THE EXPERT'S VOICE® IN WEB DEVELOPMENT

HTML5 Programmer's Reference

BUILD THE NEXT GENERATION OF MODERN WEB APPLICATIONS TODAY

Jonathan Reid

Apress®

HTML5 Programmer's Reference

Jonathan Reid

Apress®

HTML5 Programmer's Reference

Copyright © 2015 by Jonathan Reid

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

ISBN-13 (pbk): 978-1-4302-6367-8

ISBN-13 (electronic): 978-1-4302-6368-5

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director: Welmoed Spahr
Lead Editor: Ben Renow-Clarke
Technical Reviewer: Victor Sumner
Editorial Board: Steve Anglin, Mark Beckner, Gary Cornell, Louise Corrigan, Jim DeWolf, Jonathan Gennick, Robert Hutchinson, Michelle Lowman, James Markham, Susan McDermott, Matthew Moodie, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Gwenan Spearing, Matt Wade, Steve Weiss
Coordinating Editor: Christine Ricketts and Melissa Maldonado
Copy Editor: James Fraleigh
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–

eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this text is available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.

For my husband Steve, who is always there for me.

Contents at a Glance

About the Author

About the Technical Reviewer

Acknowledgments

Introduction

- Part I: HTML5 in Depth
- Chapter 1: Welcome to HTML5
- Chapter 2: HTML5 Elements
- Chapter 3: HTML5 APIs
- Chapter 4: Canvas
- Chapter 5: Related Standards
- Chapter 6: Practical HTML5

- Part II: HTML5 Reference
- Chapter 7: HTML5 Element Reference
- Chapter 8: HTML5 API Reference
- Chapter 9: Canvas Reference
- Appendix A: JavaScript Tips and Techniques

Index

Contents

About the Author

About the Technical Reviewer

Acknowledgments

Introduction

Part I: HTML5 in Depth

Chapter 1: Welcome to HTML5

What Is HTML5?

A Brief History of HTML

- Enter Hypertext
- Enter Markup Languages
- Hypertext Markup Language Is Born
- The Browser Wars
- Standards to the Rescue
- The Continuing Evolution of HTML
- The Formation of the WHATWG and the Creation of HTML5

HTML5 Features

- New Tags
- JavaScript APIs
- Related Standards

Summary

Chapter 2: HTML5 Elements

Functionality, Semantics, and the Evolution of HTML

Sections

Grouping

Semantics

Audio and Video Content

- Embedded Audio Content
- Embedded Video Content
- Specifying Multiple Sources

Interactive Elements

- Dialogs
- Progressive Disclosure

Forms

- New Form Elements
- New Form Element Attributes
- New Input Types

Deprecated Elements and Obsolete Parameters

Summary

Chapter 3: HTML5 APIs

Server-sent Events

- Client Setup
- Sending Events from the Server
- Origin Limitations
- Security
- An Example Application

WebSockets

- Connecting to the Server: Inside the WebSocket Handshake
- Receiving Information from the Server
- Sending Information to the Server
- Closing the Connection
- An Example WebSocket Application

Cross Document Messaging/Web Messaging

Web Storage

- Methods and Syntax
- Privacy and Web Storage

Drag and Drop

- The draggable Property
- Drag-and-Drop Events

The dataTransfer Object
Drag-and-Drop API Examples
Web Workers
Creating Web Workers
Inside a Web Worker
A Simple Example of a Web Worker
Common Use Cases
Summary
Chapter 4: Canvas
The Canvas Drawing Mode
The Canvas Drawing Context
Basic Drawing Commands
Gradients and Patterns
Gradients
Patterns
Images
Saving Canvas Contents
Text
Shadows
Saving Drawing State
Compositing
Clipping
Transformations
Animation
Interaction
Summary
Chapter 5: Related Standards
Geolocation
Privacy Considerations

Geolocation API
Animation Timing
Selectors
Device Orientation
The compassneedscalibration Event
The deviceorientation Event
The devicemotion Event
WebGL
SVG
Summary
Chapter 6: Practical HTML5
Browser Support
A Crash Course in Feature Detection
Building a Feature Detection Script
Working with Broken or Missing HTML5 Implementations
Example Project: MobiDex, a Mobile Dexterity Puzzle
The Playing Field UI
Generating Obstacles and Targets
Summary
Part II: HTML5 Reference
Chapter 7: HTML5 Element Reference
Sections
The article Element
The aside Element
The footer Element
The header Element
The nav Element
The section Element
Grouping
The figure and figcaption Elements
The main Element

