5
 - transformed, position in document flow, 149–150
- ellipse keyword, for radial gradient, 137
- ellipsis keyword, for text-overflow property, 74
- Embeddable Open Type (EOT) font format, 54
- empty pseudo-class, 44
 - browser support, 48
- enabled pseudo-class selector, 45–46
- end-edge keyword, for hanging-punctuation property, 79
- Ending Substring Attribute Value selector, 27–28
- end keyword
 - for box-align property, 208
 - for hanging-punctuation property, 79
 - for punctuation-trim property, 79
 - for text-align property, 75
- EOT (Embeddable Open Type) font format, 54, 55
- even keyword, 36
- Exact Attribute Value Selector, 24
- expanded keyword, for font-stretch property, 61
- experimental CSS, future-proofing, 8
- explicit grids, 234–236
- external links, adding icon to signify, 26
- external stylesheet, for Media Query, 11
- extra-condensed keyword, for font-stretch property, 61
- extra-expanded keyword, for font-stretch property, 61

F

- FaaS (Fonts as a Service), 57
- fallback for images, 238
- farthest-side keyword, for radial gradient, 138, 139
- file-type extensions, rules for, 27
- Find Me By IP, 260
- Firefox. *See also* -moz- prefix
 - 2D Transformations Module, 147
 - algorithms for column calculations, 86–87
 - and artificial font faces, 53
 - border-image property, 111
 - box-lines property, 211
 - calc() function, 232
 - and Flexible Box Layout Module, 196, 197–198, 203
 - Grouping Selector, 242
 - and Media Queries Module, 10
 - and multiple backgrounds, 96
 - support. *See* browser support and text shadows, 68
 - web resources, 258
 - zero values and layouts, 202
- Firefox Mobile browser, 20, 21
- :first-child pseudo-class, 35
- :first-letter pseudo-element, 47
- :first-line pseudo-element, 47
- first-of-type pseudo-class, 40–41
- fit-content keyword, for column widths, 225
- flat keyword, for transform-style property, 182
- flex-direction property, 212
- flex-grow property, 212
- Flexible Box Layout Module, 195
 - alignment, 208–209
 - box-ordinal-group property, 206–207
 - browser support, 213
 - changing order of boxes, 205–207
 - changing orientation, 204–205
 - cross-browser flex box with JavaScript, 211–212
 - dynamic resizing child boxes to fit parent, 199–201
 - grouping boxes, 203–204
 - initiating layout mode, 196–198
 - multiple rows or columns, 211
 - new syntax, 212
 - same-axis alignment, 209–211
 - unequal ratios, 201
 - zero values and layouts, 202
- flex-index property, 212
- flex-order property, 212

- flex-shrink property, 212
- flipping elements, 155
- float property, 237
- Fontdeck, 262
- @font-face Generator (Font Squirrel), 56
- @font-face rule, 49, 50–53
 - browser support, 64
 - bulletproof syntax, 54–56
 - and font management software, 55
 - for multiple, 51–52
 - src property, 55–56
- font faces
 - defining different, 51–52
 - true vs. artificial, 53
- font foundries, restrictions, 57
- fonts
 - forcing download, 55
 - formats, 55–56
 - licensing for web use, 57
 - name of, 51
 - OpenType fonts, 61–63
 - properties, 59–61
 - font-family, 50
 - font-size-adjust, 59–60, 64
 - font-stretch, 60–61, 64
 - font-style, 52
 - font-variant, 62
 - x-height ratio estimation, 60
- Fonts as a Service (FaaS), 57
- Fonts.com web fonts, 263
- FontShop, 263
- Fonts Module Level 3, 50
 - browser support, 64
- Fontspring, 264
- Font Squirrel, 56, 263
- form elements
 - cross-browser styling of, 46
 - states, 45–46
- fraction unit, 235
- from() function, 133, 137
- from-stop, in gradient, 132

G

- Gecko, 21
- General Sibling Combinator, 30–31
- Gentium Basic font, 53
- geometricPrecision keyword, for text-rendering property, 77
- Georgia font, 59–60
- Glazman, Daniel, 243
- Google Font API, 263
- gr (grid unit), 236

- gradients, 131–146
 - browser support, 146
 - in Firefox, repeating, 142–145
 - generator for, 261–262
 - linear, 132–136
 - adding color-stop values, 134–135
 - using, 134–135
 - in WebKit, 133–134
 - multiple, 141–142
 - radial, 136–141
 - in Firefox, 137
 - multiple color-stop values, 140–141
 - using, 138–139
 - in WebKit, 137
 - WebKit advantage, 141
 - with reflection, 160–161
- graphics. *See* images
- grid-columns property, 234
- Grid Positioning Module, 233–237
 - float property, 237
- grid-rows property, 234
- grid unit (gr), 236
- Grosskopf, Neal, 228
- grouping
 - flexible boxes, 203–204
 - selectors, 241–242

