INTRODUCING

HTML

BRUCE LAWSON
REMY SHARP

Introducing HTML5

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 © 2011 by Remy Sharp and Bruce Lawson

Project Editor: Michael J. Nolan
Development Editor: Jeff Riley/Box Twelve Communications
Technical Editors: Patrick H. Lauke (www.splintered.co.uk),
Robert Nyman (www.robertnyman.com)
Production Editor: Cory Borman
Copyeditor: Doug Adrianson
Proofreader: Darren Meiss
Compositor: Danielle Foster
Indexer: Joy Dean Lee
Back cover author photo: Patrick H. Lauke

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: 978-0-321-68729-6
ISBN 10: 0-321-68729-9

9 8 7 6 5 4 3 2 1

Printed and bound in the United States of America

CONTENTS

Introduction ix

CHAPTER 1 **Main structure** 1

The <head> 2

Using new HTML5 structural elements 6

Styling HTML5 with CSS 10

When to use the new HTML5 structural elements 13

Summary 21

CHAPTER 2 **Text** 23

Structuring main content areas 24

Adding blogposts and comments 29

Working with HTML5 outlines 30

Understanding WAI-ARIA 48

Even more new structures! 51

Redefined elements 56

Global attributes 61

Features not covered in this book 64

Summary 66

CHAPTER 3 **Forms** 67

We ❤ HTML, and now it ❤s us back 68

New input types 68

New attributes 74
Putting all this together 79
Backwards compatibility with legacy browsers . . . 82
Styling new form fields and error messages 83
Overriding browser defaults 84
Using JavaScript for DIY validation 85
Avoiding validation 86
Summary 89

CHAPTER 4 **Video and Audio** **91**

Native multimedia: why, what, and how? 92
Codecs—the horror, the horror 98
Rolling custom controls 102
Multimedia accessibility 110
Summary 113

CHAPTER 5 **Canvas** **115**

Canvas basics .118
Drawing paths .122
Using transformers: pixels in disguise 124
Capturing images 126
Pushing pixels .130
Animating your canvas paintings 134
Summary 140

CHAPTER 6 **Data Storage** **141**

Storage options 142
Web Storage. .143

Web SQL Databases 152
Summary 162

CHAPTER 7 **Offline** 163
Pulling the plug: going offline 164
The cache manifest 164
How to serve the manifest 168
The browser-server process 168
applicationCache 171
Using the manifest to detect connectivity 172
Killing the cache 174
Summary 174

CHAPTER 8 **Drag and Drop** 175
Getting into drag 176
Interoperability of dragged data 180
How to drag *any* element 182
Adding custom drag icons 183
Accessibility 184
Summary 186

CHAPTER 9 **Geolocation** 187
Sticking a pin in your visitor 188
API methods 190
How it works under the hood: it's magic 195
Summary 196

CHAPTER 10 Messages, Workers, and Sockets 197

Chit chat with the Messaging API 198

Threading using Web Workers 200

Web Sockets: working with streaming data 212

Summary 216

And finally... 216

Index 217

INTRODUCTION

Welcome to the Remy and Bruce show. 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 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.

Who's this book for?

No knowledge of HTML5 is assumed, but we expect 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 book 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 how to use each element or API or how they work in the context of each other. 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 specification that needs 900 pages to print (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 6 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 we include simply for the reason that it's really cool, we're excited about it, and it's part of the New Wave of Exciting Technologies for Making Web Apps.

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.

Nonetheless, 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 pre-empt (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 HTML was frozen at version 4.01 and a specification was released called XHTML, which was an XML version of HTML requiring XML syntax rules like 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). There was XHTML Transitional, which 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, 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 2004–2009 Apple Computer, Inc., Mozilla Foundation, and Opera Software ASA" (note that you are licensed to use, reproduce, and create derivative works).

Hickson moved from Opera 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 over-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 (http://dig.csail.mit.edu/breadcrumbs/node/166).

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 latterly Ruby, Paul Cotton of Microsoft and Maciej Stachowiak of Apple), and the WHATWG, under the continued editorship of Hickson.

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).

In search of the Spec

Because the HTML5 specification is being developed by both the W3C and WHATWG, there are different versions of the spec.

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

For the WHATWG version, go to http://whatwg.org/html5 but beware: this is titled "HTML5 (including next generation additions still in development)" and there are hugely experimental ideas in there such as the <device> element. 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 compete 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.

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 and lives at http://www.w3.org/TR/geolocation-API/

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. Good ideas were adopted from Twitter, blogs, 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" (**http://www.webstandards.org/2009/05/13/interview-with-ian-hickson-editor-of-the-html-5-specification/**). 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 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 exiting 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 need be. A consistent DOM is so important to the design of HTML5 that the language itself is defined in terms of the DOM.

NOTE There is an HTML5 spec that deals with just the aspects relevant to web authors, generated automatically from the main source available at **http://dev.w3.org/html5/markup/**.

In the interests 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 900 pages long if printed out, but only 300 or so of those are of relevance to web authors (that's you and us); the rest of it is for implementors 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 that traditional static text-images-links documents that make up the majority of the Web. Examples are online word processors, photo editing tools, mapping sites, etc. 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 what your thoughts are 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. (No browser yet ships with an HTML5 parser by default, although at time of writing Firefox has an *experimental* HTML5 parser that can be switched on from about:config by changing the preference html5.enable to true.)

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 Internet Explorer 8 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, and even though the spec went to first final draft in October 2009, browsers were already implementing HTML5 support (particularly around the APIs) before this date. Equally, HTML5 support is going to continuously increase as the browsers start rolling out the features.

This book has been written between November 2009 and May 2010. We've already amended chapters several times to take into account changes in the specification, which is looking (dare we say it?) pretty stable now. (We will regret writing that, we know!)