Semantics

- The bdi Element
- The data Element
- The mark Element
- The ruby, rp, and rt Elements
- The time Element
- The wbr Element

Audio and Video Content

- The audio Element
- The source Element
- The track Element
- The video Element

Interactive Elements

- The details and s ummary Elements

Form Elements

- The datalist Element
- The meter Element
- The output Element
- The progress Element

Chapter 8: HTML5 API Reference

Server-sent Events

WebSockets

Cross-Document Messaging/Web Messaging

Web Storage

Drag and Drop

- Specifying Draggable Elements: The draggable Property
- Handling the Interactions: Drag-and-Drop Events
- Specifying Drop Targets
- The dataTransfer Object

Web Workers

Chapter 9: Canvas Reference

The canvas Element

The Drawing Context
Defining Paths
Basic Drawing Commands
Stroking and Filling Paths
Drawing Rectangles
Gradients and Patterns
Images
Text
Shadows
Compositing
Clipping
Transformations
Saving and Restoring Canvas State

Appendix A: JavaScript Tips and Techniques

Code Formatting

JavaScript Rewards Verbosity

Comment Annotations

Using Objects as Event Handlers

Promises

Chaining Promises
Returning Values from Promises
Browser Support for Promises
Further Reading

Index

About the Author

Jonathan Reid has been building web applications professionally for almost two decades. He has built everything from simple informational websites all the way up to complex scientific and business applications, all using HTML, CSS, and JavaScript. In addition, Jon has worked at some of the top interactive agencies in the United States (EffectiveUI, Crispin Porter + Bogusky), so he knows what it is like to implement difficult and complex requirements with web technologies on a tight deadline. He currently works for Google, where he is a Senior User Experience Engineer on Google Web Designer, an HTML5 authoring tool. Jon lives in Sunnyvale with his husband of 15 years.

About the Technical reviewer

Victor Sumner is a senior software engineer at D2L Corporation, where he helps to build and maintain an integrated learning platform. As a self-taught developer, he is always interested in emerging technologies and enjoys working on and solving problems that are outside his comfort zone.

When not at the office, Victor has a number of hobbies, including photography, horseback riding, and gaming. He lives in Ontario, Canada, with his wife, Alicia, and their two children.

Acknowledgments

Writing a book is never a solitary activity. I had a lot of help along the way.

Pushkar Joshi, my colleague at Google, provided input on the Canvas chapter as well as some great suggestions for topics to include in the chapter. He was also kind enough to look through the chapters and provide feedback.

Victor Sumner did a great job as the technical reviewer, going through every line of code in this book and testing them thoroughly. Code reviews are practically a way of life for me, so I value my reviewers greatly, and Victor did a great job.

The rest of the Google Web Designer team, led by Sean Kranzberg, Tony Mowatt, and San Khong, put up with me obsessing about this book off and on for over a year. Thanks, everyone.

My husband, Steve, was patient while I barricaded myself in our home office on Saturdays to write.

And as always, my editors at Apress were supportive, patient, and diligent.

Introduction

The World Wide Web has existed for almost 25 years now. It started as a simple proposal by Tim Berners-Lee and Robert Cailliau as a way for the scientists at CERN to publish papers easily, but it rapidly grew into a platform that captured the imagination of the world.

The Web may have started out as a simple document-publishing platform, but it quickly became clear that it was destined to become much more than that. As people demanded more interactivity and richer experiences, the limits of the original HTML standards quickly became obvious. The advent of other technologies like Cascading Style Sheets and JavaScript helped, but developers were still spending extensive resources on building the experiences that people wanted.

HTML5 is meant to help solve some of these problems. The fifth incarnation of the HTML standard, HTML5, is designed to be both feature rich and easier to work with. Prior versions of HTML focused on how best to standardize document markup, which was a great way to bring standards to the chaos of the early Web. HTML5, however, is focused on providing a platform for building rich interactions. Much of HTML5 was also specifically designed with mobile technologies in mind, while older versions of HTML were not.

What This Book Covers

This book is designed to be your go-to reference for HTML5 features. It is divided into two sections.