H

- h2 element
 - break-after for, 91–92
 - rotation, 149
- hanging-punctuation property, 79
- haptic feedback, 248
- haptic-tap-strength property, 248
- haptic-tap-type property, 248
- hardware acceleration for graphics, 179
- height Media Feature, 14
- height of columns and rows, setting, 223–226
- height property, of element, 7
- hidden keyword
 - for backface-visibility property, 193
 - and transform-style value, 182
- hiding, backface of elements, 193
- horizontal keyword
 - for aspect ratio, 19
 - for box-orient property, 204
 - for resize property, 75
- :hover pseudo-class
 - background color transition for, 164
 - transition for, 169–170
- HSL (Hue, Saturation, Lightness), 125–127
 - browser support, 130

HSV (Hue, Saturation, Value), 127
HTML5, 4
html element, 44
Hue, Saturation, Lightness (HSL), 125–127
 browser support, 130
Hue, Saturation, Value (HSV), 127
Hyatt, Dave, 243
hyperlinks
 adding icon to signify external, 26
 background color, target selector to
 change, 43
 internal, 42
 selector for visual clarity, 27
hyphenation, 85–86

I

icons
 for external links, 26
 for file types, 27–28
 image masks, 104–106, 160
IE8 (Internet Explorer 8), 2, 54
image-rect property, 103
 browser support, 106
images. *See also* background images
 for borders, 111–114
 future possibilities, 237–241
 image slices, 238–239
 image sprites, 103, 239–241
 providing fallback, 238
 tiling, 102
image sprites, 103, 239–241
Image Values and Replaced Content
 Module, 131, 237, 239
 browser support, 146
img element
 break-before for, 91–92
 inside column layout, 90
Impact font, 59–60
implicit animation, 164
implicit grids, 234–236
@import rule, 11, 244
indent at paragraph start, 85
infinite keyword, for animation-iteration-
 count property, 175
inheritance, and key frames, 173
inline-axis keyword, for box-orient
 property, 204
inline-box, for display property, 198
inner-center argument, for radial
 gradient, 137
inner-radius argument, for radial
 gradient, 137
inset keyword, for box-shadow property,
 116–117

internal links, 42
Internet Explorer. *See also* -ms- prefix
 2D Transformations Module, 147
 and artificial font faces, 53
 and calc() function, 232
 and local() value, 54
 and multiple backgrounds, 96
 support. *See* browser support
 transform property, 148
 web resources, 259
Internet Explorer 5.5, page layout, 7
Internet Explorer 8 (IE8), 2, 54
iOS, device-width for, 16
iPhone, Media Query for, 15–16
irregular quarter ellipse, 108, 109,
 110–111
italic text, 40–41, 53

J

JavaScript, 57
 cross-browser flex box with, 211–212
 library, 211
 Modernizr, 260
 setting up, 216
 for simulating Template Layout
 Module, 215
jQuery, 211, 216
justifying text, 85
justify keyword, for box-pack property,
 208, 209

K

kerning, 78
kern parameter, 62
@keyframes rule, 172
KHTML layout engine, 195

L

landscape value, for orientation Media Fea-
 ture, 17–18
lang attribute, rules applied based on, 25
Language Attribute Selector, 25
Last Call status, 3
last-child pseudo-class, 40–41
last-of-type pseudo-class, 40–41
left keyword
 for border-radius property, 108
 for perspective-origin property, 190
legibility, vs. speed, optimization, 77–79
letterpress effect, 71
licensing fonts for web use, 57
ligatures, 62–63, 78

- linear animation, 167
- linear gradients, 132–136
 - adding color-stop values, 135–136
 - in Firefox, 132–133, 143–144
 - using, 134–135
 - in WebKit, 133–134
- linear keyword
 - for animation-timing function property, 174
 - for transition-timing-function property, 166
- lines, between columns, 89
- link element, 11
 - media attribute, 9
 - order for stylesheets, 17
- links. *See* hyperlinks
- local fonts, for @font-face rule, 54–55
- logic attribute, for @media rule, 12

