INTRODUCING

HTML 5

SECOND EDITION

BRUCE LAWSON
REMY SHARP

INTRODUCING HTML 5

SECOND EDITION

New Riders VOICES THAT MATTER™

BRUCE LAWSON
REMY SHARP

Introducing HTML5, Second Edition

Bruce Lawson and Remy Sharp

New Riders
1249 Eighth Street
Berkeley, CA 94710
510/524-2178
510/524-2221 (fax)

Find us on the Web at: www.newriders.com
To report errors, please send a note to errata@peachpit.com

New Riders is an imprint of Peachpit, a division of Pearson Education

Copyright © 2012 by Remy Sharp and Bruce Lawson

Project Editor: Michael J. Nolan
Development Editor: Margaret S. Anderson/Stellarvisions
Technical Editors: Patrick H. Lauke (www.splintered.co.uk),
Robert Nyman (www.robertnyman.com)
Production Editor: Cory Borman
Copyeditor: Gretchen Dykstra
Proofreader: Jan Seymour
Indexer: Joy Dean Lee
Compositor: Danielle Foster
Cover Designer: Aren Howell Straiger
Cover photo: Patrick H. Lauke (splintered.co.uk)

Notice of Rights
All rights reserved. No part of this book may be reproduced or transmitted in any form by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. For information on getting permission for reprints and excerpts, contact permissions@peachpit.com.

Notice of Liability
The information in this book is distributed on an "As Is" basis without warranty. While every precaution has been taken in the preparation of the book, neither the authors nor Peachpit 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 instructions contained in this book or by the computer software and hardware products described in it.

Trademarks
Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Peachpit was aware of a trademark claim, the designations appear as requested by the owner of the trademark. All other product names and services identified throughout this book are used in editorial fashion only and for the benefit of such companies with no intention of infringement of the trademark. No such use, or the use of any trade name, is intended to convey endorsement or other affiliation with this book.

ISBN 13: 978-0-321-78442-1
ISBN 10: 0-321-78442-1

9 8 7 6 5 4 3 2 1

Printed and bound in the United States of America

ACKNOWLEDGEMENTS

Huge thanks to coauthor-turned-friend Remy Sharp, and friend-turned-ruthless-tech-editor Patrick Lauke: *il miglior fabbro*. At New Riders, Michael Nolan, Margaret Anderson, Gretchen Dykstra, and Jan Seymour deserve medals for their hard work and their patience.

Thanks to the Opera Developer Relations Team, particularly the editor of dev.opera.com, Chris Mills, for allowing me to reuse some materials I wrote for him, Daniel Davis for his description of `<ruby>`, Shwetank Dixit for checking some drafts, and David Storey for being so knowledgeable about Web Standards and generously sharing that knowledge. Big shout to former team member Henny Swan for her support and lemon cake. Elsewhere in Opera, the specification team of James Graham, Lachlan Hunt, Philip Jägenstedt, Anne van Kesteren, and Simon Pieters checked chapters and answered 45,763 daft questions with good humour. Nothing in this book is the opinion of Opera Software ASA.

Ian Hickson has also answered many a question, and my fellow HTML5 doctors (**www.html5doctor.com**) have provided much insight and support.

Many thanks to Richard Ishida for explaining `<bdi>` to me and allowing me to reproduce his explanation. Also to Aharon Lanin. Smoochies to Robin Berjon and the Mozilla Developer Center who allowed me to quote them.

Thanks to Gez Lemon and mighty Steve Faulkner for advice on WAI-ARIA. Thanks to Denis Boudreau, Adrian Higginbotham, Pratik Patel, Gregory J. Rosmaita, and Léonie Watson for screen reader advice.

Thanks to Stuart Langridge for drinkage, immoral support, and suggesting the working title “HTML5 Utopia.” Mr. Last Week’s creative vituperation provided loadsalaffs. Thanks, whoever you are.

Thanks to John Allsopp, Tantek Çelik, Christian Heilmann, John Foliot, Jeremy Keith, Matt May, and Eric Meyer for conversations about the future of markup. Silvia Pfeiffer’s blog posts on multimedia were invaluable to my understanding.

Stu Robson braved IE6 to take the screenshot in Chapter 1, Terence Eden took the BlackBerry screenshots in Chapter 3, Julia Gosling took the photo of Remy's magic HTML5 moustache in Chapter 4, and Jake Smith provided valuable feedback on early drafts of my chapters. Lastly, but most importantly, thanks to the thousands of students, conference attendees, and Twitter followers for their questions and feedback.

This book is in memory of my grandmothers, Marjorie Whitehead, 8 March 1917–28 April 2010, and Elsie Lawson 6 June 1920–20 August 2010.

This book is dedicated to Nongyaw, Marina, and James, without whom life would be monochrome.

—Bruce Lawson

Über thanks to Bruce who invited me to coauthor this book and without whom I would have spent the early part of 2010 complaining about the weather instead of writing this book. On that note, I'd also like to thank Chris Mills for even recommending me to Bruce.

To Robert Nyman, my technical editor: when I was in need of someone to challenge my JavaScript, I knew there would always be a Swede at hand. Thank you for making sure my code was as sound as it could be. Equally to Patrick Lauke, who also whipped some of my code, and certainly parts of my English, into shape.

Thanks to the local Brighton cafés, Coffee@33 and Café Délice, for letting me spend so many hours writing this book and drinking your coffee.

To my local Brighton digital community and new friends who have managed to keep me both sane and insane over the last few years of working alone. Thank you to Danny Hope, Josh Russell, and Anna Debenham for being my extended colleagues.

Thank you to Jeremy Keith for letting me rant and rail over HTML5 and bounce ideas, and for encouraging me to publish my thoughts. Equal thanks to Jessica for letting us talk tech over beers!

To the HTML5 Doctors and Rich Clark in particular for inviting me to contribute—and also to the team for publishing such great material.

To the whole #jquery-ot channel for their help when I needed to debug, or voice my frustration over a problem, and for being someplace I could go rather than having to turn to my cats for JavaScript support.

To the #whatwg channel for their help when I had misinterpreted the specification and needed to be put back on the right path. In particular to Anne Van Kesteren, who seemed to always have the answers I was looking for, perhaps hidden under some secret rock I'm yet to discover.

To all the conference organisers that invited me to speak, to the conference goers that came to hear me ramble, to my Twitter followers that have helped answer my questions and helped spur me on to completing this book with Bruce: thank you. I've tried my best with the book, and if there's anything incorrect or out of date: ~~blame Bruce~~ buy the next edition. ;-)

To my wife, Julie: thank you for supporting me for all these many years. You're more than I ever deserved and without you, I honestly would not be the man I am today.

Finally, this book is dedicated to Tia. My girl. I wrote the majority of my part of this book whilst you were on our way to us. I always imagined that you'd see this book and be proud and equally embarrassed. That won't happen now, and even though you're gone, you'll always be with us and never forgotten.

—Remy Sharp

CONTENTS

INTRODUCTION

Welcome to the second edition of the Remy & Bruce show. Since the first edition of this book came out in July 2010, much has changed: support for HTML5 is much more widespread; Internet Explorer 9 finally came out; Google Chrome announced it would drop support for H.264 video; Opera experimented with video streaming from the user's webcam via the browser, and HTML5 fever became HTML5 hysteria with any new technique or technology being called HTML5 by clients, bosses, and journalists.

All these changes, and more, are discussed in this shiny second edition. There is a brand new Chapter 12 dealing with the realities of implementing all the new technologies for old browsers. And we've corrected a few bugs, tweaked some typos, rewritten some particularly opaque prose, and added at least one joke.

We're two developers who have been playing with HTML5 since Christmas 2008—experimenting, participating in the mailing list, and generally trying to help shape the language as well as learn it.

Because we're developers, we're interested in building things. That's why this book concentrates on the problems that HTML5 can solve, rather than on an academic investigation of the language. It's worth noting, too, that although Bruce works for Opera Software, which began the proof of concept that eventually led to HTML5, he's not part of the specification team there; his interest is as an author using the language for an accessible, easy-to-author, interoperable Web.

Who's this book for?

No knowledge of HTML5 is assumed, but we do expect that you're an experienced (X)HTML author, familiar with the concepts of semantic markup. It doesn't matter whether you're more familiar with HTML or XHTML DOCTYPEs, but you should be happy coding any kind of strict markup.

While you don't need to be a JavaScript ninja, you should have an understanding of the increasingly important role it plays in modern web development, and terms like DOM and API won't make you drop this book in terror and run away.

Still here? Good.

What this book isn't

This is not a reference book. We don't go through each element or API in a linear fashion, discussing each fully and then moving on. The specification does that job in mind-numbing, tear-jerking, but absolutely essential detail.

What the specification doesn't try to do is teach you how to use each element or API or how they work with one another, which is where this book comes in. We'll build up examples, discussing new topics as we go, and return to them later when there are new things to note.

You'll also realise, from the title and the fact that you're comfortably holding this book without requiring a forklift, that this book is not comprehensive. Explaining a 700-page specification (by comparison, the first HTML spec was three pages long) in a medium-sized book would require Tardis-like technology (which would be cool) or microscopic fonts (which wouldn't).

What do we mean by HTML5?

This might sound like a silly question, but there is an increasing tendency amongst standards pundits to lump all exciting new web technologies into a box labeled HTML5. So, for example, we've seen SVG (Scalable Vector Graphics) referred to as "one of the HTML5 family of technologies," even though it's an independent W3C *graphics* spec that's ten years old.