Part I, "HTML5 in Depth," has chapters that provide detailed examinations of the HTML5 features, including multiple examples and the current level of support at press time.

- Chapter 1, "Welcome to HTML5," is a history lesson, explaining how the World Wide Web and its technologies evolved, and how HTML5 came to be. This will hopefully help you understand why HTML5 is so much different from previous HTML standards, and will give you better insight in how HTML5 is structured.
- Chapter 2, "HTML5 Elements," covers the new semantic tags in

HTML5. As with predecessor standards, HTML5 includes a new set of tags for marking up the content in your documents. This is where you'll find out how to use the new audio and video tags, as well as a host of other HTML5 features.

- Chapter 3, "HTML5 APIs," dives into the JavaScript APIs that are specified in the HTML5 standard. You'll learn about new ways for your HTML5 applications to communicate and save data.
- Chapter 4, "Canvas," covers one of the most innovative features of HTML5: the `canvas` element. Here you'll learn how to use this element to draw, modify images, and create animations.
- Chapter 5, "Related Standards," covers several JavaScript APIs that are related to HTML5 (and frequently used with HTML5) but are not actually a part of the HTML5 standard. These APIs also tend to have a strong mobile focus.
- Chapter 6, "Practical HTML5," covers actually working with HTML5 in production projects. It covers detecting features and applying shims, and includes a complete HTML5 mobile game designed and built from the ground up.

Part II, "HTML5 Reference," contains reference chapters for all of the HTML5 features covered in Part 1. Each chapter is designed to provide an at-a-glance reference for each feature and includes a brief description of the feature, how it is used (including both syntax and examples), and where to find its standards.

- Chapter 7 is the reference chapter for HTML5 elements.
- Chapter 8 is the reference chapter for the HTML5 JavaScript APIs.
- Chapter 9 is the reference chapter for the `canvas` element.

What You Need to Know

Though there are lots of detailed examples throughout the book, it is written as a reference and not as a tutorial. I'm assuming you have an intermediate understanding of how browsers work as well as how to work with JavaScript, and at least a basic understanding of CSS and the network protocols involved in HTTP. You should be comfortable creating and editing web pages and writing your own CSS and JavaScript.

Running the Code Samples

There are extensive code samples throughout the book. You can download the samples from www.apress.com, or you can type them in by hand. Many of the examples can be run by simply loading the file into a web browser using the browser's File menu.

Some examples, though, must be run from an actual server, either due to security limitations or because you will want to view them on a mobile device. To build and test all the examples in the book, I've used Aptana Studio, available for free at http://www.aptana.com. Aptana Studio comes with an internal debugging server that you can use to run any of these examples. If you prefer a stand-alone solution, I've had very good luck with XAMPP, a stand-alone installation of the Apache web server, along with optional components like MySQL, PHP, and Perl. And of course both MacOS and Windows come with their own web server solutions that you can activate and use, as do most standard Linux distributions.

Finally, be sure to check out the "Comment Annotations" section in Appendix A, "JavaScript Tips and Techniques," for an explanation of the format of the examples and how to read the annotations.

PART I

HTML5 in Depth

CHAPTER 1

Welcome to HTML5

In this chapter, I'm going to dive into the history of HTML and how HTML5 came to be. I'll talk about the evolution of HTML from a simple proposal all the way to its current version, including reviews of related technologies. I'll also cover what HTML5 is, its scope, how it differs from previous versions, and how it fits in with other technologies.

What Is HTML5?

Hypertext Markup Language, or HTML, has been with us since 1989. Versions of HTML prior to 5 only defined markup tags for content: lists, paragraphs, headers, tables, and so on. HTML5, though, defines much more. It has new content tags (such as `<audio>` and `<video>`) but it also defines complex interactions like dragging and dropping, new network interfaces like server events, and even has new asynchronous functionality like web workers. HTML specifications prior to HTML5 also defined the tags in SGML (more on that in a bit), but the HTML5 specification is careful only to define tags in terms of annotated content and expected behavior. And because HTML5 is a big part of a new set of advanced web technologies, many times you'll see articles on the Web or in popular media that mistakenly include technologies in HTML5 that have nothing to do with HTML.

So what exactly is HTML5? Why does HTML5 define so much more than tags? How did HTML5 come about? Why is the HTML5 standard such a big departure from previous standards, both in terms of definition and scope? To answer these questions, I'll start with a quick review of how HTML came to be in the first place.