M

- Magnet Studio's Beginners Guide to OpenType, 61
- magnification, and quality, 19
- Marcotte, Ethan, "Responsive Web Design," 11
- margin, 84
 - between paragraphs, 85
 - hanging punctuation into, 79
- margin-bottom property, 85
- mask-box-image value, 160
- masking, 104
- mathematical operations
 - future possibilities, 230–233
 - for nth-* pseudo-classes, 35
- matrices, transformations with, 156–159
- matrix3d function, 187–188
- matrix function, 156–159, 187
- max-content keyword, for column widths, 225
- max-device-pixel-ratio, 19
- max() function, 232
- max-height, for Media Feature, 14
- max-width of viewport, 13
- media attribute, 9, 12
- Media Queries Module, 9–21
 - advantages, 10–11
 - browser support, 21
 - media features, 12–21
 - aspect ratio, 18–19
 - chaining multiple queries, 20
 - device width and height, 15–16
 - orientation, 17–18
 - pixel rate, 19–20
 - real world use, 16–17
 - width and height, 13–14
 - syntax, 11–12
 - @media rule, 12
- Microsoft Developer Network, CSS Compatibility page, 259
- min-content keyword, for column widths, 225
- min-device-pixel-ratio, 19
- min() function, 232
- min-height, for Media Feature, 14
- minmax function, 225
- min-width of viewport, 13
- @mix rule, 245
- mixins, extending variables using, 245–246
- mobile features, web browsing with, 10
- Modernizr, 260
- @module rule, 246
- modules, 2–3
 - future possibilities, 246–247
 - and recommendation process, 3–4
 - web resources, 257–258
- mod value, 230
- moving
 - element along axis, 185
 - elements from default position, 152–153
 - properties between states, 164
- Mozilla, and Web Open Font Format (WOFF), 55
- Mozilla Firefox. *See* Firefox
- moz- prefix, 5, 258
 - and background-clip property, 99, 101
 - and background-origin property, 101
 - and background-size property, 96
 - and border-radius property, 111
 - and box-sizing property, 6–7
 - and calculations, 232
 - and column layout, 82
 - and flexible box layout, 196, 198, 212
 - and gradients, 132,
 - linear, 132–133, 134–136
 - multiple, 141–142
 - radial, 137, 138–141
 - repeating, 142–145
 - and Grouping Selector, 242
 - and image-rect property, 103–104, 239
 - and Media Queries, 20, 21
 - and multicolored borders, 114–115
 - and OpenType fonts, 61–63
 - and transform property, 148, 157
 - and transitions, 164
- ms- prefix, 7, 8
 - and flexible box layout, 196, 212
 - and text-align property, 75–76
 - and transform property, 148, 157
- Multi-column Layout Module, 81
 - browser support, 92
- multiple keyword, for box-lines property, 211
- multiplier, for nth-* pseudo-classes, 35

N

- named anchor, 42
- names of fonts, 51
- narrower keyword, for font-stretch property, 61
- navigation menu, horizontal or vertical display, 18
- negative value (-n) increment, 39
- negative value
 - for text-indent property, 79
 - for transition-delay property, 168
- nested rules, 247
- Nokia, 248
- none keyword
 - for punctuation-trim property, 79
 - for resize property, 75
 - for text-wrap property, 77
 - for transition-property property, 165
 - for -webkit-font-smoothing property, 78
- no-repeat keyword, for background-repeat property, 102
- normal keyword
 - for animation-direction property, 175
 - for box-direction property, 205
 - for font-stretch property, 61
 - for font-style property, 52
 - for text-wrap property, 77
 - for word-wrap property, 76
- not (negation) pseudo-class, 44–45
 - browser support, 48
- not value, for media rule logic attribute, 12
- nth-* pseudo-classes, 35–39
- nth-child pseudo-class, 36–39
- nth-last-child pseudo-class, 39
- nth-last-of-type pseudo-class, 39
- nth-of-type pseudo-class, 36–39
- null value, for local() to force font download, 55

O

- odd keyword, 36
- only-child pseudo-class, 41–42
- only keyword, for media rule logic attribute, 12
- only-of-type pseudo-class, 41–42
- only operator, 15–16
- opacity property, 119, 120–122
 - browser support, 130
 - vs. rgba property, 123
- OpenType fonts, 55, 61–63
 - browser support, 64
- Opera. *See also* -o- prefix
 - background-origin property, 101
 - background-size, 96

- border-clip property, 99
- border-image property, 111
- border-radius property, 111
 - and multiple backgrounds, 96
 - support. *See* browser support
 - and text shadows, 68
 - transform property, 148
 - web resources, 259
- operating system, matching appearance, 129–130
- o- prefix, 8
 - and text-overflow property, 74
 - and transform property, 148, 157
 - and transitions, 164
- optimization, speed or legibility, 77–79
- optimizelegibility keyword, for text-rendering property, 77
- optimizeSpeed keyword, for text-rendering property, 77
- orientation
 - for flexible boxes, 204–205
 - for Media Feature, 17–18
- origin, 66–67
 - of 2D transformations, 150–151
 - of 3D transformations, 191
- outer-center argument, for radial gradient, 137
- outer-radius argument, for radial gradient, 137
- outlines for text, 65
- overflow property, 211, 223
- overflow text, restricting, 73–74

P

- padding, 6, 84
- padding-box keyword
 - for background-clip property, 98–99
 - for background-origin property, 100
 - for border-box property, 6
- page load time, background image and, 17
- paragraphs, indent at start, 85
- Partial Attribute Value Selector, 24–25
- paused keyword, for animation-play-state property, 176
- percentages, in element width calculation, 5
- Perfection Kills blog, 260
- period (.), as spacer for empty slot, 223
- perspective function, 188–190
- pixel grid, 67
- pixel rate, 19–20
 - detection, 20
- pixels, for axes measurement, 66
- point value, for gradient, 132–133