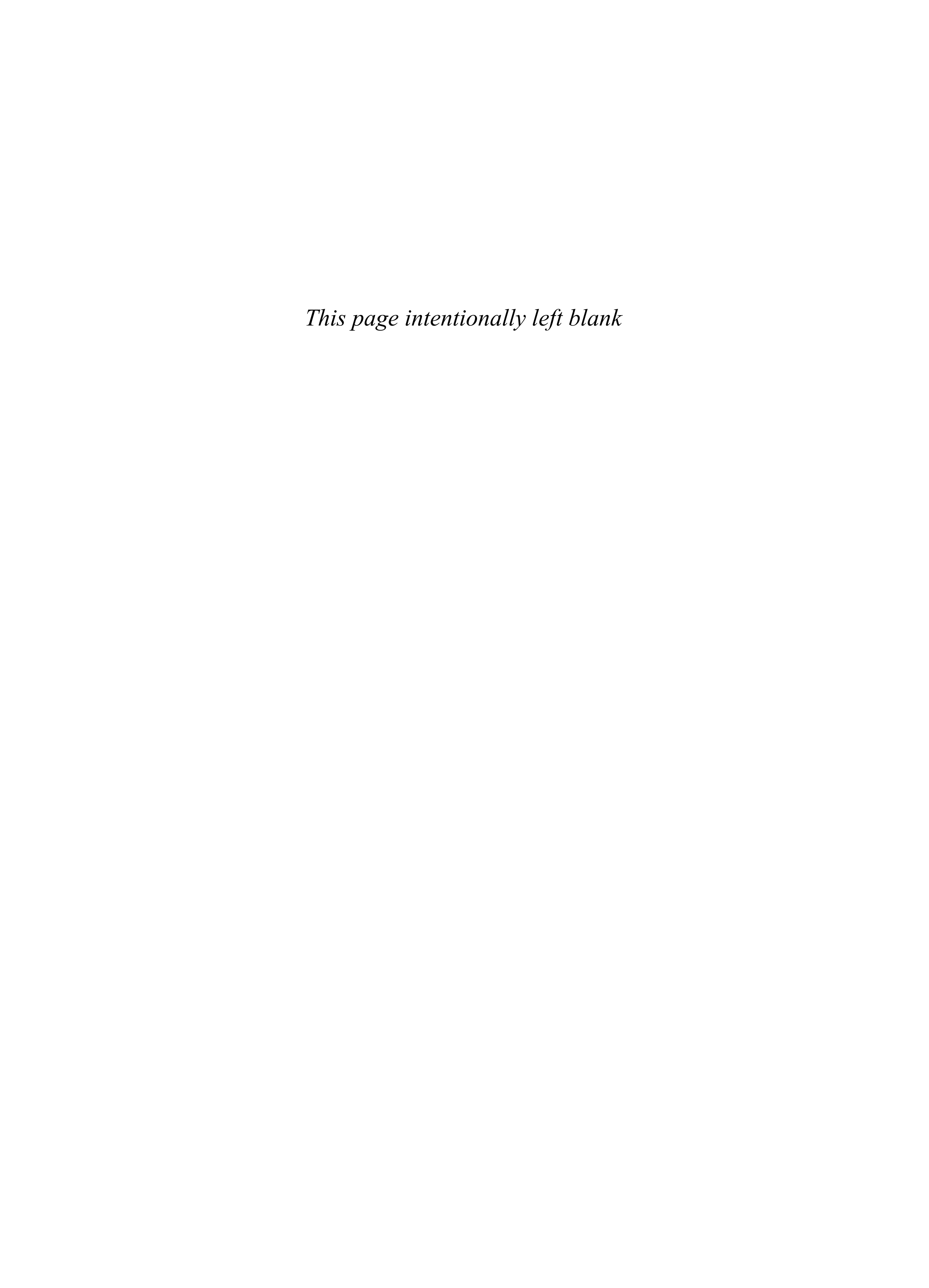
Of course, instances where we say "this is only supported in browser X" will rapidly date—which is a good thing.

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 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.



This page intentionally left blank

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 the ubiquity of its 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 for both types of media.

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, and during the writing of this book Microsoft announced forthcoming support in 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 only note 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 than that 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, or the knowledge of how to); 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 in 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 is happening 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 dropdown 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. If the dimensions of the plugin's drawing area are resized, this can sometimes have unforeseen effects—a movie playing in the plugin may not resize, but instead simply 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 non-validating `<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 somewhat black and white headlines of the tech journalists, HTML5 won't "kill" all plugins overnight. There are use-cases for plugins not covered by the new 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 DRM are probably not going to want to use HTML5 video or audio, as they will 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 to save the video. (Of course, you don't need us to point out that DRM is a fools' errand, anyway. All you do is alienate your honest users while causing minor inconvenience to dedicated pirates.)

There is a highly nascent `<device>` element rudimentarily specified for "post-5" HTML, but there is no support in browsers for it. Plugins remain the best option for a browser to transmit video and audio from the user's machine to a web page such as Daily Mugshot or Chat Roulette. After shuddering at the unimaginable loneliness that a world without Chat Roulette would represent, consider also the massive amount of content out there that will require plugins to render it for a long time to come.

Anatomy of the video element

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

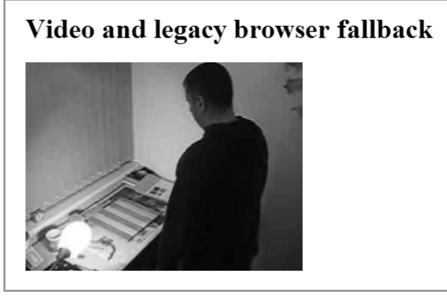
```
<video src=turkish.ogv></video>
```

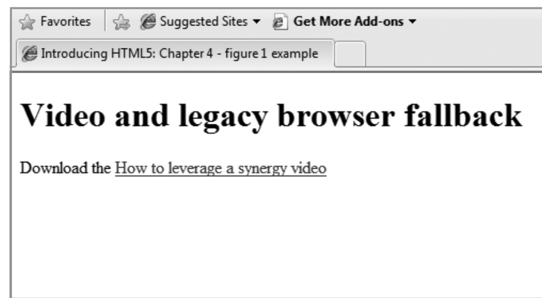
The .ogv file extension is used here to point to an Ogg Theora video.

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

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.ogv>
  Download the <a href=leverage-a-synergy.ogv>How to
  ¬ leverage a synergy video</a>
</video>
```





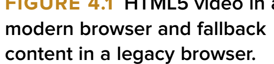


FIGURE 4.1 HTML5 video in a modern browser and fallback content in a legacy browser.

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

You can tell the browser to play the video or audio automatically, but 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.ogv autoplay>
</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.ogv controls>
</video>
```

Naturally, these differ between browsers, in the same way that form controls do, for example, but you'll find nothing too surprising. There's a play/ pause toggle, a seek bar, and volume control.