Further confusion arises from the fact that the official W3C spec is something like an amoeba: Bits split off and become their own specifications, such as Web Sockets or Web Storage (albeit from the same Working Group, with the same editors).

So what we mean in this book is "HTML5 and related specifications that came from the WHATWG" (more about this exciting acronym soon). We're also bringing a "plus one" to the party—Geolocation—which has nothing to do with our definition of HTML5, but which we've included for the simple reason that it's really cool, we're excited about it, and it's part of NEWT: the *New Exciting Web Technologies*.

Who? What? When? Why? A short history of HTML5

History sections in computer books usually annoy us. You don't need to know about ARPANET or the history of HTTP to understand how to write a new language.

Nevertheless, it's useful to understand how HTML5 came about, because it will help you understand why some aspects of HTML5 are as they are, and hopefully preempt (or at least soothe) some of those "WTF? Why did they design it like *that*?" moments.

How HTML5 nearly never was

In 1998, the W3C decided that they would not continue to evolve HTML. The future, they believed (and so did your authors) was XML. So they froze HTML at version 4.01 and released a specification called XHTML 1.0, which was an XML version of HTML that required XML syntax rules such as quoting attributes, closing some tags while self-closing others, and the like. Two flavours were developed (well, actually three, if you care about HTML Frames, but we hope you don't because they're gone from HTML5). XHTML Transitional was designed to help people move to the gold standard of XHTML Strict.

This was all tickety-boo—it encouraged a generation of developers (or at least the professional-standard developers) to think about valid, well-structured code. However, work then began on a specification called XHTML 2.0, which was a revolutionary change to the language, in the sense that it broke backwards-compatibility in the cause of becoming much more logical and better-designed.

A small group at Opera, however, was not convinced that XML was the future for all web authors. Those individuals began extracurricular work on a proof-of-concept specification that extended HTML forms without breaking backward-compatibility. That spec eventually became Web Forms 2.0, and was subsequently folded into the HTML5 spec. They were quickly joined by individuals from Mozilla and this group, led by Ian "Hixie" Hickson of Opera, continued working on the specification privately with Apple "cheering from the sidelines" in a small group that called itself the WHATWG (Web Hypertext Application Technology Working Group, **www.whatwg.org**). You can see

this genesis still in the copyright notice on the WHATWG version of the spec "© Copyright 2004–2011 Apple Computer, Inc., Mozilla Foundation, and Opera Software ASA (note that you are licensed to use, reproduce, and create derivative works)."

Hickson moved to Google, where he continued to work full-time as editor of HTML5 (then called Web Applications 1.0).

In 2006 the W3C decided that they had perhaps been overly optimistic in expecting the world to move to XML (and, by extension, XHTML 2.0): "It is necessary to evolve HTML incrementally. The attempt to get the world to switch to XML, including quotes around attribute values and slashes in empty tags and namespaces, all at once didn't work," said Tim Berners-Lee.

The resurrected HTML Working Group voted to use the WHATWG's Web Applications spec as the basis for the new version of HTML, and thus began a curious process whereby the same spec was developed simultaneously by the W3C (co-chaired by Sam Ruby of IBM and Chris Wilson of Microsoft, and later by Ruby, Paul Cotton of Microsoft, and Maciej Stachowiak of Apple), and the WHATWG, under the continued editorship of Hickson.

In search of the spec

Because the HTML5 specification is being developed by both the W3C and WHATWG, there are different versions of it. Think of the WHATWG versions as being an incubator group.

The official W3C snapshot is **www.w3.org/TR/html5/**, while **http://dev.w3.org/html5/spec/** is the latest editor's draft and liable to change.

The WHATWG has dropped version numbers, so the "5" has gone; it's just "HTML,—the living standard." Find this at **http://whatwg.org/html** but beware there are hugely experimental ideas in there. Don't assume that because it's in this document it's implemented anywhere or even completely thought out yet. This spec does, however, have useful annotations about implementation status in different browsers.

There's a one-page version of the complete WHATWG specifications called "Web Applications 1.0" that incorporates everything from the WHATWG at **http://www.whatwg.org/specs/web-apps/current-work/complete.html** but it might kill your browser as it's massive with many scripts.

A lot of the specification is algorithms really intended for those implementing HTML (browser manufacturers, for example). The spec that we have bookmarked is a useful version for the Web at **http://developers.whatwg.org**, which removes all the stuff written for implementers and presents it with attractive CSS, courtesy of Ben Schwarz. This contains the experimental stuff, too.

Confused? **http://wiki.whatwg.org/wiki/FAQ#What_are_the_various_versions_of_the_spec.3F** lists and describes these different versions.

Geolocation is not a WHATWG spec. You can go to **http://www.w3.org/TR/geolocation-API/** to find it.

The process has been highly unusual in several respects. The first is the extraordinary openness; anyone could join the WHATWG mailing list and contribute to the spec. Every email was read by Hickson or the core WHATWG team (which included such luminaries as the inventor of JavaScript and Mozilla CTO Brendan Eich, Safari and WebKit Architect David Hyatt, and inventor of CSS and Opera CTO Håkon Wium Lie).

Good ideas were implemented and bad ideas rejected, regardless of who the source was or who they represented, or even where those ideas were first mooted. Additional good ideas were adopted from Twitter, blogs, and IRC.

In 2009, the W3C stopped work on XHTML 2.0 and diverted resources to HTML5 and it was clear that HTML5 had won the battle of philosophies: purity of design, even if it breaks backwards-compatibility, versus pragmatism and "not breaking the Web." The fact that the HTML5 working groups consisted of representatives from all the browser vendors was also important. If vendors were unwilling to implement part of the spec (such as Microsoft's unwillingness to implement `<dialog>`, or Mozilla's opposition to `<bb>`) it was dropped. Hickson has said, "The reality is that the browser vendors have the ultimate veto on everything in the spec, since if they don't implement it, the spec is nothing but a work of fiction." Many participants found this highly distasteful: Browser vendors have hijacked "our Web," they complained with some justification.

It's fair to say that the working relationship between W3C and WHATWG has not been as smooth as it could be. The W3C operates under a consensus-based approach, whereas Hickson continued to operate as he had in the WHATWG—as benevolent dictator (and many will snort at our use of the word *benevolent* in this context). It's certainly the case that Hickson had very firm ideas of how the language should be developed.

The philosophies behind HTML5

Behind HTML5 is a series of stated design principles (**http://www.w3.org/TR/html-design-principles**). There are three main aims to HTML5:

- Specifying current browser behaviours that are interoperable
- Defining error handling for the first time
- Evolving the language for easier authoring of web applications

Not breaking existing web pages

Many of our current methods of developing sites and applications rely on undocumented (or at least unspecified) features incorporated into browsers over time. For example, `XMLHttpRequest` (XHR) powers untold numbers of Ajax-driven sites. It was invented by Microsoft, and subsequently reverse-engineered and incorporated into all other browsers, but had never been specified as a standard (Anne van Kesteren of Opera finally specified it as part of the WHATWG). Such a vital part of so many sites left entirely to reverse-engineering! So one of the first tasks of HTML5 was to document the undocumented, in order to increase interoperability by leaving less to guesswork for web authors and implementors of browsers.

It was also necessary to unambiguously define how browsers and other user agents should deal with invalid markup. This wasn't a problem in the XML world; XML specifies "draconian error handling" in which the browser is required to stop rendering if it finds an error. One of the major reasons for the rapid ubiquity and success of the Web (in our opinion) was that even bad code had a fighting chance of being rendered by some or all browsers. The barrier to entry to publishing on the Web was democratically low, but each browser was free to decide how to render bad code. Something as simple as

```
<b><i>Hello mum!</b></i>
```

(note the mismatched closing tags) produces different DOMs in different browsers. Different DOMs can cause the same CSS to have a completely different rendering, and they can make writing JavaScript that runs across browsers much harder than it needs to be. A consistent DOM is so important to the design of HTML5 that the language itself is defined in terms of the DOM.

In the interest of greater interoperability, it's vital that error handling be identical across browsers, thus generating the exact same DOM even when confronted with broken HTML. In order for that to happen, it was necessary for someone to specify it. As we said, the HTML5 specification is well over 700 pages long, but only 300 or so are relevant to web authors (that's you and us); the rest of it is for implementers of browsers, telling them *exactly* how to parse markup, even bad markup.

Web applications

An increasing number of sites on the Web are what we'll call web applications; that is, they mimic desktop apps rather than traditional static text-images-links documents that make up the majority of the Web. Examples are online word processors, photo-editing tools, mapping sites, and so on. Heavily powered by JavaScript, these have pushed HTML 4 to the edge of its capabilities. HTML5 specifies new DOM APIs for drag and drop, server-sent events, drawing, video, and the like. These new interfaces that HTML pages expose to JavaScript via objects in the DOM make it easier to write such applications using tightly specified standards rather than barely documented hacks.

Even more important is the need for an open standard (free to use and free to implement) that can compete with proprietary standards like Adobe Flash or Microsoft Silverlight. Regardless of your thoughts on those technologies or companies, we believe that the Web is too vital a platform for society, commerce, and communication to be in the hands of one vendor. How differently would the Renaissance have progressed if Caxton held a patent and a monopoly on the manufacture of printing presses?

Don't break the Web

There are exactly umpty-squillion web pages already out there, and it's imperative that they continue to render. So HTML5 is (mostly) a superset of HTML 4 that continues to define how browsers should deal with legacy markup such as `<font>`, `<center>`, and other such presentational tags, because millions of web pages use them. But authors should not use them, as they're obsolete. For web authors, semantic markup still rules the day, although each reader will form her own conclusion as to whether HTML5 includes enough semantics, or too many elements.