- portrait value, for orientation Media Feature, 17–18
- position, of background images, 98–101
- position property, for creating rows, 216–219
- precedence, for properties, 44
- prefixes
 - browser-specific, 7–8
 - for properties, 5
- presentation, vs. behavior, 163
- preserve-3d keyword, for transform-style property, 182
- properties
 - of fonts, 59–61
 - moving between states, 164
 - potential problems from unprefixed, 8
 - precedence for, 44
 - prefix for, 5
- Proposed Recommendation status, 3
- protocols, link icon for, 26
- pseudo-classes, 33
 - browser support, 48
 - empty, 44
 - first-of-type, 40–41
 - last-child, 40–41
 - last-of-type, 40–41
 - not (negation), 44–45
 - nth-*, 35–39
 - nth-child, 36–39
 - nth-last-child, 39
 - nth-last-of-type, 39
 - nth-of-type, 36–39
 - only-child, 41–42
 - only-of-type, 41–42
 - root, 44
 - structural, 34–42
 - target, 42–43
- pseudo-elements, 33, 46–47
 - browser support, 48
 - double colon (::) for, 47
 - ::slot(), 220–223
- pseudo-selectors, 23
- PT Sans font, 61
- punctuation properties, 79
 - browser support, 80
- punctuation-trim property, 79
- px (CSS Pixel unit), 19

Q

- quarter ellipse, 108. *See also* irregular quarter ellipse
- “quirks” mode, 7
- Quirks Mode box model, 202
- Quirks Mode website, 259

R

- radial gradients, 136–141
 - multiple color-stop values, 140–141
 - using, 138–139
 - in Firefox, 137
 - in WebKit, 137, 141
- radial keyword, 137
- radius of quarter ellipse, 108
- readability, 84–86
- ready() event (jQuery), 216
- recommendation process, for
 - modules, 3–4
- Recommendation status, 3
- Red, Green, Blue (RGB) model, 119
 - HSL vs., 127
- Red, Green, Blue, Alpha (RGBA) model, 122
 - browser support, 130
 - and graceful degradation, 124–125
- reflection
 - of element, creating, 155
 - gradients with, 160–161
 - with WebKit, 159–161
- regular quarter ellipse, 108
- repeat() function, for complex grids, 234
- repeating gradients
 - browser support, 146
 - in Firefox, 142–145
- repeat keyword
 - for background-repeat property, 102
 - for border-image property, 112
- repeat-x keyword, for background-repeat property, 102
- repeat-y keyword, for background-repeat property, 102
- resize property, 74
 - browser support, 80
- reusable code, 245
- reverse keyword, for box-direction property, 205
- reverse play for animation, 175
- RGB (Red, Green, Blue) model, 119
 - HSL vs., 127
- rgba() color function, 71
- RGBA (Red, Green, Blue, Alpha) model, 122
 - browser support, 130
 - and graceful degradation, 124–125
- right keyword
 - for border-radius property, 108
 - for perspective-origin property, 190
- root pseudo-class, 44
 - browser support, 48
- rotate3d function, 184

- rotate function
 - for three dimensions, 183
 - for transform property, 149–151
- rotation of element, with skew, 154, 158
- rounded corners for borders, 108–111
- round keyword
 - for background-repeat property, 102
 - for border-image property, 112, 113
- rows
 - setting height and width, 223–226
 - Template Layout Module for creating, 216–219
- row strings, 217
 - on display property, 219
 - rules for, 220
- rules (lines), between columns, 89
- running keyword, for animation-play-state property, 176

S

Safari browser for Mac OS X.

See also WebKit

- local() value, font-name argument for, 54–55
- support, border-image property, 111
- and text shadows, 67
- same keyword, for position property, 219
- Samsung Galaxy S, pixel density, 19
- Scalable Vector Graphics (SVG)
 - font type, 55
 - language, 147
- scale3d function, 186
- scale function, 154–155
 - for 3D transformations, 186–187
- scaleZ function, 186
- scaling
 - background images, 96–98
 - elements, 157
- scope, and variables names, 246
- screen, width of, 15
- ::selection pseudo-element, 47
 - browser support, 48
- selectors, 5, 23–31
 - attribute, 24–29
 - Arbitrary Substring, 28–29
 - Beginning Substring, 25–27
 - chaining, 29
 - Ending Substring, 27–28
 - browser support, 31, 48
 - grouping, 241–242
 - nested, 247
 - ::slot() pseudo-element and, 221
- Selectors Level 3 Module, 23–24
 - browser support, 31