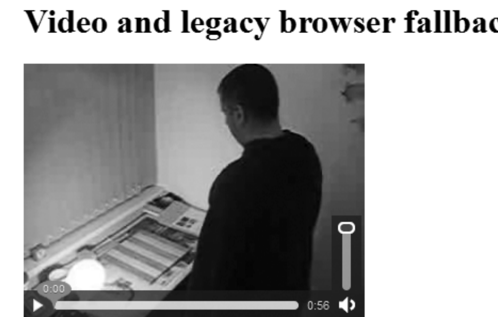


FIGURE 4.2 The default controls in Firefox 3.6 (similar in all modern browsers).

NOTE Browsers have different levels of keyboard accessibility. Firefox's native controls don't appear when JavaScript is disabled (the contextual menu allows the user to stop and start the movie, but there is the issue of discoverability, and it doesn't seem possible to choose these options without JS.) Opera's accessible native controls are always present when JavaScript is disabled, regardless of whether the `controls` attribute is specified.

Chrome and Safari have issues with keyboard accessibility. We anticipate increased keyboard accessibility as manufacturers iron out teething problems.

Notice that these controls appear when a user hovers over a video or when she tabs to the video. It's also possible to tab through the different controls. 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, nothing is rendered visually on the page at all, but is, of course, there in the DOM and fully controllable via JavaScript and the new APIs.

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.

height, width

These attributes tell the browser the size in pixels of the video. (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 is the intrinsic width of the poster frame, if that is available. Otherwise it is 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 "letter-boxed" 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.

preload

Maybe you're pretty sure that the user wants to activate the media (he'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 states of 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.

1. `preload=auto` (or just `preload`)

A suggestion to the browser that it should begin downloading the entire file. Note that we say "suggestion." The browser may ignore this—perhaps because it detected very slow connection or a setting in a mobile browser "Never preload media" to save the user's bandwidth.

2. `preload=none`

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

3. `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.

NOTE The specification for `preload` changed in March 2010 and is not implemented anywhere as of April 2010.

src

As on an `<img>`, this attribute points to the file to be displayed. 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. Using a single source file with the `src` attribute is only really useful for rapid prototyping or for intranet sites where you know the user's browser and which codecs it supports.

Codecs—the horror, the horror

Early drafts of the HTML5 specification mandated that all browsers should at least have built-in support for multimedia in 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, it's often used with Theora for video and combined together in the Ogg container format.

However, these codecs 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. Opera and Firefox support Theora and Vorbis. Safari doesn't, preferring instead to provide native support for the H.264 video codec and MP3 audio. Microsoft has announced that IE9 will also support H.264, which is also supported on iPhone and Android. Google Chrome supports Theora and H.264 video, and Vorbis and MP3 audio. Confused?

As we were finishing this book, Google announced it is open-sourcing a video codec called VP8. This is a very high-quality codec, and when combined with Vorbis in a container format based on the Matroska format, it's collectively known as "webM".

Opera, Firefox and Chrome have announced it will support it. IE9 will, if the codec is separately installed. VP8 will be included in Adobe's Flash Player and every YouTube video will be in webM format.

Like Theora, it's a royalty-free codec. In this chapter, you can substitute .ogv examples with .webm for high quality video, once browser support is there.

The rule is: provide both royalty-free (webM or Theora) *and* H.264 video in your pages, and both Vorbis and MP3 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 websites.

Multiple <source> elements

To do this, you need to encode your multimedia twice: once as Theora 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. 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.

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.ogv type='video/ogg;
        ¬ codecs="theora, vorbis"'>
3       <source src=leverage-a-synergy.mp4 type='video/mp4;
        ¬ codecs="avc1.42E01E, mp4a.40.2"'>
4   <p>Your browser doesn't support video.
5   Please download the video in <a href=leverage-a-
    ¬ synergy.ogv>Ogg</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 to give it default controls. Line 2 offers an Ogg Theora video and uses 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. We could also offer a WebM video here as a high-quality royalty-free option. Notice that we used quotation marks around these parameters. 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.

Line 3 offers an H.264 video. The codec strings for H.264 and AAC are more complicated than those for Ogg because there are several profiles for H.264 and AAC. Higher profiles require more CPU to decode, but they are better compressed and take less bandwidth.

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 all 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. 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? Because the MP4 file type can also be played by the Flash player plugin, you can use the MP4 movie in combination as a fallback for Internet Explorer 8 and older versions of other browsers.

The code for this is as hideous as you'd expect for a transitional hack, but it works everywhere a 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.ogv type='video/ogg;
   ¬ codecs="theora, vorbis"'>
   <source src=leverage-a-synergy.mp4 type='video/mp4;
   ¬ codecs="avc1.42E01E, mp4a.40.2"'>
<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 html5media library http://static.etianen.com/html5media/ to hijack the `<video>` element and automagically add necessary fallback by adding one line of JavaScript in the head of your page.

Encoding royalty-free video and audio

Ideally, you should start the conversion from the source format itself, rather than recompressing an already compressed version. Double compression can seriously reduce the quality of the final output.

On the audio side of things, 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.

The free application evom (http://thelittleappfactory.com/evom/) can make Ogg Theora on a Mac through a nice graphical interface. Windows and Mac users can download Miro Video Converter (www.mirovideoconverter.com/), which allows you to drag a file into its window for conversion into 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 to Ogg on Windows or Linux. OggConvert (http://oggconvert.tristanb.net/) is a useful utility for Linux users.

Alternatively, the Firefox extension Firefogg and its associated website http://firefogg.org/ provides an easy web-based conversion. TinyOgg (http://tinyogg.com/) converts Flash video to Ogg for download, and can even be fed a YouTube URL.

The conversion process can also be automated and handled server-side. For instance in a CMS environment, you may not be able to control the format in which authors upload their files, so you may want to do compression at the server end. The open-source ffmpeg library (http://ffmpeg.org/) 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, then use a `<video>` element on your page but link to the source file on their servers.

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. 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 send different files that are optimised for different screen sizes.

NOTE We use `min-device-width` rather than `min-width` to cater to devices that have a viewport into the content—that is, every full-featured smartphone browser, as this gives us the width of the viewport display.