As a bonus, HTML5's unambiguous parsing rules should ensure that ancient pages will work interoperably, as the HTML5 parser will be used for all HTML documents once it's implemented in all browsers.

What about XML?

HTML5 is not an XML language (it's not even an SGML language, if that means anything important to you). It *must* be served as text/html. If, however, you need to use XML, there is an XML serialisation called XHTML5. This allows all the same

features, but (unsurprisingly) requires a more rigid syntax (if you're used to coding XHTML, this is exactly the same as you already write). It *must* be well-formed XML and it *must* be served with an XML MIME type, even though IE8 and its antecedents can't process it (it offers it for downloading rather than rendering it). Because of this, we are using HTML rather than XHTML syntax in this book.

HTML5 support

HTML5 is moving very fast now. The W3C specification went to last call in May 2011, but browsers were implementing HTML5 support (particularly around the APIs) long before then. That support is going to continue growing as browsers start rolling out features, so instances where we say "this is only supported in browser X" will rapidly date—which is a good thing.

New browser features are very exciting and some people have made websites that claim to test browsers' HTML5 support. Most of them wildly pick and mix specs, checking for HTML5, related WHATWG-derived specifications such as Web Workers and then, drunk and giddy with buzzwords, throw in WebGL, SVG, the W3C File API, Media Queries, and some Apple proprietary whizbangs before hyperventilating and going to bed for a lie-down.

Don't pay much attention to these sites. Their point systems are arbitrary, their definition of HTML5 meaningless and misleading.

As Patrick Lauke, our technical editor, points out, "HTML5 is not a race. The idea is not that the first browser to implement all will win the Internet. The whole idea behind the spec work is that all browsers will support the same feature set consistently."

If you want to see the current state of support for *New Exciting Web Technologies*, we recommend **http://caniuse.com** by Alexis Deveria.

Let's get our hands dirty

So that's your history lesson, with a bit of philosophy thrown in. It's why HTML5 sometimes willfully disagrees with other specifications—for backwards-compatibility, it often defines what browsers actually do, rather than what an RFC document specifies they ought to do. It's why sometimes HTML5 seems like a kludge or a compromise—it is. And if that's the price we have to pay for an interoperable open Web, then your authors say, "*Viva* pragmatism!"

Got your seatbelt on?

Let's go.

CHAPTER 4

Video and Audio

Bruce Lawson and Remy Sharp

A LONG TIME AGO, in a galaxy that feels a very long way away, multimedia on the Web was limited to tinkling MIDI tunes and animated GIFs. As bandwidth got faster and compression technologies improved, MP3 music supplanted MIDI and real video began to gain ground. All sorts of proprietary players battled it out—Real Player, Windows Media, and so on—until one emerged as the victor in 2005: Adobe Flash, largely because of its ubiquitous plugin and the fact that it was the delivery mechanism of choice for YouTube.

HTML5 provides a competing, open standard for delivery of multimedia on the Web with its native video and audio elements and APIs. This chapter largely discusses the `<video>` element, as that's sexier, but most of the markup and scripting are applicable to `<audio>` as well.

Native multimedia: why, what, and how?

In 2007, Anne van Kesteren wrote to the Working Group:

"Opera has some internal experimental builds with an implementation of a `<video>` *element. The element exposes a simple API (for the moment) much like the* `Audio()` `object:` `play()`, `pause()`, `stop()`. *The idea is that it works like* `<object>` *except that it has special* `<video>` *semantics much like* `<img>` *has image semantics."*

While the API has increased in complexity, van Kesteren's original announcement is now implemented in all the major browsers, including Internet Explorer 9.

An obvious companion to a `<video>` element is an `<audio>` element; they share many similar features, so in this chapter we discuss them together and note only the differences.

<video>: Why do you need a <video> element?

Previously, if developers wanted to include video in a web page, they had to make use of the `<object>` element, which is a generic container for "foreign objects." Due to browser inconsistencies, they would also need to use the previously invalid `<embed>` element and duplicate many parameters. This resulted in code that looked much like this:

```
<object width="425" height="344">
<param name="movie" value="http://www.youtube.com/
¬ v/9sEI1AUFJKw&hl=en_GB&fs=1&"></param>
<param name="allowFullScreen"
value="true"></param>
<param name="allowscriptaccess"
value="always"></param>
<embed src="http://www.youtube.com/
¬ v/9sEI1AUFJKw&hl=en_GB&fs=1&"
type="application/x-shockwave-flash"
allowscriptaccess="always"
allowfullscreen="true" width="425"
height="344"></embed>
</object>
```

This code is ugly and ungainly. Worse still is the fact that the browser has to pass the video off to a third-party plugin; hope that the user has the correct version of that plugin (or has the rights to download and install it, and the knowledge of how to do so); and then hope that the plugin is keyboard accessible—along with all the other unknowns involved in handing the content to a third-party application.

Plugins can also be a significant cause of browser instability and can create worry for less technical users when they are prompted to download and install newer versions.

Whenever you include a plugin in your pages, you're reserving a certain drawing area that the browser delegates to the plugin. As far as the browser is concerned, the plugin's area remains a black box—the browser does not process or interpret anything that happens there.

Normally, this is not a problem, but issues can arise when your layout overlaps the plugin's drawing area. Imagine, for example, a site that contains a movie but also has JavaScript or CSS-based drop-down menus that need to unfold over the movie. By default, the plugin's drawing area sits on top of the web page, meaning that these menus will strangely appear behind the movie.

Problems and quirks can also arise if your page has dynamic layout changes. Resizing the dimensions of the plugin's drawing area can sometimes have unforeseen effects—a movie playing in the plugin may not resize, but instead simply may be cropped or display extra white space. HTML5 provides a standardised way to play video directly in the browser, with no plugins required.

> **NOTE** `<embed>` is finally standardised in HTML5; it was never part of any previous flavour of (X)HTML.

One of the major advantages of the HTML5 video element is that, finally, video is a full-fledged citizen on the Web. It's no longer shunted off to the hinterland of `<object>` or the nonvalidating `<embed>` element.

So now, `<video>` elements can be styled with CSS. They can be resized on hover using CSS transitions, for example. They can be tweaked and redisplayed onto `<canvas>` with JavaScript. Best of all, the innate hackability that open web standards provide is opened up. Previously, all your video data was locked away; your bits were trapped in a box. With HTML5 multimedia, your bits are free to be manipulated however you want.

What HTML5 multimedia isn't good for

Regardless of the sensationalist headlines of the tech journalists, HTML5 won't "kill" all plugins overnight. There are use-cases for plugins not covered by the new spec.

> **NOTE** If you're really, really anxious to do DRM, check out **http://lists.whatwg.org/htdig.cgi/whatwg-whatwg.org/2010-July/027051.html** for Henri Sivonen's suggested method, which requires no changes to the spec.

Copy protection is one area not dealt with by HTML5—unsurprisingly, given that it's a standard based on openness. So people who need digital rights management (DRM) are probably not going to want to use HTML5 video or audio, as they'll be as easy to download to a hard drive as an `<img>` is now. Some browsers offer simple context-menu access to the URL of the video, or even let the user save the video. Developers can view source, find the reference to the video's URL, and download it that way. (Of course, you don't need us to point out that DRM is a fool's errand, anyway. All you do is alienate your honest users while causing minor inconvenience to dedicated pirates.)

HTML5 can't give us adaptive streaming either. This is a process that adjusts the quality of a video delivered to a browser based on changes to network conditions to ensure the best experience. It's being worked on, but it isn't there yet.

Plugins currently remain the best cross-browser option for accessing the user's webcam or microphone and then transmitting video and audio from the user's machine to a web page such as Daily Mugshot or Chatroulette, although `getUserMedia` and WebRTC are in the cards for Chrome, Opera, and Firefox—see "Video conferencing, augmented reality" at the end of this chapter. After shuddering at the unimaginable loneliness that a world without Chatroulette would represent, consider also the massive amount of content already out there on the web that will require plugins to render it for a long time to come.

Anatomy of the video and audio elements

At its simplest, to include video on a page in HTML5 merely requires this code:

```
<video src=turkish.webm></video>
```

The .webm file extension is used here to point to a WebM-encoded video.

Similar to <object>, you can put fallback markup between the tags for older web browsers that do not support native video. You should at least supply a link to the video so users can download it to their hard drives and watch it later on the operating system's media player. **Figure 4.1** shows this code in a modern browser and fallback content in a legacy browser.

```
<h1>Video and legacy browser fallback</h1>
<video src=leverage-a-synergy.webm>
  Download the <a href=leverage-a-synergy.webm>How to
  ¬ leverage a synergy video</a>
</video>
```

FIGURE 4.1 HTML5 video in a modern browser and fallback content in a legacy browser.

However, this example won't actually do anything just yet. All you can see here is the first frame of the movie. That's because you haven't told the video to play, and you haven't told the browser to provide any controls for playing or pausing the video.

autoplay

While you can tell the browser to play the video or audio automatically once the web page is loaded, you almost certainly shouldn't, as many users (and particularly those using assistive technology, such as a screen reader) will find it highly intrusive. Users on mobile devices probably won't want you using their bandwidth without them explicitly asking for the video. Nevertheless, here's how you do it:

```
<video src=leverage-a-synergy.webm autoplay>
  <!-- your fallback content here -->
</video>
```

controls

Providing controls is approximately 764 percent better than autoplaying your video. See **Figure 4.2**. You can use some simple JavaScript to write your own (more on that later) or you can tell the browser to provide them automatically:

```
<video src=leverage-a-synergy.webm controls>
</video>
```

Naturally, these differ between browsers, as the spec doesn't prescribe what the controls should look like or do, but most browsers don't reinvent the wheel and instead have stuck to what has become the general norm for such controls—there's a play/pause toggle, a seek bar, and volume control.