- semi-condensed keyword, for font-stretch property, 61
- semi-expanded keyword, for font-stretch property, 61
- setTemplateLayout function, 216
- shadows
 - for box elements, 115–117
 - for text, 65, 67–71
 - multiple, 70–71
- shape argument, for radial gradient, 137
- ­ HTML code, 85–86
- Simple Attribute Selector, 24
- sine function, 158
- single keyword, for box-lines property, 211
- size of elements
 - changing, 74–75, 154–155
 - dynamic resizing child boxes to fit parent, 199–201
- skew function, 153–154
- skewing element, with matrices, 157–158
- slices
 - for frame image, 112, 113
 - for images, 238–239
- ::slot() pseudo-element, 220–223
- slots, 220–223
 - creating empty, 223
 - spanning multiple rows, 222
- small-caps keyword, for font-variant property, 62
- Smashing Magazine*, 264
- smcp property, 62
- soft hyphens, adding, 85–86
- space keyword
 - for background-repeat property, 102
 - for border-image property, 112
- spacer for empty slot, period (.) as, 223
- speed, vs. legibility, optimization, 77–79
- spread value, negative, for box-shadow property, 116
- @sprite rule, 239
- src property
 - for @font-face rule, 55–56
 - for font file location, 50
 - local() value, 54
- stack, for @font-face rule src property, 55–56
- standardization process, 1
- start keyword
 - for box-align property, 208
 - for hanging-punctuation property, 79
 - for punctuation-trim property, 79
 - for text-align property, 75
- states, moving properties between, 164
- stretch keyword
 - for border-image property, 112, 113
 - for box-align property, 208

- structural pseudo-classes, 34–42
 - browser support, 48
- styles, for different media, 9
- stylesheets
 - external, for Media Query, 11
 - order of mobile or desktop, 17
 - repetition in, 241–242
- subpixel-antialiased keyword, for `-webkit-font-smoothing` property, 78
- suppress value, for `text-wrap` property, 77
- SVG (Scalable Vector Graphics)
 - font type, 55
 - language, 147
- syntax, 4–8
- system fonts, “web-safe,” 49

T

- table element, 234
- tables for weather forecast, 38
- `tan` (tangent) function, 158
- target pseudo-class, 42–43
 - browser support, 48
 - as trigger, 171
- Template Layout Module, 215
 - browser support, 228
 - jQuery for, 216
 - multiple rows, 219–220
 - position and display to create rows, 216–219
 - slots and `::slot()` pseudo-element, 220–223
- testing
 - aspect ratio, 18–19
 - colors for accessibility, 71
- text
 - adding definition to, 72–73
 - aligning, 75–76
 - browser support for effects, 80
 - justifying, 85
 - punctuation properties, 79
 - restricting overflow, 73–74
 - shadows, 65, 67–71
 - letterpress effect, 71
 - multiple, 70–71
 - wrapping, 76–77
- `text-align-last` property, 75
 - browser support, 80
- `text-align` property, 75–76, 85
 - browser support, 80
- textarea element
 - disabled attribute of, 45
 - resizing, 75
- `text-fill-color` property, 72
- `text-indent` property, 79
- text keyword, for `border-clip` property, 100
- Text Level 3 Module, 65
 - browser support, 80
- `text-outline` property, 72–73
 - browser support, 80
- text-outlining, 65
- `text-overflow` property, 74
 - browser support, 80
- `text-rendering` property, 77–79
 - browser support, 80
- `text-shadow` property, 67–71
 - alpha value, 124
 - `blur-radius` option for, 69–70, 71
 - browser support, 80
- `text-stroke-color` property, 72
- `text-stroke` property, 72–73
 - browser support, 80
- `text-stroke-width` property, 72, 73
- `text-wrap` property, 77
 - browser support, 80

- thead element, selecting children, 39
- three-dimensional. *See* 3D elements; 3D transformations
- tiling images, 102
- time requirement for transition, 165
- `to()` function, 133, 137
- top keyword
 - for `border-radius` property, 108
 - for `perspective-origin` property, 190
- to-stop, in gradient, 132
- touchscreen devices
 - applying rules to elements on, 21
 - haptic feedback, 248
- transformations. *See also* 3D transformations
 - with matrices, 156–159
 - multiple, 156
- `transform-origin` property, 150–151
 - for 3D transformations, 191–192
- `transform` property, 148
 - for 3D transformations, 182–190
 - and element position in document flow, 149–150
 - rotate function, 149–151
 - skew function, 153–154
 - translate function, 152–153
- `transform-style` property, 182
- transitions, 164–172. *See also* animations
 - adding multiple, 170–171
 - browser support, 178
 - complete example, 169–170
 - shorthand, 169
 - `transition-delay` property, 168
 - `transition-duration` property, 165

- transition-property property, 165, 170
- transition-timing-function property,
 - 166–168
 - cubic-bezier function, 166–168
 - keywords, 166
 - triggers, 171–172
- Transitions Module, 163, 164
 - browser support, 178
- translate3d function, 185, 186
- translate function, 152–153, 157
- translateX function, 152, 185–186
- translateY function, 152, 185–186
- translateZ function, 185, 186
- transparency, opacity property for, 120–122
- triggers, for transitions, 171–172
- TrueType font format, 55
- Type Folly, 262
- Typekit, 57, 262
- type selectors, 23
- Typotheque, 57