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

```
<video controls>
      <source src=hi-res.ogv ... media="(min-device-width:
      ¬ 800px)">
      <source src=lo-res.ogv>
</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 media element, and therefore the audio and video elements, is that the JavaScript API is super easy. The API for both audio and video descend from the same media API, so they're nearly exactly the same. The only difference in these elements is that the video element has `height` and `width` attributes and a `poster` attribute. The events, the methods, and all other attributes are the same. 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 talks about the new API and that we have new simple methods such as `play()`, `pause()` (there's no stop method: simply pause and 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 4.1 provides a reference list of media attributes and events.

TABLE 4.1 Media Attributes and Events

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

Using JavaScript and the new media API you can 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.

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.ogv" type="video/ogg" />
  <source src="leverage-a-synergy.ogv" type="video/mp4" />
  Your browser doesn't support video.
  Please download the video in <a href="leverage-a-
  ¬ synergy.ogv">Ogg</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` variable it will refer to the button element:

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

▶ ►

▐▐ ▐ (twice)

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

We're using `BA;`, 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 `▐`, 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 video is currently paused, start playing, or if the video has finished then we need to reset the current time to 0, that is, move the playhead back to the start of the video.
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:

```
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();
}
```

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 to always 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:

```
if (video.ended) {
  video.currentTime = 0;
}
if (video.paused) {
  video.play();
} else {
  video.pause();
}
// which could be written as: video[video.paused ? 'play' :
¬ 'pause']();
```

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. The upcoming IE9 will support video, but it thankfully also supports the standardised `addEventListener`, so our code will work there, too.

In the simplified code if the video has ended, we reset it, 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);
```

Now whenever the video is played, paused, or has reached the end, the function associated with the relevant event is 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 that the keyword `this` refers to the

video element, as that's the context in which the `updateSeekable` function will be executed, and the duration attribute is the length of the media in seconds

However, there's sometimes a subtle issue in Firefox in its video element that causes the `video.seekable.end()` value *not* to be the same as the duration. Or rather, once the media is fully downloaded and processed, the final duration doesn't match the `video.seekable.end()` value. To work around this issue, we can also listen for the `durationchange` event using the same `updateSeekable` function. This way, if the duration does change *after* the last process event, the `durationchange` event fires and our buffer element will have the correct width:

```
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() : 0)) + '%';
}
```

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.

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

However, sometimes this event won't fire right away (or at least when you're expecting it to fire). Sometimes the video suspends download because the browser is trying to save downloading too much for you. 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 particularly 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 to play the video at the point in which `loadeddata` has fired, it forces browsers like Firefox to go from a suspended state to downloading the rest of the media content, allowing it to play the whole video. So, in fact, the correct point in the event cycle to enable the user interface is the `loadeddata`:

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

Preloading metadata

A recent addition to the media element is the `preload` attribute (so new that it's not supported in browsers right now). It allows developers to tell browsers only to download the header information about the media element, which would include the metadata. If support for this attribute does make its way into browsers, it stands to reason we should listen for the `loadedmetadata` event over the `loadeddata` event if you wanted to initalise the duration and slider controls of the media.

Fast forward, slow motion, and reverse

The spec provides an attribute, `playbackRate`. By default the assumed `playbackRate` is 1, meaning normal playback 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 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 () {
      video.currentTime += direction;
      return arguments.callee; // allows us to run 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.

Multimedia accessibility

We've talked about the keyboard accessibility of the video element, but what about transcripts, 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 only meant 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, in a noisy environment or needing to conserve bandwidth).

The theory of HTML5 multimedia accessibility is excellent. The original author should make a subtitle file and put it in the container Ogg or MP4 file along with the multimedia files, and the browser will offer a user interface whereby the user can get those captions or subtitles. Even if the video is "embedded" on 1,000 different sites (simply by using an external URL as the source of the video/audio element), those sites get the subtitling

information for free, so we get "write once, read everywhere" accessibility.

That's the theory. In practice, no one knows how to do this; the spec is silent, browsers do nothing. That's starting to change; at the time of this writing (May 2010), the WHATWG have added a new `<track>` element to the spec, which allows addition of various kinds of information such as subtitles, captions, description, chapter titles, and metadata.

The WHATWG is specifying a new timed text format called WebSRT (www.whatwg.org/specs/web-apps/current-work/multipage/video.html#websrt) for this information, which is one reason that this shadowy 29th element isn't in the W3C version of the spec. The format of the `<track>` element is

```
<track kind=captions src=captions.srt>
```

But what can you do right now? There is no one true approach to this problem, but here we'll present one possible (albeit hacky) interim solution.

Bruce made a proof of concept that displays individual lines of a transcript, which have been timestamped using the new HTML5 `data-*` attributes:

```
<article class=transcript lang=en>
<p><span data-begin=3 data-end=5>Hello, good evening and
¬ welcome.</span>
<span data-begin=7.35 data-end=9.25>Let's welcome Mr Last
¬ Week, singing his poptabulous hit &ldquot;If I could turn
¬ back time!&rdquot;</span>
</p>
...
</article>
```

JavaScript is used to hide the transcript `<article>`, hook into the `timeupdate` event of the video API, and overlay spans as plain text (therefore stylable with CSS) over (or next to) the video element, depending on the current playback time of the video and the timestamps on the individual spans. See it in action at http://dev.opera.com/articles/view/accessible-html5-video-with-javascripted-captions/. See **Figure 4.6**.

FIGURE 4.6 The script superimposes the caption over the video as delectable selectable text.

The data-* attributes (custom data attributes)

HTML5 allows custom attributes on any element. These can be used to pass information to local scripts.

Previously, to store custom data in your markup, authors would do something annoying like use classes: `<input class="spaceship shields-5 lives-3 energy-75">`. Then your script would need to waste time grabbing these class names, such as `shields-5`, splitting them at a delimiter (a hyphen in this example) to extract the value. In his book, *PPK on JavaScript* (New Riders, ISBN 0321423305), Peter Paul Koch explains how to do this and why he elected to use custom attributes in some HTML4 pages, making the JavaScript leaner and easier to write but also making the page technically invalid. As it's much easier to use `data-shields=5` for passing name/value pairs to scripts, HTML5 legitimises and standardises this useful, real-world practice.