Browsers have chosen to visually hide the controls, and only make them appear when the user hovers or sets focus on the controls via the keyboard. It's also possible to move through the different controls using only the keyboard. This native keyboard accessibility is already an improvement on plugins, which can be tricky to tab into from surrounding HTML content.

If the `<audio>` element has the `controls` attribute, you'll see them on the page. Without the attribute, you can hear the audio but nothing is rendered visually on the page at all; it is, of course, there in the DOM and fully controllable via JavaScript and the new APIs.

NOTE Browsers have different levels of keyboard accessibility. Firefox's native controls are right and left arrows to skip forward/back (up and down arrows after tabbing into the video), but there is no focus highlight to show where you are, and so no visual clue. The controls don't appear if the user has JavaScript disabled in the browser; so although the contextual menu allows the user to stop and start the movie, there is the problem of discoverability.

Opera's accessible native controls are always present when JavaScript is disabled, regardless of whether the `controls` attribute is specified.

IE9 has good keyboard accessibility. Chrome and Safari appear to lack keyboard accessibility. We anticipate increased keyboard accessibility as manufacturers iron out teething problems.

FIGURE 4.2 The default controls in Firefox. (These are similar in all modern browsers.)

poster

The **poster** attribute points to an image that the browser will use while the video is downloading, or until the user tells the video to play. (This attribute is not applicable to <audio>.) It removes the need for additional tricks like displaying an image and then removing it via JavaScript when the video is started.

If you don't use the poster attribute, the browser shows the first frame of the movie, which may not be the representative image you want to show.

The behavior varies somewhat on mobile devices. Mobile Safari does grab the first frame if no poster is specified; Opera Mobile conserves bandwidth and leaves a blank container.

muted

The **muted** attribute, a recent addition to the spec (read: "as yet, very little support"), gives a way to have the multimedia element muted by default, requiring user action to unmute it. This video (an advertisement) autoplays, but to avoid annoying users, it does so without sound, and allows the user to turn the sound on:

```
<video src="adverts.cgi?kind=video" controls autoplay loop
¬ muted></video>
```

height, width

The **height** and **width** attributes tell the browser the size of the video in pixels. (They are not applicable to <audio>.) If you leave them out, the browser uses the intrinsic width of the video resource, if that is available. Otherwise it uses the intrinsic width of the poster frame, if that is available. If neither is available, the browser defaults to 300 pixels.

If you specify one value but not the other, the browser adjusts the size of the unspecified dimension to preserve the video's aspect ratio.

If you set both width and height to an aspect ratio that doesn't match that of the video, the video is not stretched to those dimensions but is rendered letterboxed inside the video element of your specified size while retaining the aspect ratio.

loop

The **loop** attribute is another Boolean attribute. As you would imagine, it loops the media playback. Support is flaky at the moment, so don't expect to be able to have a short audio sample and be able to loop it seamlessly. Support will get better—browsers as media players is a new phenomenon.

preload

Maybe you're pretty sure that the user wants to activate the media (she's drilled down to it from some navigation, for example, or it's the only reason to be on the page), but you don't want to use `autoplay`. If so, you can suggest that the browser preload the video so that it begins buffering when the page loads in the expectation that the user will activate the controls.

```
<video src=leverage-a-synergy.ogv controls preload>
</video>
```

There are three spec-defined values for the **preload** attribute. If you just say `preload`, the user agent can decide what to do. A mobile browser may, for example, default to not preloading until explicitly told to do so by the user. It's important to remember that a web developer can't control the browser's behavior: `preload` is a hint, not a command. The browser will make its decision based on the device it's on, current network conditions, and other factors.

- `preload=auto` (or just `preload`)

  This is a suggestion to the browser that it should begin downloading the entire file.

- `preload=none`

  This state suggests to the browser that it shouldn't preload the resource until the user activates the controls.

- `preload=metadata`

  This state suggests to the browser that it should just prefetch metadata (dimensions, first frame, track list, duration, and so on) but that it shouldn't download anything further until the user activates the controls.

src

As on an `<img>`, the `src` attribute points to audio or video resource, which the browser will play if it supports the specific codec/container format. Using a single source file with the `src` attribute is really only useful for rapid prototyping or for intranet sites where you know the user's browser and which codecs it supports.

However, because not all browsers can play the same formats, in production environments you need to have more than one source file. We'll cover this in the next section.

> **NOTE** So long as the http endpoint is a streaming resource on the Web, you can just point the `<video>` or `<audio>` element at it to stream the content.

Codecs—the horror, the horror

Early drafts of the HTML5 specification mandated that all browsers should have built-in support for multimedia in at least two codecs: Ogg Vorbis for audio and Ogg Theora for movies. Vorbis is a codec used by services like Spotify, among others, and for audio samples in games like Microsoft Halo.

However, these requirements for default format support were dropped from the HTML5 spec after Apple and Nokia objected, so the spec makes no recommendations about codecs at all. This leaves us with a fragmented situation, with different browsers opting for different formats, based on their ideological and commercial convictions.

Currently, there are two main container/codec combinations that developers need to be aware of: the new WebM format (**www.webmproject.org**) which is built around the VP8 codec that Google bought for $104 million and open licensed, and the ubiquitous MP4 format that contains the royalty-encumbered H.264 codec. H.264 is royalty-encumbered because, in some circumstances, you must pay its owners if you post videos that use that codec. We're not lawyers so can't give you guidance on which circumstances apply to you. Go to **www.mpegla.com** and have your people talk to their people's people.

In our handy cut-out-and-lose chart (**Table 4.1**), we also include the Ogg Theora codec for historical reasons—but it's really only useful if you want to include support for older versions of browsers with initial `<video>` element support like Firefox 3.x and Opera 10.x.

NOTE At time of writing, Chrome still supports H.264 but announced it will be discontinued. Therefore, assume it won't be supported.

TABLE 4.1 Video codec support in modern browsers.

	WEBM (VP8 CODEC)	MP4 (H.264 CODEC)	OGV (OGG THEORA CODEC)
Opera	Yes	No	Yes
Firefox	Yes	No	Yes
Chrome	Yes	Yes—see Note, *support will be discontinued*	Yes
IE9 +	Yes (but codec must be installed manually)	Yes	No
Safari	No	Yes	No

Marvel at the amazing coincidence that the only two browsers that support H.264 are members of the organization that collects royalties for using the codec (**www.mpegla.com/main/programs/AVC/Pages/Licensors.aspx**).

A similarly fragmented situation exists with audio codecs, for similar royalty-related reasons (see **Table 4.2**).

TABLE 4.2 Audio codec support in modern browsers.

	.OGG/ .OGV (VORBIS CODEC)	MP3	MP4/ M4A (AAC CODEC)	WAV
Opera	Yes	No	No	Yes
Firefox	Yes	No	No	Yes
Chrome	Yes	Yes	Yes	Yes
IE9 +	No	Yes	Yes	No
Safari	No	Yes	Yes	Yes

NOTE It's possible to polyfill MP3 support into Firefox. JSmad (jsmad.org) is a JavaScript library that decodes MP3s on the fly and reconstructs them for output using the Audio Data API, although we wonder about performance on lower-spec devices. Such an API is out-of-the-scope of this book—though we've included things like geolocation which aren't part of HTML5, single-vendor APIs are stretching the definition too far.

The rule is: provide both a royalty-free WebM *and* an H.264 video, and both a Vorbis and an MP3 version of your audio, so that nobody gets locked out of your content. Let's not repeat the mistakes of the old "Best viewed in Netscape Navigator" badges on sites, or we'll come round and pin a "n00b" badge to your coat next time you're polishing your FrontPage CD.

Multiple <source> elements

To do this, you need to encode your multimedia twice: once as WebM and once as H.264 in the case of video, and in both Vorbis and MP3 for audio. Then, you tie these separate versions of the file to the media element.

What's the "best" codec?

Asking what's "better" (WebM or MP4) starts an argument that makes debating the merits of Mac or PC seem like a quiet chat between old friends.

To discuss inherent characteristics, you need to argue about macroblock type in B-frames and six-tap filtering for derivation of half-pel luma sample predictions—for all intents and purposes, "My flux capacitor is bigger than yours!"

Suffice it to say that for delivering video across the Web, both WebM and MP4 offer good-enough quality at web-friendly compression. Ogg Theora is less web-friendly.

The real differences are royalty encumbrance and hardware acceleration. Some people need to pay if they have MP4/H.264 video on their website.

There are many chips that perform hardware decoding of H.264, which is why watching movies on your mobile phone doesn't drain the battery in seconds as it would if the video were decoded in software. At the time of this writing (July 2011, a year after WebM was open sourced), hardware-decoding chips for WebM are just hitting the market.