U

- Ubuntu, buttons in WebKit, 129–130
- UI element states, 45–46
 - browser support, 48
- ultra-condensed keyword, for font-stretch property, 61
- ultra-expanded keyword, for font-stretch property, 61
- underline text, 40–41
- unprefixed property, potential problems, 8
- unrestricted value, for text-wrap property, 77
- url() notation, 238, 241
- @use directive, 247

V

- Values and Units Module, 230
- variables
 - currentColor, 127–129
 - extending using mixins, 245–246
 - future possibilities, 242–244
 - sprites assigned, 241
 - WebKit alternatives, 245
- @variables rule, 243
- var() notation, 243
- @var rule, 245
- Vector Markup Language (VML), 261
- vertical-align property, 222, 223
- vertical keyword, for box-orient property, 204
- vertical value
 - for aspect ratio, 19
 - for resize property, 75

- viewpoint, from perspective function, 189
- visible keyword, for backface-visibility property, 193
- visual impairments, letterpress effect and, 71
- VML (Vector Markup Language), 261

W

- weather forecast table, 38
- Web FontFonts, 263
- web fonts, 49–64
 - browser support for, 64
 - real-world example, 57–58
 - web resources, 262–264
- WebKit, 3
 - algorithms for column calculations, 86–87
 - and artificial font faces, 53
 - background-origin property, 100–101
 - border-clip property, 99
 - border overlap issue, 125
 - border-radius property, 108, 111
 - box-lines property, 211
 - button appearances on Ubuntu, 129–130
 - CSS extensions, 245–247
 - and multiple backgrounds, 96
 - support. *See* browser support
 - transform-origin subproperties, 192
 - variables, alternatives, 245
 - web resources, 258
- webkit- prefix, 5, 258
 - and 3D transformations, 179
 - and animations, 172, 173
 - and background-clip property, 99–101
 - and background-size property, 96
 - and border-radius property, 111
 - and column layout, 82, 88
 - and flexible box layout, 196, 198, 212
 - and gradients, 132
 - linear, 133–136
 - radial, 137, 138–141
 - and image masks, 105–106
 - and pixel ratio, 19–20
 - and reflections, 159–161
 - and text rendering, 78
 - and text-stroke property, 72–73
 - and transform property, 148, 157
 - and transitions, 164
- Web Open Font Format (WOFF), 55, 57
- web pages, layers, 163
- web resources, 257–264
 - on 3D matrix, 187
 - on 3D transforms, 179
 - 456 Berea Street blog, 46

- web resources (*continued*)
 - on angle argument, 133
 - book website, 181, 252
 - browsers, 258–259
 - browser support, 259–260
 - on Cartesian coordinate system, 180
 - Chunk font, 50
 - code-generation tools, 261–262
 - CSS modules, 257–258
 - on cubic Bézier curves, 168
 - on direction vector calculation, 184
 - estimation tool for x-height ratio, 60
 - feature detection and simulation, 260–261
 - on flexible box layout, 212
 - Flexie, 211
 - font foundries, 57
 - on fonts, 63
 - on Grid Positioning Module, 233–237
 - on image sprites, 103
 - Image Values and Replaced Content Module, 146, 237
 - jQuery, 211
 - Magnet Studio’s Beginners Guide to OpenType, 61
 - on matrices, 156
 - Media Queries gallery, 11
 - on Quirks Mode box model, 202
 - on Template Layout Module, 215, 228
 - on transitions, 165
 - trigonometric functions, 157
 - web fonts, 262–264
 - “web-safe” system fonts, 49
 - web use, licensing fonts for, 57

- Westciv, 181
- When Can I Use, 259
- wider keyword, for font-stretch property, 61
- widescreen, query for multiple options, 20
- width
 - of columns and rows, setting, 223–226
 - total for element, 4–5
- width Media Feature, 12
- windows, decorative header for browser, 13–14
- WOFF (Web Open Font Format), 55, 57
- word-wrap property, 76–77
 - browser support, 80
- Working Draft status, 3
- wrapping text, 76–77

X

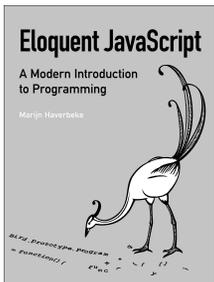
- x-axis, 66
- x-height, 59
- x-offset, 68, 71

Y

- y-axis, 66
- y-offset, 68, 71

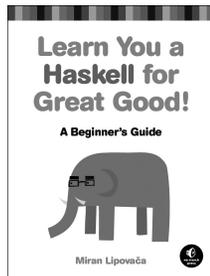
Z

- z-axis, 179, 181
- zebra striping for tables, 38–39



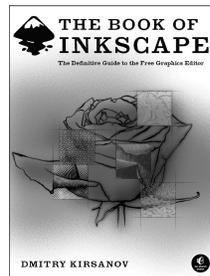
ELOQUENT JAVASCRIPT
A Modern Introduction to Programming

by MARIJN HAVERBEKE
JANUARY 2011, 224 PP., \$29.95
ISBN 978-1-59327-282-1



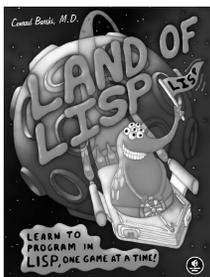
LEARN YOU A HASKELL
FOR GREAT GOOD!