We're using `data-begin` and `data-end`; they could just as legitimately be `data-start` and `data-finish`, or (in a different genre of video) `data-ooh-matron` and `data-slapandtickle`. Like choosing class or id names, you should pick a name that matches the semantics.

Custom data attributes are only meant for passing information to the site's own scripts, for which there are no more appropriate attributes or elements.

The spec says "These attributes are not intended for use by software that is independent of the site that uses the attributes" and are therefore not intended to pass information to crawlers or third-party parsers. That's a job for microformats, microdata, or RDFa.

When the `data-*` attributes are fully supported in a browser, JavaScript can access the properties using `element.dataset.foo` (where the `data-foo` attribute contains the value). Support can be emulated using JavaScript by extending the `HTMLElement` object, which typically isn't possible in IE9 alpha release and below, which you can see here: **http://gist.github.com/362081**. Otherwise scripts can access the values via the `get`/`setAttribute` methods. The advantage of the dataset property over `setAttribute` is that it can be enumerated, but also, when fully implemented in browsers, setting a dataset attribute automatically sets the content attribute on the element giving you a shorthand syntax for setting custom data.

For more information, see the spec **http://dev.w3.org/html5/spec/Overview.html#custom-data-attribute**.

The BBC has a similar experiment at http://open.bbc.co.uk/rad/demos/html5/rdtv/episode2/ that takes in subtitles from an external JavaScript file http://open.bbc.co.uk/rad/demos/html5/rdtv/episode2/rdtv-episode2.js, which is closer to the vision of HTML5, but it doesn't have the side effect of allowing search engines to index the contents of the transcript.

Silvia Pfeiffer, a contractor for Mozilla, has some clever demos using HTML5 videos and some extra extensions (that are not part of the spec) at www.annodex.net/~silvia/itext/.

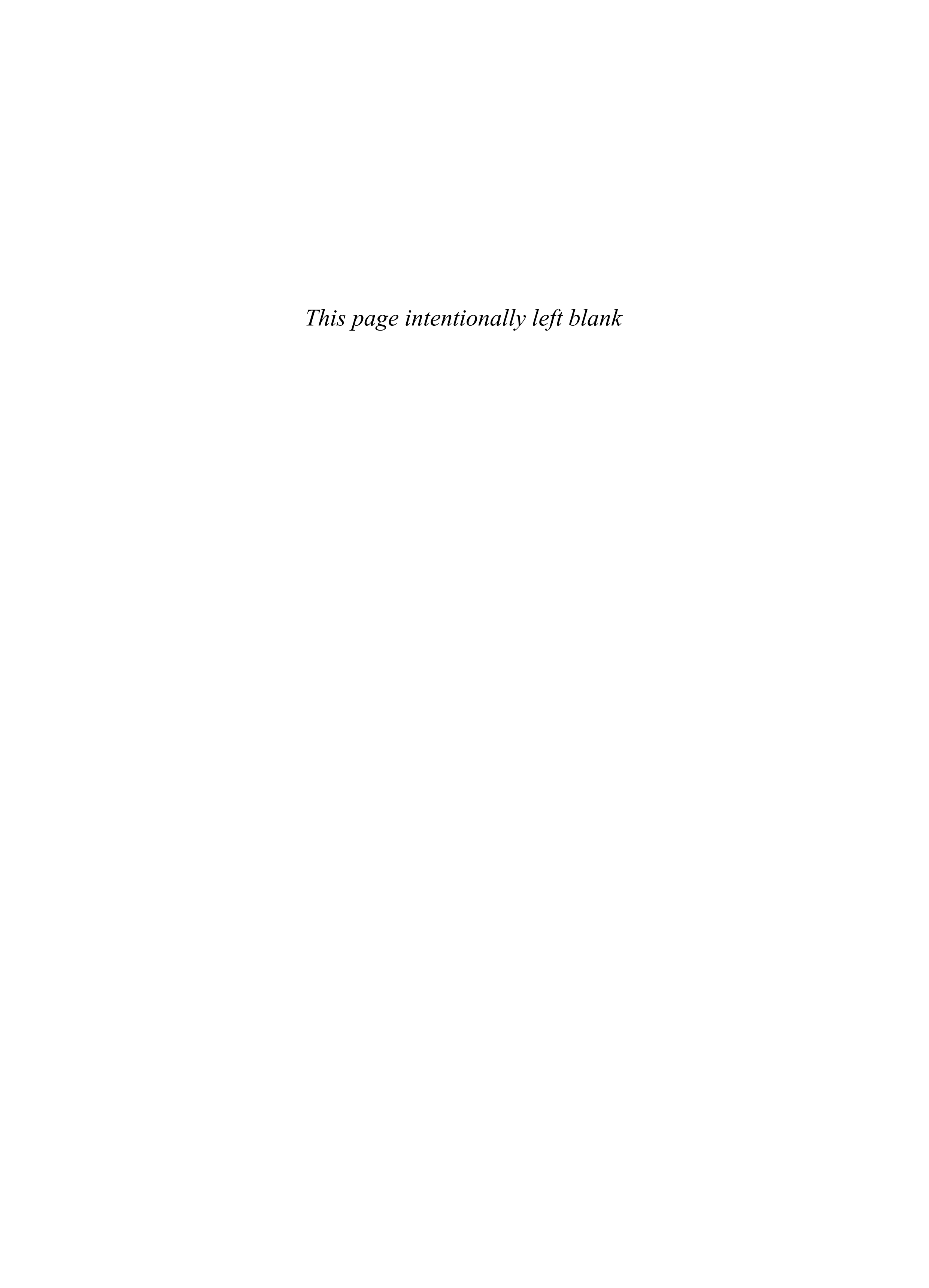
Summary

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, developers can easily build their 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. The Kaltura player (http://www.html5video.org/) is an open source video player that works in all browsers. jPlayer (http://www.happyworm.com/jquery/jplayer/) 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.

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 amazinger than that?