A Brief History of HTML

HTML's humble beginnings go all the way back to 1989. At that time the most common ways of sharing information online was via e-mail, Usenet newsgroups, and public FTP sites. E-mail and newsgroups made it easy for people to communicate directly with one another, and FTP sites provided a way for people to provide access to sets of files. The main issue is that all of these forms of sharing information required different software, and a certain level of skill to be able to actually navigate the Internet—though at that time the Internet was considerably smaller than it is today.

Tim Berners-Lee proposed a better solution in 1989. At the time he was working at the European Organization for Nuclear Research (known better by its French acronym CERN, for Conseil Européen pour la Recherche Nucléaire), and he was keenly aware of the need for a better way to share information online. In particular, Berners-Lee needed to solve the problem of sharing technical documents online. CERN produced huge amounts of technical documentation, ranging from nuclear physics papers intended for publication to internal policy documents, and they needed a solution that would work for all of these different use cases.

Berners-Lee found himself trying to solve two problems at once:

- He needed a solution that provided a way of visually formatting the information that CERN scientists were producing. This information could take the form of documents such as published papers as well as data observed during experiments.
- He needed a solution that was capable of handling cross-references and embedding graphics and other media. Many of the documents and at CERN included diagrams and graphics and referenced one another, or other internal data sources, or even external documents and data sources.

Fortunately, Berners-Lee already had some experience with solving these problems. Back in 1980 when he was a contractor at CERN he had built a prototype system called ENQUIRE that provided some of the functionality the organization needed, but failed to scale well. It did, however, employ a very important key concept: hypertext.

Enter Hypertext

Hypertext is text with references to other information that the user can activate to gain immediate access to that information. This includes information contained within the same document as well as information in external documents or other data sources. These links are referred to as *hyperlinks*. In the case of most modern computers, hypertext is displayed on the screen and hyperlinks are activated by clicking them with a

mouse or (in the case of touchscreens) tapping them with a finger. The term *hypermedia* is an extension of the concept of hypertext to include not only hyperlinks but graphics, audio, video, and other sources of information.

The concept of hypermedia has been around for quite some time. In 1945 American engineer and inventor Vannevar Bush wrote an essay titled "As We May Think" for the *Atlantic Monthly*. As part of the essay Bush proposed a "memory extender" or "memex," a device that people can use to store all of their personal information sources: books, records, albums, and so on. The memex would provide a person with access to all of their information through the use of a set of bookmarks, and could be expanded as needed.

Tip You can read "As We May Think" on the *Atlantic*'s web site at www.theatlantic.com/magazine/archive/1945/07/as-we-may-think/303881/.

In 1960 Ted Nelson founded Project Xanadu in an attempt to build a word processing system capable of storing multiple versions of many documents that would allow the user to move through those documents in a nonsequential fashion. He referred to these nonsequential paths as "zippered lists" and posited that by using these zippered lists, new documents could be formed from pieces of other documents in a process he called "transclusion." In 1963 Nelson coined the terms *hypertext* and *hypermedia*, which were first published in his paper "Complex Information Processing: A File Structure for the Complex, the Changing and the Indeterminate" (available at http://dl.acm.org/citation.cfm?id=806036). At that time Nelson used *hypertext* to refer to editable text rather than a text-based cross-reference, so the term has had some semantic drift since Nelson first coined it.

Tip Project Xanadu is alive and well today at www.xanadu.com/, and even has a demonstration of a "xanadoc" that is created on demand via transclusion.

In 1962 American engineer and inventor Douglas Englebart began work on his "oNLine System" or "NLS." The NLS was the first system that included most of the modern computer features available today: a pointing device, windows, separate programs for presenting different kinds of data, information organized by relevance, hypermedia links, and so forth. Englebart demonstrated the NLS at the Fall Joint Computer Conference in San Francisco in December 1968. This demonstration was groundbreaking not only because it was the first to show all of these modern features in

use at once, but also because it used state-of-the-art video conferencing technology to show the user interface for NLS as Englebart used it. Because the demonstration was so groundbreaking in scope, it is often referred to as “The Mother of All Demos.”