A Beginner's Guide
by MIRAN LIPOVAČA
APRIL 2011, 400 PP., \$44.95
ISBN 978-1-59327-283-8



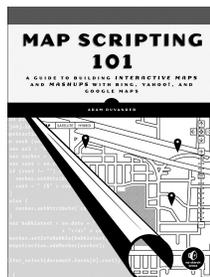
THE BOOK OF INKSCAPE
The Definitive Guide to the Free Graphics Editor

by DMITRY KIRSANOV
SEPTEMBER 2009, 472 PP., \$44.95
ISBN 978-1-59327-181-7



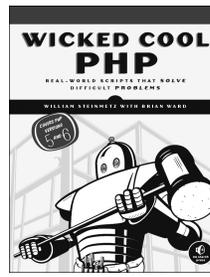
LAND OF LISP
Learn to Program in Lisp,
One Game at a Time!

by CONRAD BARSKI, M.D.
OCTOBER 2010, 504 PP., \$49.95
ISBN 978-1-59327-281-4



MAP SCRIPTING 101
An Example-Driven Guide to Building
Interactive Maps with Bing, Yahoo!,
and Google Maps

by ADAM DUVANDER
AUGUST 2010, 376 PP., \$34.95
ISBN 978-1-59327-271-5



WICKED COOL PHP
Real-World Scripts That Solve
Difficult Problems

by WILLIAM STEINMETZ with BRIAN WARD
FEBRUARY 2008, 216 PP., \$29.95
ISBN 978-1-59327-173-2

PHONE:
800.420.7240 OR
415.863.9900
MONDAY THROUGH FRIDAY,
9 A.M. TO 5 P.M. (PST)

EMAIL:
SALES@NOSTARCH.COM

WEB:
WWW.NOSTARCH.COM

UPDATES: Visit <http://nostarch.com/css3.htm> for updates, errata, and other information.

ABOUT THE AUTHOR

Peter has been a professional web developer for 11 years, starting at the height of the dot-com boom. He has worked freelance and full time for agencies and corporations, for clients including Orange, Skype, Cisco Systems, and the soccer club he passionately follows, Arsenal. He now works for digital agency Poke in Shoreditch, London.

He specializes in frontend development, mostly HTML, CSS, and JavaScript, and is a firm proponent of web standards and semantic markup. He keeps his own blog about web technologies at <http://www.broken-links.com/>, was a long-time writer at <http://www.css3.info/>, and has written for *Dev.Opera* and the UK web magazine *.net* (known in the US as *Practical Web Design*). He has given talks at London's web development community meetings and other public events, and aims to do more of this in the future.

Peter was born in Bristol but has lived in London for 16 years, the last seven of them with his wife, Ana. He loves to read, anything from literature to history (especially natural history and evolution) to psychology and is a big fan of independent comics and film.

This is his first book.

ABOUT THE TECHNICAL REVIEWER

Joost de Valk is a well-known specialist in the fields of search engine optimization (SEO), web design, and web development, and often speaks on these topics. Currently, Joost is a freelance consultant in SEO, web development, and online marketing strategy, working for such clients as eBay, RTL, salesforce.com, and the European Patent Office. He has built many a plugin for WordPress (with over 3.5 million downloads), hosts the weekly WordPress Podcast, and blogs about all that and more on his blog, <http://www.yoast.com/>.

THE FUTURE OF
WEB DESIGN
IS NOW



CSS3 is behind most of the eye-catching visuals on the Web today, but the official documentation can be dry and hard to follow and browser implementations are scattershot at best.

The Book of CSS3 distills the dense technical language of the CSS3 specification into plain English and shows you what CSS3 can do right now, in all major browsers. With real-world examples and a focus on the principles of good design, it extends your CSS skills, helping you transform ordinary markup into stunning, richly-styled web pages.

You'll master the latest cutting-edge CSS3 features and learn how to:

- Stylize text with fully customizable outlines, drop shadows, and other effects
- Create, position, and resize background images on the fly
- Spice up static web pages with event-driven transitions and animations

- Apply 2D and 3D transformations to text and images
- Use linear and radial gradients to create smooth color transitions
- Tailor a website's appearance to smartphones and other devices

A companion website includes up-to-date browser compatibility charts and live CSS3 examples for you to explore.