This page intentionally left blank

INDEX

A

<a> element, 54
accessibility. *See also* WAI-ARIA
  canvas element, 139
  dragging and dropping, 184–185
  multimedia, 110–113
  outlining algorithm, 36–37
Accessible Rich Internet Applications. *See* WAI-ARIA
addEventListener method, 106–110, 199, 208
<address> element, 58
animating paintings, 134–137
APIs, retained-mode *versus* immediate mode, 124
<applet> element, 60
ARIA (Accessible Rich Internet Applications). *See* WAI-ARIA
aria-* attribute, 63
aria-grabbed attribute, 185
aria-required attribute, 76
aria-valuenow attribute, 81–82
<article> element, 20–21, 37–42, 52, 54, 58, 111
  block-level links, 38
  comments as nested articles, 29–30
Asian languages, 55
<aside> element, 17, 19–20, 33, 52, 54
attributes attribute, 63
Audacity software, 101
<audio> element, 54, 94, 96, 99–100
autocomplete attribute, 74, 78
autofocus attribute, 75
autoplay attribute, 95

B

<b> element, 59
Baranovskiy, Dmitry, 124
base64 encoded assets, 133
beginPath method, 122–123
<big> element, 60
object> element, 92–93
<blink> element, 60
block-level elements, 38, 54
<blockquote> element, 28, 34–35
<body> element, 3–4, 5, 27–28, 34
boldface, <b> element, 59
bug reports, 12
<button> element, 54, 68

C

Camen, Kroc, 100
cancelEvent function, 179
canplaythrough and canplay events, 108
canPlayType method, 102–103
<canvas> element/canvases, 54
  accessibility, 139
  animating paintings, 134–137
  basics, 118–119
  capturing images, 126–129
  data URLs, saving to, 132–133
  drawing applications, 115–116
  drawing state, 137
  fill styles, gradients and patterns, 118–122
  Harmony application, 115, 117
  MS Paint replication, 115–116
  paths, 122–124
  pixels, pushing, 130–132
  rectangles, 118
    gradients and patterns, 118–120
  rendering text, 138–139
  resizing canvases, 122
  transformation methods, 124–126
case sensitivity, pattern attribute, 78
<center> element, 60
character encoding, UTF-8, 2
charset="utf-8" attribute, XHTML and XML *versus* HTML5, 2
checkValidity attribute, 86
checkValidity method, 85–86
Chisholm, Wendy, 51
cite attribute, 28
<cite> element, 58
classes
  attributes, 6, 8
  names, Google index research, 6
clear attribute, 147
clearInterval method, 127
clearRect method, 125
clearWatch method, 190
codecs, 98–99
color input type, 74
Comet, 212, 215
<command> element, 62, 65
comments as nested articles, 29–30

Contacts API, 70
<content> element, 9
content models, 54
contenteditable attribute, 61
contentWindow object, 199
context object, canvas attribute, 126
contextmenu attribute, 62
controls attribute, 54, 95–96
cookies, 142–143
Coordinated Universal Time (UTC), 26
coords object, 191
copyrights, <small> element, 18, 24, 60
Cotton, Paul, xii
createElement method, 121
createPattern method, 119–121, 126
createRadialGradient method, 120
Crockford, Douglas, 148
CSS (Cascading Style Sheets), 10
  <body> element requirement, 11
  display:inline, 54
  headers and footers for body and articles, 27–28
  IE, 5, 11–12
  outlines, 35–36
  WAI-ARIA, 50
CSS Basic User Interface module, 83
CSS Media Queries specification, 102

D

data-* attribute, 62, 112
data storage
  cookies, 142
  Web SQL Databases, 142, 152–162
  Web Storage API, 142–151
data URLs, 132–133
<datalist> element, 74–75
date input type, 70–71
dates, machine-readable, 26
datetime attribute, 26
datetime input type, 71
Davis, Daniel, 55
<dd> element, 57
definition lists, 57
<del> element, 54
delete method, 68
Designing with Progressive Enhancement: Building the Web that Works for Everyone, 51
<details> element, 34, 52–54
<device> element, 94
disclaimers, <small> element, 18, 24, 60
display:block, 12
display:inline, CSS, 54
<div> element, HTML 4, 7–8
<dl> element, 57
DOCTYPE, 2
<!doctype html> tags, 2
dragend event, 184
draggable attribute, 62
dragging and dropping
  accessibility, 184–185
  basics, 176–179
  custom drag icons, 183
  dragged data, interoperability, 180–182
  enabling elements for dragging, 182–183
DragonFly plug-in, 150
dragover event, 178
dragstart event, 179, 183–185
draw function, 136
drawImage method, 126–130
dropEffect method, 185
<dt> element, 57
durationchange event, 108

E

Eich, Brendan, xii
<em> element, 54–55, 58–60
email input type, 69–70, 82
<embed> element, 54, 64, 92–93
embedded content models, 54
emphasis effect, 54–55, 58–59
enableHighAccuracy method, 194
end method, 107
error handling, 192–193
event object, 198–199
executeSql method, 154, 158–161

F

"fat footers," 19–20
Faulkner, Steve, 50
ffmpeg library, 101
<fieldset> element, 34, 68, 86
<figcaption> element, 53
<figure> element, 34, 53
fill styles, gradients and patterns, 118–122
fillRect method, 119
fillStyle method, 119–121
fillText method, 138–139
Firebug plug-in, 149
Firefogg software, 101
Firefox Contacts addon, 70
flow content models, 54
<font> element, 60
<footer> element, 16, 18–20, 28
forEach method, 156

form attribute, 68
<form> element/forms
  <button> element, 68
  comments, 79
  <datalist> element, 74–75
  date pickers, 83
  delete, 68
  <fieldset> element, 68
  form fields, 83
  get, 68
  <input> element, 68
    onchange, 81
    type=...autocomplete, 74, 78
    type=...autofocus, 75
    type=color, 74
    type=date, 70–71
    type=datetime, 71
    type=email, 68–69, 82
    type=...list, 74–75
    type=...max, 74, 78
    type=...min, 74, 78
    type=month, 71
    type=...multiple, 69, 74, 76
    type=number, 72, 82
    type=...pattern, 74, 76–78
    type=...placeholder, 75–76
    type=range, 72–73, 80
    type=...required, 69, 76
    type=search, 73
    type=...step, 74, 78–79
    type=tel, 73, 82
    type=text, 68–69
    type=time, 71
    type=url, 70, 82
    type=week, 72
    update, 68
    validation
      built-in, 68
      JavaScript, 68
  <keygen> element, 68
  <label> element, 68
  <meter> element, 68, 80
  new types, 68
  <object> element, 68
  <output> element, 68
  post, 68
  <progress> element, 68, 80
  <select> element, 68
  sliders with values, 80–83
  <textarea> element, 68
  validation elements, 85–86
formats, consistent use, 3
formnovalidate attribute, 87
frames, removed from HTML5, 60
furigana/ruby, 55