Tip The demonstration can be seen on Stanford University’s web site at http://web.stanford.edu/dept/SUL/library/extra4/sloan/Mous

---

Berners-Lee had built ENQUIRE on the concept of hypertext (Figure 1-1). Within ENQUIRE a given document was represented by a single page of information called a “card,” which was essentially a list of hyperlinks defining what the document included, how it was used, a description, and who the authors were. The links could easily be followed by activating them, allowing the user to explore the entire network of documents.

```
@ENQUIRE
Enquire V 1.1

Hello!
Opening file (PSK-PCP)VAC-V1:ENQR...

PSB Vacuum Control System                          (concept)  <   0>
--- ------ ------- ------


 [ 1] described-by: Enquiry System
      An experimental system for which this is a test.

 [ 2] includes: Vacuum History System
      Records and displays slow changes in pressure.

 [ 3] includes: Vacuum Equipment modules
      Perform all the hardware interface

 [ 4] includes: Control and status applications programs
      Provide operator interaction from the consoles.

 [ 5] described-by: Controle du System a Vide du Booster 11-2-80
      Operational specification of the software

 [ 6] includes: PSB Pump Surveillance System          PCP 228
      Allows rapid monitoring of pressure changes

[number      ]
```

Figure 1-1. *A Screenshot of ENQUIRE*

In this respect, ENQUIRE was similar to an online version of a library's card catalog system, and unfortunately required a significant amount of effort to keep updated.

ENQUIRE also didn't address the requirement for visually formatting documents. However, CERN already made use of a possible solution in the form of a document markup language.

Enter Markup Languages

Document markup languages are programming languages that provide a way to annotate (or "mark up" as an editor marks up a document under review) a document in such a way that the annotations are syntactically distinct from the main content document. Markup languages exist in three broad categories based on the goal of the annotations:

- *Presentational* markup languages are used to describe how a document should be presented to the user. Most modern word processors use presentational markup in the form of binary codes embedded in the document. Presentational markup is typically designed for a specific program or display method and thus not meant to be human readable.
- *Procedural* markup languages provide annotations that specify how the contents of the document should be processed, often in the context of layout and typesetting for printing. One of the most common examples of a procedural markup language is PostScript.
- *Descriptive* markup languages are used to annotate the document with descriptions of its content. Descriptive markup does not give any indication of how the contents should be processed or displayed; that is left up to the processing agent.

Document markup languages have existed for decades. The first widely known document markup language was presented by computer scientist William Tunnicliffe in 1967, but IBM researcher Charles Goldfarb is typically called the "father" of modern markup languages because of his invention of the IBM Generalized Markup Language (GML) in 1969. Goldfarb was responsible for pushing IBM to include GML in its document management solutions. GML would eventually evolve into Standard Generalized Markup Language (SGML, which became an ISO standard (ISO 8879:1986

Information processing—Text and office systems -- Standard Generalized Markup Language) in 1986 with Goldfarb as the chair of the committee.

SGML isn't a language you use directly; instead it is a "meta-language"—a language that is used to define other languages. In this case, SGML is used to define markup languages that can then be used to describe documents. Specifically, SGML requires that markup languages describe a document's structure and content attributes (vs. describing how to process the document), and that the markup languages be rigorously defined so that processing and viewing software can be built that follows the same rules. Languages defined by SGML are referred to as "SGML applications" (not to be confused with applications that run on computers and perform tasks). Common SGML applications include XML (the eXtensible Markup Language) and DocBook (a markup language designed for technical documentation).

CERN had been using an SGML application called SGMLguid) for marking up its documents, and Berners-Lee recognized that a combination of SGMLguid with hypertext could be the solution he needed for CERN's document management problems.

Hypertext Markup Language Is Born

In late 1989, Berners-Lee proposed a pilot project that would employ hypertext and a simple markup language as its basis. Berners-Lee envisioned that hyperlinks would be the key feature that tied all of the disparate documents together:

> *HyperText is a way to link and access information of various kinds as a web of nodes in which the user can browse at will. Potentially, HyperText provides a single user-interface to many large classes of stored information such as reports, notes, data-bases, computer documentation and on-line systems help.*
>
> from "WorldWideWeb: Proposal for a HyperText Project," 12 November 1990 (`www.w3.org/Proposal.html`)

This proposal outlined a simple client/server network protocol for the new "web" of documents and how they would function together to transfer information from the server to the viewing client. Berners-Lee dubbed the new protocol "the Hypertext Transfer Protocol" or HTTP. The project was approved, and Berners-Lee and his team began working on what would eventually become the World Wide Web.