The Web can be an ugly place—add a little style to it with *The Book of CSS3*.

ABOUT THE AUTHOR

Peter Gasston has been a web developer for over 10 years in both agency and corporate settings. He was one of the original contributors to the website CSS3.info, has been published in the UK's *.net* magazine, and runs the web technology blog Broken Links (<http://www.broken-links.com/>). He lives in London, England.



THE BOOK OF CSS3

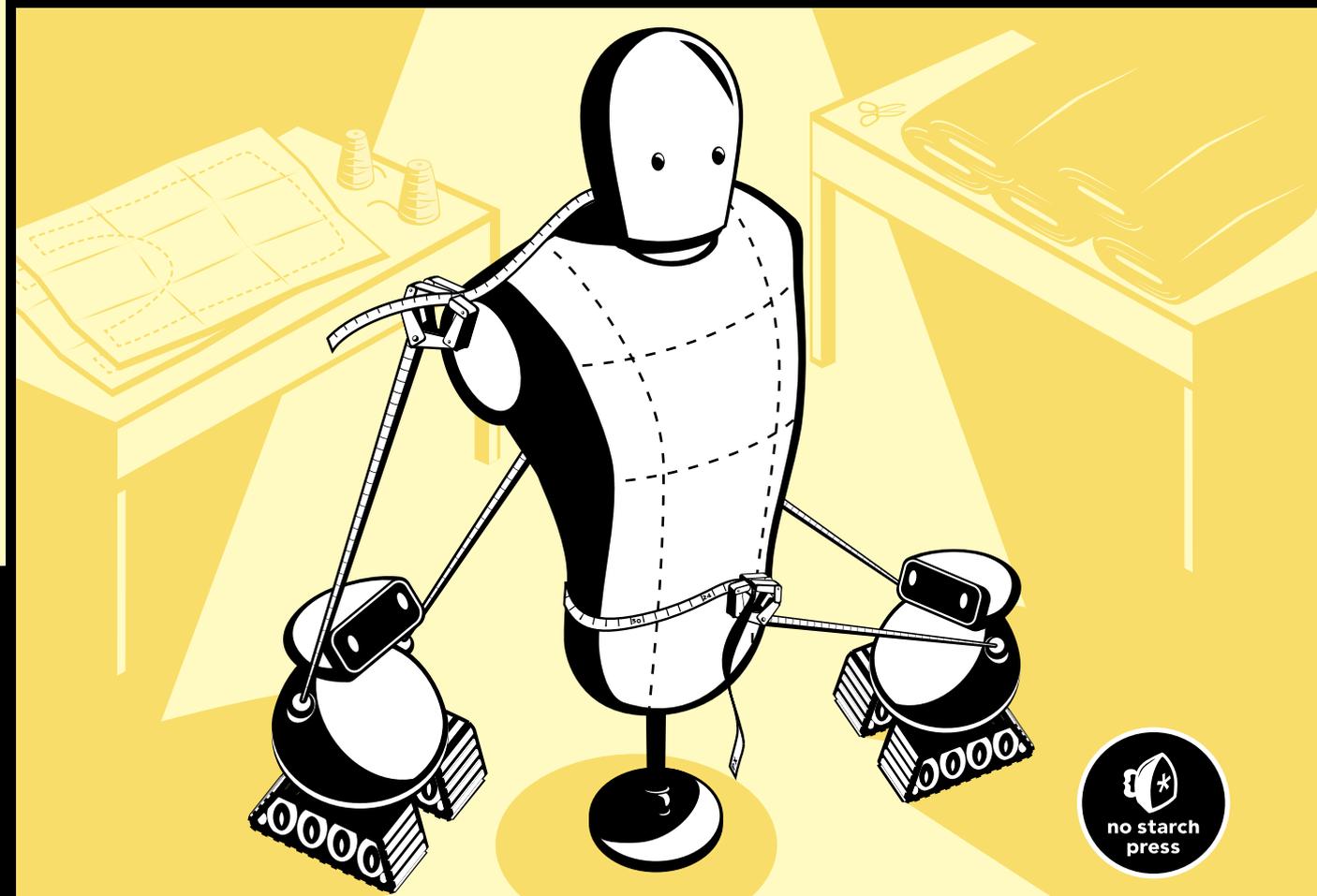
GASSTON



THE BOOK OF CSS3

A DEVELOPER'S GUIDE TO THE
FUTURE OF WEB DESIGN

PETER GASSTON



THE FINEST IN GEEK ENTERTAINMENT™

www.nostarch.com

OTABIND "I LIE FLAT."

This book uses a lay-flat binding that won't snap shut.

ISBN: 978-1-59327-286-9



9 781593 272869



5 3 4 9 5

\$34.95 (\$40.95 CDN)



6 89145 72863 7

SHelve IN:
COMPUTERS / INTERNET /
WEB DEVELOPMENT