Previously, we've used the `<video src="...">` syntax to specify the source for our video. This works fine for a single file, but how do we tell the browser that there are multiple versions (using different encoding) available? Instead of using the single `src` attribute, you nest separate **`<source>`** elements for each encoding with appropriate `type` attributes inside the `<audio>` or `<video>` element and let the browser download the format that it can display. Faced with multiple `<source>` elements, the browser will look through them (in source order) and choose the first one it finds that it thinks it can play (based on the `type` attribute—which gives explicit information about the container MIME type and the codec used—or, missing that, heuristic based on file extension). Note that in this case we do not provide a `src` attribute in the media element itself:

1. `<video controls>`
2. `<source src=leverage-a-synergy.mp4 type='video/mp4; ¬codecs="avc1.42E01E, mp4a.40.2"'>`
3. `<source src=leverage-a-synergy.webm type='video/webm; ¬codecs="vp8, vorbis"'>`
4. `<p>Your browser doesn't support video.`
5. `Please download the video in <a href=leverage-a- ¬synergy.webm>webM</a> or <a href=leverage-a- ¬synergy.mp4>MP4</a> format.</p>`
6. `</video>`

Line 1 tells the browser that a video is to be inserted and gives it default controls. Line 2 offers an MP4 version of the video. We've put the `mp4` first, because some old versions of Mobile Safari on the iPad have a bug whereby they only look at the first `<source>` element, so that if it isn't first, it won't be played. We're using the `type` attribute to tell the browser what kind of container format is used (by giving the file's MIME type) and what codec was used for the encoding of the video and the audio stream. If you miss out on the `type` attribute, the browser downloads a small bit of each file before it figures out that it is unsupported, which wastes bandwidth and could delay the media playing.

Notice that we used quotation marks around these parameters—the spec uses `'video/mp4; codecs="avc..."'` (single around the outside, double around the codec). Some browsers stumble when it's the other way around. Line 3 offers the WebM equivalent. The codec strings for H.264 and AAC are more complicated than those for WebM because there are several profiles for H.264 and AAC, to cater for different categories of devices and connections. Higher profiles require more CPU to decode, but they are better compressed and take less bandwidth.

We could also offer an Ogg video here for older versions of Firefox and Opera, after the WebM version, so those that can use the higher-quality WebM version pick that up first, and the older (yet still HTML5 `<video>` element capable) browsers fall back to this.

Inside the `<video>` element is our fallback message, including links to *both* formats for browsers that can natively deal with neither video type but which is probably on top of an operating system that can deal with one of the formats, so the user can download the file and watch it in a media player outside the browser.

OK, so that's native HTML5 video for users of modern browsers. What about users of legacy browsers—including Internet Explorer 8 and older?

Video for legacy browsers

Older browsers can't play native video or audio, bless them. But if you're prepared to rely on plugins, you can ensure that users of older browsers can still experience your content in a way that is no worse than they currently get.

NOTE The content between the tags is fallback content only for browsers that do not support the `<video>` element at all. A browser that understands HTML5 video but can't play any of the formats that your code points to will not display the "fallback" content between the tags, but present the user with a broken video control instead. This has bitten me on the bottom a few times. Sadly, there is no video record of that.

Remember that the contents of the `<video>` element can contain markup, like the text and links in the previous example? Here, we'll place an entire Flash video player movie into the fallback content instead (and of course, we'll also provide fallback for those poor users who don't even have that installed). Luckily, we don't need to encode our video in yet another format like FLV (Flash's own legacy video container); because Flash (since version 9) can load MP4 files as external resources, you can simply point your custom Flash video player movie to the MP4 file. This combination should give you a solid workaround for Internet Explorer 8 and older versions of other browsers. You won't be able to do all the crazy video manipulation stuff we'll see later in this chapter, but at least your users will still get to see your video.

The code for this is as hideous as you'd expect for a transitional hack, but it works anywhere that Flash Player is installed—which is almost everywhere. You can see this nifty technique in an article called "Video for Everybody!" by its inventor, Kroc Camen (**http://camendesign.com/code/video_for_everybody**).

Alternatively, you could host the fallback content on a video hosting site and embed a link to that between the tags of a `video` element:

```
<video controls>

   <source src=leverage-a-synergy.mp4 type='video/mp4;
   ¬ codecs="avc1.42E01E, mp4a.40.2"'>
   <source src=leverage-a-synergy.webm type='video/webm;
   ¬ codecs="vp8, vorbis"'>
<embed src="http://www.youtube.com/v/cmtcc94Tv3A&hl=
¬ en_GB&fs=1&rel=0" type="application/x-shockwave-flash"
¬ allowscriptaccess="always" allowfullscreen="true"
¬ width="425" height="344">
</video>
```

You can use the HTML5 Media Library (**http://html5media.info**) to hijack the `<video>` element and automatically add necessary fallback by adding one line of JavaScript in the page header.

Encoding royalty-free video and audio

Ideally, you should start the conversion from the source format itself, rather than recompressing an already compressed version which reduces the quality of the final output. If you already have a web-optimised, tightly compressed MP4/H.264 version, don't convert that one to WebM/VP8, but rather go back to your original footage and recompress that if possible.

For audio, the open-source audio editing software Audacity (**http://audacity.sourceforge.net/**) has built-in support for Ogg Vorbis export.

For video conversion, there are a few good choices. For WebM, there are only a few encoders at the moment, unsurprisingly for such a new codec. See **www.webmproject.org/tools/** for the growing list.

For Windows and Mac users we can highly recommend Miro Video Converter (**www.mirovideoconverter.com**), which allows you to drag a file into its window for conversion into WebM, Theora, or H.264 optimised for different devices such as iPhone, Android Nexus One, PS2, and so on.

The free VLC (**www.videolan.org/vlc/**) can convert files on Windows, Mac, and Linux.

For those developers who are not afraid by a bit of command-line work, the open-source FFmpeg library (**http://ffmpeg.org**) is the big beast of converters. `$ ffmpeg -i video.avi video.webm` is all you need.

The conversion process can also be automated and handled server-side. For instance, in a CMS environment, you may be unable to control the format in which authors upload their files, so you may want to do compression at the server end. ffmpeg can be installed on a server to bring industrial-strength conversions of uploaded files (maybe you're starting your own YouTube killer?).

If you're worried about storage space and you're happy to share your media files (audio and video) under one of the various CC licenses, have a look at the Internet Archive (**www.archive.org/create/**), which will convert and host them for you. Just create a password and upload, and then use a `<video>` element on your page but link to the source file on their servers.

Another option for third-party conversion and hosting is vid.ly. The free service allows you to upload any video up to 2GB via the website, after which they will convert it. When your users come to the site, they will be served a codec their browser understands, even on mobile phones.

Sending differently compressed videos to handheld devices

Video files tend to be large, and sending very high-quality video can be wasteful if sent to handheld devices where the small screen sizes make high quality unnecessary. There's no point in sending high-definition video meant for a widescreen monitor to a handheld device screen, and most users of smartphones and tablets will gladly compromise a little bit on encoding quality if it means that the video will actually load over a mobile

connection. Compressing a video down to a size appropriate for a small screen can save a lot of bandwidth, making your server and—most importantly—your mobile users happy.

HTML5 allows you to use the `media` attribute on the `<source>` element, which queries the browser to find out screen size (or number of colours, aspect ratio, and so on) and to send different files that are optimised for different screen sizes.

> **NOTE** We use `min-device-width` rather than `min-width`. Mobile browsers (which vary the reported width of their viewport to better accommodate web pages by zooming the viewport) will then refer to the nominal width of their physical screen.

This functionality and syntax is borrowed from the CSS Media Queries specification **www.w3.org/TR/css3-mediaqueries** but is part of the markup, as we're switching source files depending on device characteristics. In the following example, the browser is "asked" if it has a min-device-width of 800 px—that is, does it have a wide screen. If it does, it receives `hi-res.webm`; if not, it is sent `lo-res.webm`:

```
<video controls>
      <source src=hi-res.webm ... media="(min-device-width:
      ¬ 800px)">
      <source src=lo-res.webm>
      ...
</video>
```

Also note that you should still use the `type` attribute with `codecs` parameters and fallback content previously discussed. We've just omitted those for clarity.

Rolling custom controls

One truly spiffing aspect of the `<video>` and `<audio>` media elements is that they come with a super easy JavaScript API. The API's events and methods are the same for both `<audio>` and `<video>`. With that in mind, we'll stick with the sexier media element: the `<video>` element for our JavaScript discussion.

As you saw at the start of this chapter, Anne van Kesteren has spoken about the new API and about the new simple methods such as `play()`, `pause()` (there's no stop method: simply pause and move to the start), `load()`, and `canPlayType()`. In fact, that's *all* the methods on the media element. Everything else is events and attributes.

Table 4.3 provides a reference list of media attributes, methods, and events.

TABLE 4.3 Media Attributes, Methods, and Events

ATTRIBUTES	METHODS	EVENTS
error state	load()	loadstart
error	canPlayType(type)	progress
network state	play()	suspend
src	pause()	abort
currentSrc	addTrack(label, kind, language)	error
networkState		emptied
preload		stalled
buffered		play
ready state		pause
readyState		loadedmetadata
seeking		loadeddata
controls		waiting
controls		playing
volume		canplay
muted		canplaythrough
tracks		seeking
tracks		seeked
playback state		timeupdate
currentTime		ended
startTime		ratechange
duration		
paused		
defaultPlaybackRate		
playbackRate		
played		
seekable		
ended		
autoplay		
loop		
width [video only]		
height [video only]		
videoWidth [video only]		
videoHeight [video only]		
poster [video only]		

Using JavaScript and the new media API, you have complete control over your multimedia—at its simplest, this means that you can easily create and manage your own video player controls. In our example, we walk you through some of the ways to control the video element and create a simple set of controls. Our example won't blow your mind—it isn't nearly as sexy as the <video> element itself (and is a little contrived!)—but you'll get a good idea of what's possible through scripting. The best bit is that the UI will be all CSS and HTML. So if you want to style it your own way, it's easy with just a bit of web standards knowledge—no need to edit an external Flash Player or similar.