After creating both client and server software for the new document system, Berners-Lee published the first document that defined a basic set of tags that could be used to mark up documents that were to be included in the new online document web.

This document, titled “HTML Tags,” defined 18 tags that could be used to mark up the contents of a document in such a way that the new web clients would be able to parse and display them. Almost all of the tags came from SGMLguid except one: the anchor tag. The anchor tag was the implementation of the hypertext linking feature that was so important to the new system.

Tip You can read the original “HTML Tags” document in the W3C’s historical archive at `www.w3.org/History/19921103-hypertext/hypertext/WWW/MarkUp/Tags.html`.

This first document was a simple list of tags with a description of how to use them to describe the content of a document. Later the tags were formalized as an SGML application in 1993 with the publication of “Hypertext Markup Language (HTML)”(`www.w3.org/MarkUp/draft-ietf-iiir-html-01.txt`) as a working draft submitted to the Internet Engineering Task Force (IETF). This draft expired and was followed by a competing draft titled “HTML+ (Hypertext Markup Format)” later that same year, authored by Dave Raggett.

OPEN AND COLLABORATIVE

Tim Berners-Lee worked to keep the HTML definition process open and collaborative, leveraging the knowledge and experience of many participants. These early collaborations paved the way not only for the creation of the entire web technology ecosystem that would be designed through public collaboration, but also for the creation of the core groups that would maintain the projects.

The Browser Wars

While working on the definition of HTML, Tim Berners-Lee was also working on the first software that could make use of the new web of documents. In 1991 Berners-Lee released the first web browser, “WorldWideWeb,” for the NeXTStep platform. There was significant interest from other programmers in developing their own web browsers, so in 1993 Berners-Lee released a portable C library called `libwww` to the public domain so that anyone could work on building web browsers. (The library was available prior to that as part of the larger WorldWideWeb software application.)

By this point there were several experimental web browser projects on multiple platforms. Some of these were simple text-based browsers that could be used from any

terminal, such as the Lynx browser. Others were graphical applications for use in the graphic desktops of the time.

One of the most popular of the graphical applications was Mosaic, developed at the National Center for Supercomputing Applications (NCSA) at the University of Illinois. Work on Mosaic was begun in late 1992 by Marc Andreesen and Eric Bina, with the first release in 1993.

In 1994 Andreesen left the NCSA to found a company called Mosaic Communications, where they built a new browser from entirely new code. The new browser was called Netscape Navigator (and eventually Mosaic Communications was renamed to Netscape Communications).

The actual Mosaic code base itself was licensed from the NCSA by a company called Spyglass, Inc. Spyglass never did anything with the code, and in 1995 Microsoft licensed the code from them, modified it, and renamed it Internet Explorer.

Both Netscape and Microsoft began expanding the capabilities of their browsers, adding new HTML tags and other features. Netscape added JavaScript (codenamed "mocha" and originally released as "LiveScript") in Navigator in 1995. Microsoft quickly followed with their own version of the same language, called JScript to avoid trademark issues, in 1996.

Netscape Navigator and Internet Explorer both had radically different implementations of the same features, as well as their own proprietary features. A given HTML document could render one way in Navigator and look completely different when rendered by Internet Explorer. Even simple HTML markup produced significantly different visual results in the two browsers, and any attempt to do anything more advanced was simply not possible.

This set the stage for the so-called Browser Wars. Anyone producing content for the Web had to make a choice: choose a single browser to support, or spend significant resources to try and support both (in many cases this meant producing two different versions of the same content, one version for each browser). It became commonplace to see web sites that were optimized for only one browser, with graphics indicating the choice, as shown in Figure 1-2.

Figure 1-2. *Graphics from the Browser Wars*

Microsoft handily won the first round of the Browser Wars by including Internet Explorer as a standard part of the Windows operating system. This gave Internet Explorer a huge base of installations and little reason for people to pay for Netscape. By 1999 Internet Explorer made up 96% of the browser usage on the World Wide Web. Netscape Communications was acquired by AOL, and Netscape Navigator (by then called Netscape Communicator) was mothballed.