G

geolocation API, xii, 187–195
get method, 68
getAttribute method, 112
getCurrentPosition method, 189–194
getData method, 178–180
getImageData method, 130–132
getItem method, 146–148, 151
getTime method, 156
getTweets method, 156
“The Guardian” case study, 42–47

H

h1..h6 elements, 54
H.264 specification, 98–101
Harmony application, 115, 117
<head> element, 2–4, 12
<header> element, 13–15
heading content models, 54
height attribute, 96–97
<hgroup> element, 13, 33–34
Hickson, Ian, iii, xi–xiii, 6, 175
hidden attribute, 62
highlighter pen effect, 54–55
hiragana alphabet, 55–56
Hiroshi Ichikawa, 212
<hr> element, 59–60
HSLA color picker, 88–89
<html> tags
  importance, 4–5
  optional tags, 3–4
  primary language declaration, 4–5
HTML5
  <content> element, 9
  history, x
  <http://html5.validator.nu> tag, 5
  philosophies, xiii
  W3C specification, x
  WHATWG (Web Hypertext Application Technology Working Group) specification, x–xiv
  XML and XHTML, xi-xii, xvi, 2–3,
“The HTML5 <ruby> element in words of one syllable or less,” 55
html5 shiv, 54
html5canvas library, 118
HTMLElement object, 112
<http://html5.validator.nu> tags, 5
Hyatt, David, xii

I

<i> element, 59
Ichikawa, Hiroshi, 212
IDs, names in Google index research, 6
<iframe> element, 54, 60
<img> element, 54, 94
importScripts method, 207, 210
"Incite a riot," 58
inline elements, 54
<input> element
  forms, 68
  onchange attribute, 81
  type attribute
    autocomplete, 74, 78
    autofocus, 75
    color, 74
    date, 70–71
    datetime, 71
    email, 68–69, 82
    list, 74–75
    max, 74, 78
    min, 74, 78
    month, 71
    multiple, 69, 74, 76
    number, 72, 82
    pattern, 74, 76–78
    placeholder, 75–76
    range, 72–73, 80
    required, 69, 76
    search, 73
    step, 74, 78–79
    tel, 73, 82
    text, 68–69
    time, 71
    url, 70, 82
    week, 72
<ins> element, 54
INSERT statements, 156–157
insertId attribute, 158
interactive content models, 54
Internet Archive, 101
"Introduction to WAI-ARIA," 51, 184
italics, <i> element, 59
item attribute, 63
itemprop attribute, 63

J

Japanese language, 55–56
JavaScript
  <body> element requirement, 11
  degrees to radians conversion, 120
  element validation, 85–86
  focus command, tabindex attribute, 63
  form validation, 68
    Modernizr library, 82
  IE application of CSS to HTML5, 11–12
  IE Print Protector, 12
  library, 75
  media API, 102–104
  Modernizr library, 82
  outlines, 31
  pattern attribute, 77
  polyfilling, 75
  PPK on JavaScript, 112
jQuery library, 134
jQuery Visualize, 139
JSON library, 148

K

Keith, Jeremy, 58
key method, 146–147
<keygen> element, 54, 64–65, 68
Koch, Peter-Paul, 112, 141–142

L

<label> element, 54, 68
Langridge, Stuart, 54
legacy browsers
  backwards compatibiity, 82–83
  <body> element requirement, 11
  input type problems, 68–79
  multimedia elements, 100–101
  <script> element, JavaScript default, 11
  styling, 12
  videos, 94–98
legal restrictions, <small> element, 18, 24, 60
Lemon, Gez, 51, 184
Levithan, Steven, 76
list input type, 74–75
lists
  definition lists, 57
  ordered lists, 56–57
  unordered lists, 16
load method, 102–103
loadeddata event, 108–109, 128
loadstart event, 108
localStorage method, 143–144, 146, 149–150, 200
loop attribute, 97

M

machine-readable data
  dates and times, 16
  microdata attribute, 65

MAMA crawler, Opera, 6
<mark> element, 54–55
<marquee> element, 60
max attribute, 74, 78
maximumAge method, 194
media. *See also* <audio> element; <video> element
  accessibility, 110–113
  attributes, 102–104
  codecs, 98–100
    H.264 specification, 98–101
    handheld devices, 101–102
    legacy browsers, 100–101
    software, 101
    <source> elements, multiple, 99–100
  custom controls, 102–110
  events, 102–104, 106–108
  HTML5 shortcomings, 94
  Internet Archive, 101
  methods, 102–104
  royalty-free, 101
media attribute, 102
<menu> element, 54, 62, 65
message property, 193
Messaging API, 198–200
<meta charset=utf-8> tags, 2
<meta> tags, XHTML and XML *versus* HTML5, 2–3
metadata content models, 54
<meter> element, 65, 68, 80
microdata attribute, 65
Microsoft Word 2007 outline view, 30
Mill, Bill, 134
min attribute, 74, 78
Miro Video Converter, 101
Modernizr library, 82
month input type, 71
moveTo method, 123
MS Paint replication, 115–116
multimedia. *See* media
multiple attribute, 69, 74, 76

N

Nas, Will, 73
<nav> element, 15–18, 33, 54
Newhouse, Mark, 16
Nitot, Tristan, 130–131
novalidate attribute, 87
number input type, 72, 82
NVDA screen reader, 51

O

<object> element, 54, 68
offline
  applicationCache, 164, 171–172, 174
  browser-server process, 168–171
  CACHE MANIFEST, 164–167
  FALLBACK, 165–167, 172–173
  killing caches, 174
  NETWORK, 167
  serving manifests, 168
Ogg Theora and Vorbis codecs, 98, 101
OggConvert software, 101
 element, 16, 56–57
onchange attribute, 81
ondragover event, 177
ondrop event, 177–178
onforminput event, 80
oninputchange event, 88–89
onload event, 121
onmessage event handler, 199, 202–206, 209–210, 213–215
open attribute, 53
openDatabase method, 154–155
ordered lists, 56–57
outlines/outlining algorithm
  accessibility, 36–37
  <article> element, 37–42
  case study, 42–47
  <hgroup> element, 33–34
  JavaScript implementation, 31
  Microsoft Word 2007 outline view, 30
  <section> element, 31–33, 37–41, 41–42
  sectioning content, 31
  sectioning roots, 34–35
  styling with CSS, 35–36
  tool at gsnedders.html5.org/outliner/, 31–32
<output> element, 68, 80–81