Our hand-rolled basic video player controls will have a play/pause toggle button and allow the user to scrub along the timeline of the video to skip to a specific section, as shown in **Figure 4.3**.

FIGURE 4.3 Our simple but custom video player controls.

NOTE Some browsers, in particular Opera, will show the native controls even if JavaScript is disabled; other browsers, mileage may vary.

Our starting point will be a video with native controls enabled. We'll then use JavaScript to strip the native controls and add our own, so that if JavaScript is disabled, the user still has a way to control the video as we intended:

```
<video controls>
  <source src="leverage-a-synergy.webm" type="video/webm" />
  <source src="leverage-a-synergy.mp4" type="video/mp4" />
  Your browser doesn't support video.
  Please download the video in <a href="leverage-a-
  ¬ synergy.webm">WebM</a> or <a href="leverage-a-
  ¬ synergy.mp4">MP4</a> format.
</video>
<script>
var video = document.getElementsByTagName('video')[0];
video.removeAttribute('controls');
</script>
```

Play, pause, and toggling playback

Next, we want to be able to play and pause the video from a custom control. We've included a button element that we're going to bind a click handler and do the play/pause functionality from. Throughout my code examples, when I refer to the `play` object it will refer to this button element:

```
<button class="play" title="play">&#x25BA;</button/>
```

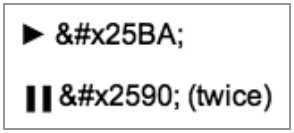

FIGURE 4.4 Using XML entities to represent play and pause buttons.

We're using `&#x25BA;`, which is a geometric XML entity that *looks* like a play button. Once the button is clicked, we'll start the video and switch the value to two pipes using `&#x2590;`, which looks (a little) like a pause, as shown in **Figure 4.4**.

For simplicity, I've included the button element as markup, but as we're progressively enhancing our video controls, all of these additional elements (for play, pause, scrubbing, and so on) should be generated by the JavaScript.

In the play/pause toggle, we have a number of things to do:

1. If the user clicks on the toggle and the video is currently paused, the video should start playing. If the video has previously finished, and our playhead is right at the end of the video, then we also need to reset the current time to 0, that is, move the playhead back to the start of the video, before we start playing it.
2. Change the toggle button's value to show that the next time the user clicks, it will toggle from pause to play or play to pause.
3. Finally, we play (or pause) the video:

```
playButton.addEventListener('click', function () {
  if (video.paused || video.ended) {
    if (video.ended) {
      video.currentTime = 0;
    }
    this.innerHTML = ''; // &#x2590;&#x2590; doesn't
    ¬ need escaping here
    this.title = 'pause';
    video.play();
  } else {
    this.innerHTML = ''; // &#x25BA;
    this.title = 'play';
    video.pause();
  }
}, false);
```

The problem with this logic is that we're relying entirely on our own script to determine the state of the play/pause button. What if the user was able to pause or play the video via the native video element controls somehow (some browsers allow the user to right click and select to play and pause the video)? Also, when the video comes to the end, the play/pause button would still show a pause icon. Ultimately, we need our controls always to relate to the state of the video.

Eventful media elements

The media elements fire a broad range of events: when playback starts, when a video has finished loading, if the volume has changed, and so on. So, getting back to our custom play/pause button, we strip the part of the script that deals with changing its visible label:

```
playButton.addEventListener('click', function () {
  if (video.ended) {
    video.currentTime = 0;
  }
  if (video.paused) {
    video.play();
  } else {
    video.pause();
  }
}, false);
```

In the simplified code, if the video has ended we reset it, and then toggle the playback based on its current state. The label on the control itself is updated by separate (anonymous) functions we've hooked straight into the event handlers on our video element:

```
video.addEventListener('play', function () {
  play.title = 'pause';
  play.innerHTML = '';
}, false);
video.addEventListener('pause', function () {
  play.title = 'play';
  play.innerHTML = '';
}, false);
video.addEventListener('ended', function () {
  this.pause();
}, false);
```

> **NOTE** In these examples, we're using the `addEventListener` DOM level 2 API, rather than the `attachEvent`, which is specific to Internet Explorer up to version 8. IE9 supports video, but it thankfully also supports the standardised `addEventListener`, so our code will work there, too.

Whenever the video is played, paused, or has reached the end, the function associated with the relevant event is now fired, making sure that our control shows the right label.

Now that we're handling playing and pausing, we want to show the user how much of the video has downloaded and therefore how much is playable. This would be the amount of *buffered* video available. We also want to catch the event that says how much video has been played, so we can move our visual slider to the appropriate location to show how far through the video we are, as shown in **Figure 4.5**. Finally, and most importantly, we need to capture the event that says the video is *ready* to be played, that is, there's enough video data to start watching.

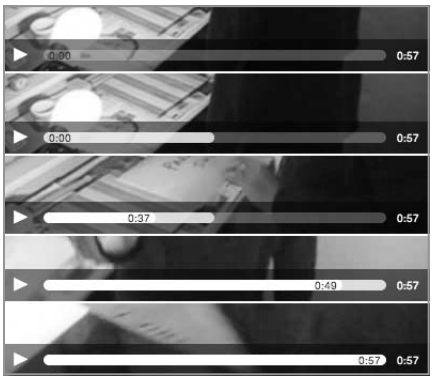

FIGURE 4.5 Our custom video progress bar, including seekable content and the current playhead position.

Monitoring download progress

The media element has a "progress" event, which fires once the media has been fetched but potentially before the media has been processed. When this event fires, we can read the `video.seekable` object, which has a `length`, `start()`, and `end()` method. We can update our seek bar (shown in **Figure 4.5** in the second frame with the whiter colour) using the following code (where the buffer variable is the element that shows how much of the video we can seek and has been downloaded):

```
video.addEventListener('progress', updateSeekable, false);
function updateSeekable() {
  var endVal = this.seekable && this.seekable.length ?
  ¬ this.seekable.end() : 0;
  buffer.style.width = (100 / (this.duration || 1) *
  ¬ endVal) + '%';
}
```

The code binds to the progress event, and when it fires, it gets the percentage of video that can be played back compared to the length of the video. Note the keyword `this` refers to the `video` element, as that's the context in which the `updateSeekable` function will be executed. The `duration` attribute is the length of the media in seconds.

However, there's some issues with Firefox. In previous versions the seekable length didn't match the actual duration, and in the latest version (5.0.1) seekable seems to be missing altogether. So to protect ourselves from the seekable time range going a little awry, we can also listen for the progress event and default to the duration of the video as backup:

```
video.addEventListener('durationchange', updateSeekable,
¬ false);
video.addEventListener('progress', updateSeekable, false);
function updateSeekable() {
  buffer.style.width = (100 / (this.duration || 1) *
    (this.seekable && this.seekable.length ? this.seekable.
    ¬ end() : this.duration)) + '%';
}
```

It's a bit rubbish that we can't reliably get the seekable range. Alternatively we could look to the `video.buffered` property, but sadly since we're only trying to solve a Firefox issue, this value in Firefox (currently) doesn't return *anything* for the `video.buffered.end()` method—so it's not a suitable alternative.

When the media file is ready to play

When your browser first encounters the video (or audio) element on a page, the media file isn't ready to be played just yet. The browser needs to download and then decode the video (or audio) so it can be played. Once that's complete, the media element will fire the `canplay` event. *Typically* this is the time you would initialise your controls and remove any "loading" indicator. So our code to initialise the controls would *typically* look like this:

```
video.addEventListener('canplay', initialiseControls,
¬ false);
```

Nothing terribly exciting there. The control initialisation enables the play/pause toggle button and resets the playhead in the seek bar.

However, sometimes this event won't fire right away (or when you're expecting it to). Sometimes the video suspends download because the browser is trying to prevent overwhelming your system. That can be a headache if you're expecting the `canplay` event, which won't fire unless you give the media element a bit of a kicking. So instead, we've started listening for the `loadeddata` event. This says that there's some data that's been loaded, though not necessarily all the data. This means that the metadata is available (height, width, duration, and so on) and *some* media content—but not *all* of it. By allowing the user to start playing the video at the point in which `loadeddata` has fired, browsers like Firefox are forced to go from a suspended state to downloading the rest of the media content, which lets them play the whole video.

NOTE The events to do with loading fire in the following order: `loadstart`, `durationchange`, `loadedmetadata`, `loadeddata`, `progress`, `canplay`, `canplaythrough`.

You may find that in most situations, if you're doing something like creating a custom media player UI, you might not need the actual video data to be loaded—only the metadata. If that's the case, there's also a `loadedmetadata` event which fires once the first frame, duration, dimensions, and other metadata is loaded. This may in fact be all you need for a custom UI.

So the correct point in the event cycle to enable the user interface is the `loadedmetadata`:

```
video.addEventListener('loadedmetadata', initialiseControls,
¬ false);
```

Media loading control: preload

Media elements also support a `preload` attribute that allows you to control how much of the media is loaded when the page renders. By default, this value is set to `auto`, but you can also set it to `none` or `metadata`. If you set it to `none`, the user will see either the image you've used for the poster attribute, or nothing at all if you don't set a poster. Only when the user tries to play the media will it even request the media file from your server.

By setting the `preload` attribute to `metadata`, the browser will pull down required metadata about the media. It will also fire the `loadedmetadata` event, which is useful if you're listening for this event to set up a custom media player UI.

A race to play video

Here's where I tell you that as much as native video and audio smells of roses, there's a certain pong coming from somewhere. That somewhere is a problem in the implementation of the media element that creates what's known as a "race condition."