BROWSER WARS: NETSCAPE STRIKES BACK

AOL open-sourced the Netscape Communicator code base and entrusted it to the newly formed nonprofit organization named the Mozilla Foundation. The Mozilla Foundation continued to build upon the Navigator code base as an open source project and gained considerable momentum, adding new features to the browser including e-mail and HTML editing features. In late 2002 a stripped-down browser-only version of the suite was created, initially called Phoenix, then Firebird, and then later (due to project naming conflicts) Firefox. Firefox went on to successfully challenge Internet Explorer's hold on the browser market in what many people refer to as the second round of the Browser Wars.

Standards to the Rescue

Combating the fragmentation of the Web meant bringing all parties to the table and agreeing upon technology standards everyone could build upon. Standards provided a common ground for both browser manufacturers and content creators:

- By adopting standards as part of their manufacturing process, browser manufacturers would provide a predictable platform for the Web.
- By adapting standards as part of their coding practices, content creators could be assured that their content would render consistently across all browsers.

In October 1994, that's exactly what Tim Berners-Lee did, in a move that harkened back to his desire to keep the Web open and collaborative. He left CERN and formed the World Wide Web Consortium (W3C), a standards organization devoted to web technologies. The consortium was made up of anyone who wanted to participate in defining and maintaining the standards for web technologies: companies that eventually included Microsoft, Apple, Facebook, and Google; government organizations like NASA and National Institute of Standards and Technologies; universities like Stanford University and the University of Oxford; research organizations like CERN; and nonprofit organizations like the Mozilla Foundation and the Electronic Frontier Foundation.

The W3C standards process starts by publishing a working draft for a standard. The consortium members can then comment on the draft, which can undergo considerable evolution. Once the draft has solidified, a candidate recommendation is published. Candidate recommendations are reviewed from an implementation viewpoint—how difficult will it be to implement and use the standard. Once the implementers have had their say, the draft moves to the proposed recommendation status. Proposed recommendations go before the W3C advisory council for final approval. Once that final approval is granted, the standard is given the status of an official W3C recommendation.

Standards didn't resolve the browser wars overnight. It took a while before browser manufacturers implemented the standards. Microsoft in particular espoused an "embrace and extend" philosophy in which they agreed to the standards but also continued to add on their own proprietary technologies in an attempt to make Internet Explorer a more attractive platform for web development. In the end, though, the demand for consistent behavior across all browsers won out, and standards provided the blueprint for the victory.

The Continuing Evolution of HTML

The HTML standard was initially maintained by the IETF, which published the HTML 2.0 standard in 1995 as RFC 1866.

Note "RFC" stands for "Request for Comments," which means the document was published and stakeholders were invited to comment on it as part of an ongoing review process.

The W3C took over the HTML standard in 1996. In 1997 the W3C published the HTML 3.2 standard. This version officially deprecated several vendor-specific features and further stabilized the standard for both browser manufacturers and content creators. In less than a year the W3C published HTML 4.0. This version of HTML moved the standard in the direction of purely semantic markup: many visual tags such as those that created bold or italic tags were deprecated in favor of using Cascading Style Sheets (CSS). The W3C published HTML 4.1 in 1999, which was essentially HTML 4.0 with some minor edits and corrections. In 2000 HTML 4.1 because an ISO standard: ISO/IEC 15445:2000.

All of these HTML versions were defined as SGML applications. Each tag along with its attributes was defined using SGML rules, as show in in Listing 1-1.

Listing 1-1. SGML Definition of the UL Tag in HTML 4.1

```
<!ELEMENT UL - - (LI)+                    -- unordered list -->
<!ATTLIST UL
  %attrs;                                 -- %coreattrs, %i18n,
%events -->
<!ELEMENT OL - - (LI)+                    -- ordered list -->
<!ATTLIST OL
  %attrs;                                 -- %coreattrs, %i18n,
%events -->
```

As the standards progressed, content creators had to follow them more and more strictly in order to guarantee consistent behavior across browsers.

THE RISE AND FALL OF XHTML

In 2008 a new SGML application was proposed that would provide a smaller and more manageable subset of SGML directives. Called the Extensible Markup Language, or XML, it was also meant to be used to define data markup languages. The HTML 4 standard was quickly translated into XML, resulting in XHTML. The XHTML 1.0 standard was published in 2000.

XHTML was meant to make the HTML language more modular and extensible.

XHTML syntax is stricter than plain HTML, and errors in XHTML markup will cause the rendering agent to publish an error and stop rather than revert to a base behavior and continue. XHTML was never widely adopted, however, because of lack of backward compatibility with older content and lack of browser support.