P–Q

The Paciello Group, ARIA information, 50–51
paragraph-level thematic breaks, <hr> element, 59–60
Parker, Todd, et al, 51
path API/paths, 122–124
pattern attribute, 74, 76–78
pause method, 102–103
Pfeiffer, Silvia, 113
phrasing content models, 54
Pieters, Simon, 12
placeholder attribute, 75–76
play method, 102–103
playbackRate attribute, 109–110

polyfilling, 75
post method, 68
poster attribute, 96
postMessage method, 198–199, 202–210, 213
PPK on JavaScript, 112
preload attribute, 97, 109
processing.js library, 134
<progress> element, 65, 68, 80
progress event, 108
pubdate attribute, 27
public-key cryptography, 65
putImageData method, 132

R

radians, 120
range input type, 72–73, 80
Raphael library, 124
rectangles, gradients and patterns, 118–120
regular expressions, 76–77
removeItem method, 147
required attribute, 69, 76
Resig, John, 12, 134
restore method, 137
reversed attribute, 57
RGBA color picker, 88
role attribute, 63
role=main tags, WAI-ARIA, 9
rotate method, 124–126
Rouget, Paul, 130–131
rowAffected attribute, 158
rows attribute, 158
<rp> element, 55–56
<rt> element, 55–56
Ruby, Sam, xiv
<ruby> element, 55–56

S

save method, 137
saveTweets method, 156
scalar measurements, 65
scale method, 124
scoped attribute, 65
screen readers
  HTML5 and ARIA, 51
  problems, 64
<script> element, 11
search input type, 73
Searchhi script, 54
<section> element, 18, 33, 37–42, 54, 85–86
sectioning content, 18, 31
  models, 54
sectioning roots, 34–35
<select> element, 54, 68
SELECT statements, 158
sessionStorage method, 143–151
setAttribute method, 112
setCustomValidity method, 84–85
setData method, 179–181
setDragImage method, 183
setInterval method, 125, 127, 203
setItem method, 146–148, 151
setOnline method, 173
setTimeout method, 203
sidebars, 17–18
Silverlight, 118
<small> element, 18, 24
<source> element, 99–100
spellcheck attribute, 63
SQLite, 152
src attribute, 98
Stachowiak, Maciej, xii
start attribute, 56
start method, 107
step attribute, 74, 78–79
strokeRect method, 119
strokeStyle method, 119
<strong> element, 55, 59
<style scoped> element, 65
subject attribute, 63
<summary> element, 52
SVG (Scalable Vector Graphics) API, x, 54, 124,
swapCache method, 171–172
syntax, consistent use, 3

T

tabindex (+"-1") attribute, 63–64, 185
"Taming Lists," 16
<td> element, 34
tel input type, 73, 82
testOnline method, 173
<textarea> element, 54, 68, 85–86
time
  machine-readable, 26
  UTC (Coordinated Universal Time), 26
<time> element, 16, 26–27
time input type, 71
timeout method, 194
timestamp object, 191
timeupdate event, 111, 128
TinyOgg software, 101
toDataURL method, 132–133
transaction method, 161–162
transform method, 124

translate method, 124–126, 137–138
Twitter API, 155–161
2D canvas API, 115, 117, 124
type attribute
  <input> element, 54
    autocomplete, 74, 78
    autofocus, 75
    color, 74
    date, 70–71
    datetime, 71
    email, 68–69, 82
    list, 74–75
    max, 74, 78
    min, 74, 78
    month, 71
    multiple, 69, 74, 76
    number, 72, 82
    pattern, 74, 76–78
    placeholder, 75–76
    range, 72–73, 80
    required, 69, 76
    search, 73
    step, 74, 78–79
    tel, 73, 82
    text, 68–69
    time, 71
    url, 70, 82
    week, 72
  <menu> element, 54

U

 element, 16
Universal Design for Web Applications, 51
unordered lists, 16
update method, 68
updateSeekable function, 108
url input type, 70, 82
usemap attribute, 54
UTC (Coordinated Universal Time), 26
UTF-8 character encoding, 2

V

valid attribute, 86
validation
  ARIA, 49
  avoiding, 86–87
  built-in for forms, 68
  custom messages, 84–85
  elements with JavaScript, 85–86
  <http://html5.validator.nu> tag, 5
  pros and cons, 5
validity attribute, 86
van Kesteren, Anne, xiv, 92, 102,
<video> element, 54
  attributes, 95–98
  legacy browsers, 100–101
  reasons needed, 92–93
  sources, 99–100
"Video for Everybody!", 100
VLC software, 101
VoiceOver screen reader, 51

W

W3C
  Geolocation API, 187
  HTML5 specification, xiv
WAI-ARIA (Web Accessibility Initiative's Accessible Rich Internet Applications) suite, 48–49
  attributes
    aria-required, 76
    aria-valuenow, 81–82
  document landmarks and structure, 49–50
  HTML5, combining with, 50
  information not built into HTML5, 50
  resources, 50–51
  role=main tags, 9
  screen readers, 51
  specification, 51
  transitional accessibility, 81–82
watchPosition method, 189–194
Web Applications 1.0, xi-xii
"A Web Developer's Responsibility," 12
Web Sockets API, x, 212–215
Web SQL Databases, 142, 152–162, 208
Web Storage API, x, 142–151
Web Workers API, 198, 200–211
WebKit browsers, 82
week input type, 72
WHATWG (Web Hypertext Application Technology Working Group), 111
width attribute, 96
willValid attribute, 86
Wilson, Chris, xiv
window object, 198–199
ws:// server protocol:, 213

X–Z

XHTML *versus* XML and HTML5, 2–3
XML *versus* HTML5 and XHTML, 2–3
XMLHttpRequest object, 198, 203, 210, 212