A race, what now?

In this situation, the race condition is where an expected sequence of events fires in an unpredicted order. In particular, the events fire *before* your event handler code is attached.

The problem is that it's possible, though not likely, for the browser to load the media element before you've had time to bind the event listeners.

For example, if you're using the `loadedmetadata` event to listen for when a video is ready so that you can build your own fancy-pants video player, it's possible that the native video HTML element may trigger the events *before* your JavaScript has loaded.

Workarounds

There are a few workarounds for this race condition, all of which would be nice to avoid, but I'm afraid it's just something we need to code for defensively.

WORKAROUND #1: HIGH EVENT DELEGATION

In this workaround, we need to attach an event handler on the `window` object. This event handler *must* be above the media element. The obvious downside to this approach is that the script element is above our content, and risks blocking our content from loading (best practice is to include all script blocks at the end of the document).

Nonetheless, the HTML5 specification states that media events should bubble up the DOM all the way to the window object. So when the `loadedmetadata` event fires on the window object, we check where the event originated from, via the `target` property, and if that's our element, we run the setup code. Note that in the example below, I'm only checking the `nodeName` of the element; you may want to run this code against all audio elements or you may want to check more properties on the DOM node to make sure you've got the right one.

```
<script>
function audioloaded() {
  // setup the fancy-pants player
}

window.addEventListener('loadedmetadata', function (event) {
  if (event.target.nodeName === 'AUDIO') {
    // set this context to the DOM node
    audioloaded.call(event.target);
  }
}, true);

</script>

<audio src="hanson.mp3">
  <p>If you can read this, you can't enjoy the soothing
  ¬ sound of the Hansons.</p>
</audio>
```

WORKAROUND #2: HIGH AND INLINE

Here's a similar approach using an inline handler:

```
<script>
function audioloaded() {
  // setup the fancy-pants player
}
</script>

<audio src="hanson.mp3" onloadedmetadata=
¬ "audoloaded.call(this)">
  <p>If you can read this, you can't enjoy the soothing
  ¬ sound of the Hansons.</p>
</audio>
```

Note that in the inline event handler I'm using `.call(this)` to set the `this` keyword to the audio element the event fired upon. This means it's easier to reuse the same function later on if browsers (in years to come) do indeed fix this problem.

By putting the event handler inline, the handler is attached as soon as the DOM element is constructed, therefore it is in place *before* the `loadedmetadata` event fires.

WORKAROUND #3: JAVASCRIPT GENERATED MEDIA

Another workaround is to insert the media using JavaScript. That way you can create the media element, attach the event handlers, and *then* set the source and insert it into the DOM.

Remember: if you do insert the media element using JavaScript, you need to either insert all the different `source` elements manually, or detect the capability of the browser, and insert the `src` attribute that the browser supports, for instance WebM/video for Chrome.

I'm not terribly keen on this solution because it means that those users without JavaScript don't get the multimedia at all. Although a lot of HTML5 is "web applications," my gut (and hopefully yours, too) says there's something fishy about resorting to JavaScript *just* to get the video events working in a way that suits our needs. Even if your gut isn't like mine (quite possible), big boys' Google wouldn't be able to find and index your amazing video of your cat dancing along to Hanson if JavaScript was inserting the video. So let's move right along to workaround number 4, my favourite approach.

WORKAROUND #4: CHECK THE READYSTATE

Probably the best approach, albeit a little messy (compared to a simple video and event handler), is to simply check the `readyState` of the media element. Both audio and video have a `readyState` with the following states:

- `HAVE_NOTHING = 0;`
- `HAVE_METADATA = 1;`
- `HAVE_CURRENT_DATA = 2;`
- `HAVE_FUTURE_DATA = 3;`
- `HAVE_ENOUGH_DATA = 4;`

Therefore if you're looking to bind to the `loadedmetadata` event, you only want to bind if the `readyState` is 0. If you want to bind before it has enough data to play, then bind if `readyState` is less than 4.

Our previous example can be rewritten as:

```
<audio src="hanson.mp3">
  <p>If you can read this, you can't enjoy the soothing
  ¬ sound of the Hansons.</p>
</audio>

<script>
function audioloaded() {
  // setup the fancy-pants player
}

var audio = document.getElementsByTagName('audio')[0];

if (audio.readyState > 0) {
  audioloaded.call(audio);
} else {
  audio.addEventListener('loadedmetadata', audioloaded,
  ¬ false);
}
</script>
```

This way our code can sit nicely at the bottom of our document, and if JavaScript is disabled, the audio is still available. All good in my book.

Will this race condition ever be fixed?

Technically I can understand that this issue has always existed in the browser. Think of an image element: if the load event fires *before* you can attach your load event handler, then nothing is going to happen. You might see this if an image is cached and loads too quickly, or perhaps when you're working in a development environment and the delivery speed is like Superman on crack—the event doesn't fire.

Images don't have ready states, but they do have a `complete` property. When the image is being loaded, complete is false. Once the image is done loading (note this could also result in it *failing* to load due to some error), the complete property is true. So you could, before binding the load event, test the complete property, and if it's true, fire the load event handler manually.

Since this logic has existed for a long time for images, I would expect that this same logic is being applied to the media element, and by that same reasoning, *technically* this isn't a bug, as buggy as it may appear to you and me!

Fast forward, slow motion, and reverse

The spec provides an attribute, **playbackRate**. By default, the assumed playbackRate is 1, meaning normal playback is at the intrinsic speed of the media file. Increasing this attribute speeds up the playback; decreasing it slows it down. Negative values indicate that the video will play in reverse.

Not all browsers support playbackRate yet (only WebKit-based browsers and IE9 support it right now), so if you need to support fast forward and rewind, you can hack around this by programmatically changing currentTime:

```
function speedup(video, direction) {
  if (direction == undefined) direction = 1; // or -1 for
  ¬ reverse

  if (video.playbackRate != undefined) {
    video.playbackRate = direction == 1 ? 2 : -2;
  } else { // do it manually
    video.setAttribute('data-playbackRate', setInterval
    ¬ ((function playbackRate () {
      video.currentTime += direction;

return playbackRate; // allows us to run the
¬ function once and setInterval
    })(), 500));
  }
}

function playnormal(video) {
  if (video.playbackRate != undefined) {
    video.playbackRate = 1;
  } else { // do it manually
    clearInterval(video.getAttribute('data-playbackRate'));
  }
}
```

As you can see from the previous example, if playbackRate is supported, you can set positive and negative numbers to control the direction of playback. In addition to being able to rewind and fast forward using the playbackRate, you can also use a fraction to play the media back in slow motion using video.playbackRate = 0.5, which plays at half the normal rate.

Full-screen video

For some time, the spec prohibited full-screen video, but it's obviously a useful feature so WebKit did its own proprietary thing with `WebkitEnterFullscreen();`. WebKit implemented its API in a way that could only be triggered by the user initiating the action; that is, like pop-up windows, they can't be created unless the user performs an action like a click. The only alternative to this bespoke solution by WebKit would be to stretch the video to the browser window size. Since *some* browsers have a full-screen view, it's possible to watch your favourite video of Bruce doing a Turkish belly dance in full screen, but it would require the user to jump through a number of hoops—something we'd all like to avoid.

In May 2011, WebKit announced it would implement Mozilla's full-screen API (**https://wiki.mozilla.org/Gecko:FullScreenAPI**). This API allows any element to go full-screen (not only `<video>`)—you might want full-screen `<canvas>` games or video widgets embedded in a page via an `<iframe>`. Scripts can also opt in to having alphanumeric keyboard input enabled during full-screen view, which means that you could create your super spiffing platform game using the `<canvas>` API and it could run full-screen with full keyboard support.

As Opera likes this approach, too, we should see something approaching interoperability. Until then, we can continue to fake full-screen by going full-window by setting the video's dimensions to equal the window size.

Multimedia accessibility

We've talked about the keyboard accessibility of the video element, but what about transcripts and captions for multimedia? After all, there is no `alt` attribute for video or audio as there is for `<img>`. The fallback content between the tags is meant only for browsers that can't cope with native video, not for people whose browsers can display the media but can't see or hear it due to disability or situation (for example, being in a noisy environment or needing to conserve bandwidth).

There are two methods of attaching synchronized text alternatives (captions, subtitles, and so on) to multimedia, called *in-band* and *out-of-band*. In-band means that the text file is included in the multimedia container; an MP4 file, for example, is actually a container for H.264 video and AAC audio, and can

hold other metadata files too, such as subtitles. WebM is a container (based on the open standard Matroska Media Container format) that holds VP8 video and Ogg Vorbis audio. Currently, WebM doesn't support subtitles, as Google is waiting for the Working Groups to specify the HTML5 format: "WHATWG/W3C RFC will release guidance on subtitles and other overlays in HTML5 <video> in the near future. WebM intends to follow that guidance". (Of course, even if the container can contain additional metadata, it's still up to the media player or browser to expose that information to the user.)

Out-of-band text alternatives are those that aren't inside the media container but are held in a separate file and associated with the media file with a child <track> element:

```
<video controls>
<source src=movie.webm>
<source src=movie.mp4>
<track src=english.vtt kind=captions srclang=en>
<track src=french.vtt kind=captions srclang=fr>
<p>Fallback content here with links to download video
¬ files</p>
</video>
```

This example associates two caption tracks with the video, one in English and one in French. Browsers will have some UI mechanism to allow the user to select the one she wants (listing any in-band tracks, too).

The <track> element doesn't presuppose any particular format, but the browsers will probably begin by implementing the new WebVTT format (previously known as WebSRT, as it's based on the SRT format) (**www.whatwg.org/specs/web-apps/current-work/multipage/the-video-element.html#webvtt**).

This format is still in development by WHATWG, with lots of feedback from people who really know, such as the BBC, Netflix, and Google (the organisation with probably the most experience of subtitling web-delivered video via YouTube). Because it's still in flux, we won't look in-depth at syntax here, as it will probably be slightly different by the time you read this.

WebVTT is just a UTF-8 encoded text file, which looks like this at its simplest:

```
WEBVTT

00:00:11.000 --> 00:00:13.000
Luftputefartøyet mitt er fullt av ål
```

This puts the subtitle text “Luftputefartøyet mitt er fullt av ål” over the video starting at 11 seconds from the beginning, and removes it when the video reaches the 13 second mark (not 13 seconds later).

No browser currently supports WebVTT or <track> but there are a couple of polyfills available. Julien Villetorte (@delphiki) has written Playr (**www.delphiki.com/html5/playr/**), a lightweight script that adds support for these features to all browsers that support HTML5 video (**Figure 4.6**).

FIGURE 4.6 **Remy reading Shakespeare’s Sonnet 155, with Welsh subtitle displayed by Playr.**

WebVTT also allows for bold, italic, and colour text, vertical text for Asian languages, right-to-left text for languages like Arabic and Hebrew, ruby annotations (see Chapter 2), and positioning text from the default positioning (so it doesn’t obscure key text on the screen, for example), but only if you need these features.

The format is deliberately made to be as simple as possible, and that’s vital for accessibility: *If it’s hard to write, people won’t do it*, and all the APIs in the world won’t help video be accessible if there are no subtitled videos.

Let’s also note that having plain text isn’t just important for people with disabilities. Textual transcripts can be spidered by search engines, pleasing the Search Engine Optimists. And, of course, text can be selected, copied, pasted, resized, and styled with CSS, translated by websites, mashed up, and all other kinds of wonders. As Shakespeare said in Sonnet 155, “If thy text be selectable/‘tis most delectable.”

NOTE Scott Wilson’s VTT Caption Creator (**http://scottbw.wordpress.com/2011/06/28/creating-subtitles-and-audio-descriptions-with-html5-video/**) is a utility that can help author subtitles to be used as standalone HTML, or a W3C Widget.

Synchronising media tracks

HTML5 will allow for alternative media tracks to be included and synchronised in a single `<audio>` or `<video>` element .

You might, for example, have several videos of a sporting event, each from different camera angles, and if the user moves to a different point in one video (or changes the playback rate for slow motion), she expects all the other videos to play in sync. Therefore, different media files need to be grouped together.

This could be a boon for accessibility, allowing for sign-language tracks, audio description tracks, dubbed audio tracks, and similar additional or alternative tracks to the main audio/video tracks.

MediaElement.js, King of the Polyfills

MediaElement.js (**www.mediaelementjs.com**) is a plugin developed by John Dyer (**http://j.hn**), a web developer for Dallas Theological Seminary.

Making an HTML5 player isn't rocket surgery. The problem comes when you're doing real world video and you need to support older browsers that don't support native multimedia or browsers that don't have the codec you've been given.

Most HTML5 players get around this by injecting a completely separate Flash Player. But there are two problems with this approach. First, you end up with two completely different playback UIs (one in HTML5 and one in Flash) that have to be skinned and styled independently. Secondly, you can't use HTML5 Media events like "ended" or "timeupdate" to sync other elements on your page.

MediaElement.js takes a different approach. Instead of offering a bare bones Flash player as a fallback, it includes a custom player that mimics the entire HTML5 Media API. Flash (or Silverlight, depending on what the user has installed) renders the media and then bubbles fake HTML5 events up to the browser. This means that with MediaElement.js, even our old chum IE6 will function as if it supports <video> and <audio>. John cheekily refers to this as a fall "forward" rather than a fallback.

On mobile systems (Android, iOS, WP7), MediaElement.js just uses the operating system's UI. On the desktop, it supports all modern browsers with true HTML5 support and upgrades older browsers. Additionally, it injects support using plugins for unsupported codecs support. This allows it to play MP4, Ogg, and WebM, as well as WMV and FLV and MP3.

MediaElement.js also supports multilingual subtitles and chapter navigation through <track> elements using WebVTT, and there are plugins for Wordpress, Drupal, and BlogEngine.net, making them a no-brainer to deploy and use on those platforms.

A noble runner-up to the crown is LeanBack Player **http://dev.mennerich.name/showroom/html5_video/** with WebVTT polyfilling, no dependency on external libraries, and excellent keyboard support.

NOTE On 25 August 2011, the American Federal Communications Commission released FCC 11-126, ordering certain TV and video networks to provide video description for certain television programming.

Providing descriptions of a program's key visual elements in natural pauses in the program's dialogue is a perfect use of `mediagroup` and the associated API.

This can be accomplished with JavaScript, or declaratively with a **`mediagroup`** attribute on the `<audio>` or `<video>` element:

```
<div>
  <video src="movie.webm" autoplay controls
  ¬ mediagroup=movie></video>
  <video src="signing.webm" autoplay
  ¬ mediagroup=movie></video>
 </div>
```

This is very exciting, and very new, so we won't look further: the spec is constantly changing and there are no implementations.

Video conferencing, augmented reality

As we mentioned earlier, accessing a device's camera and microphone was once available only to web pages via plugins. HTML5 gives us a way to access these devices straight from JavaScript, using an API called `getUserMedia`. (You might find it referred to as the <device> element on older resources. The element itself has been spec'd away, but the concept has been moved to a pure API.)

An experimental build of Opera Mobile on Android gives us a glimpse of what will be possible once this feature is widely available. It connects the camera to a `<video>` element using JavaScript by detecting whether `getUserMedia` is supported and, if so, setting the stream coming from the camera as the `src` of the `<video>` element:

NOTE `getUserMedia` is a method of the navigator object according to the spec. Until the spec settles down, though, Opera (the only implementors so far) are putting it on the `opera` object.

```
<!DOCTYPE html>
<h1>Simple web camera display demo</h1>
<video autoplay></video>
<script type="text/javascript">
var video = document.getElementsByTagName('video')[0],
        heading = document.getElementsByTagName('h1')[0];

if(navigator.getUserMedia) {
  navigator.getUserMedia('video', successCallback,
  ¬ errorCallback);
  function successCallback( stream ) {
    video.src = stream;
  }
  function errorCallback( error ) {
    heading.textContent =
         "An error occurred: [CODE " + error.code + "]";
  }
```

```
} else {
  heading.textContent =
       "Native web camera streaming is not supported in
       ¬ this browser!";
}
</script>
```

Once you've done that, you can manipulate the video as you please. Rich Tibbett wrote a demo that copies the video into canvas (thereby giving you access to the pixel data), looks at those pixels to perform facial recognition, and draws a moustache on the face, all in JavaScript (see **Figure 4.7**).

FIGURE 4.7 Remy Sharp, with a magical HTML5 moustache. (Photo by Julia Gosling)

Norwegian developer Trygve Lie has made demos of `getUserMedia` that use Web Sockets (see Chapter 10) to send images from an Android phone running the experimental Opera Mobile build to a desktop computer. See **https://github.com/trygve-lie/demos-html5-realtime** for the source code and a video demonstrating it.

Obviously, giving websites access to your webcam could create significant privacy problems, so users will have to opt-in, much as they have to do with geolocation. But that's a UI concern rather than a technical problem.

Taking the concept even further, there is also a Peer-to-Peer API being developed for HTML, which will allow you to hook up your camera and microphone to the `<video>` and `<audio>` elements of someone else's browser, making it possible to do video conferencing.

In May 2011, Google announced WebRTC, an open technology for voice and video on the Web, based on the HTML5 specifications. WebRTC uses VP8 (the video codec in WebM) and two audio codecs optimised for speech with noise and echo cancellation, called iLBC, a narrowband voice codec, and iSAC, a bandwidth-adaptive wideband codec (see **http://sites.google.com/site/webrtc/**).

As the project website says, "We expect to see WebRTC support in Firefox, Opera, and Chrome soon!"

Summary

You've seen how HTML5 gives you the first credible alternative to third-party plugins. The incompatible codec support currently makes it harder than using plugins to simply embed video in a page and have it work cross-browser.

On the plus side, because video and audio are now regular elements natively supported by the browser (rather than a "black box" plugin) and offer a powerful API, they're extremely easy to control via JavaScript. With nothing more than a bit of web standards knowledge, you can easily build your own custom controls, or do all sorts of crazy video manipulation with only a few lines of code. As a safety net for browsers that can't cope, we recommend that you also add links to download your video files outside the `<video>` element.

There are already a number of ready-made scripts available that allow you to easily leverage the HTML5 synergies in your own pages, without having to do all the coding yourself. jPlayer (**www.jplayer.org**) is a very liberally licensed jQuery audio player that degrades to Flash in legacy browsers, can be styled with CSS, and can be extended to allow playlists. For video, you've already met Playr, MediaElement.js and LeanBack Player which are my weapons of choice, but many other players exist. There's a useful video player comparison chart at **http://praegnanz.de/html5video/**.

Accessing video with JavaScript is more than writing new players. In the next chapter, you'll learn how to manipulate native media elements for some truly amazing effects, or at least our heads bouncing around the screen—and who could conceive of anything more amazing than that?

INDEX

E

F

G

H

I

J

K

L

M

P

Q

R

S

T

U

V

W

X - Z