The Formation of the WHATWG and the Creation of HTML5

By 2004, the W3C was focusing its efforts on XHTML 2.0. However, some members of the consortium felt that the XML-based direction wasn't the correct path to follow for web technologies. The Mozilla Foundation and Opera Software presented a position paper to the W3C in June 2004. This paper focused on web applications as a whole: how to build them, what technologies they should employ, backward compatibility with existing web browsers, and so forth. The paper included a draft specification for Web Forms as an example of direction. You can read the paper on the W3C's web site at `www.w3.org/2004/04/webapps-cdf-ws/papers/opera.html`. The paper asked more questions than it answered, but overall it pointed in a different direction than the W3C's current XML-based solutions. In the end the W3C voted down the paper, opting to continue with XML solutions.

Many stakeholders felt very strongly about looking at web applications in the holistic fashion proposed by the paper, so a group was formed to focus on the creation of a web applications standard. Called the Web Hypertext Application Technology Working Group (WHATWG), members included individuals from Apple Inc., the Mozilla Foundation, and Opera Software. Initially they created a draft proposal for a Web Applications standard, which covered all of the features that the group felt was important for creating rich, interactive web applications, including:

- New semantic markup tags for common content patterns such as footers, sidebars, and pull quotes.
- New state managemen0074 and data storage features.
- Native drag-and-drop interactions.
- New network features such as server-pushed events.

This new standard was eventually merged with the Web Forms standard (also being worked on by the WHATWG) and the combined standard was renamed HTML5. This is why the HTML5 standard is not an SGML application, and why it covers so much more

than just markup: it was designed to provide better tools for creating web applications.

In 2007 the W3C's HTML group adopted the WHATWG's HTML5 specification and began moving forward with it. Both groups have continued to maintain their own versions of the same standard. By mutual agreement the W3C maintains the canonical standard for HTML5. The WHATWG's standard is considered a "living standard," which is therefore never complete and always evolving. In this way the W3C's standard is like a snapshot of the WHATWG's standard.

THE W3C HTML5 STANDARD

The W3C's HTML5 standard is available at `www.w3.org/TR/html5/Overview.html`. It is officially a W3C recommendation.

THE WHATWG LIVING STANDARD

The WHATWG HTML standard is located at `https://html.spec.whatwg.org/multipage/index.html`.

HTML5 Features

Because it was designed to enable the creation of rich interactive web applications, HTML5 specifies a lot more than just markup tags—though it covers those as well.

New Tags

The HTML5 standard specifies a host of new tags for marking up documents. New sectioning tags provide ways of indicating common design patterns such as footers and navigation components and providing improved semantic information for screen readers. New grouping tags offer ways to indicate groups of content such as figures. And of course, HTML5 includes the new audio and video tags, for embedding multimedia into web applications as easily as images. HTML5 also includes a whole set of new interactive elements for implementing common design patterns such as dialogs and progressive disclosure.

Since it includes the Web Forms specification, HTML5 also includes many new

form elements, including data lists (filterable dropdowns), meters and progress bars, and sliders. HTML5 also specifies several new form attributes to allow for richer interactions with forms. Now with simple attributes you can specify placeholder text in a form field, or indicate what form field should have focus (be active) when the page is loaded.

Canvas

HTML5 specifies the new `canvas` feature, a way to programmatically draw on a web page. The `canvas` also includes features for text, layer blending, and image manipulation.

JavaScript APIs

The HTML5 standard includes a set of new JavaScript APIs to add more features to web applications. There are new APIs for client/server communication, including the ability for servers to push events to web pages and new ways of securing communication across documents and domains. There are also features for storing data locally in the browser, drag-and-drop interactions, and multithreading.

Related Standards

There are a family of related standards that interact with HTML5, and are maintained by the W3C, but aren't technically members of the HTML5 standard. These include features like geolocation, device orientation, and WebGL.

Summary

In this chapter I covered the history of HTML and how HTML5 came about, including:

- the origins of the underlying technologies,
- the browser wars, and
- the birth of the standards.

I also covered the overall composition of the HTML5 standard and its relatives.

Enough history! The next chapter will dive into the new HTML5 elements, including the `audio` and `video` elements.

CHAPTER 2

HTML5 Elements

Though the HTML5 specification is much more complex than previous versions, like those versions it includes definitions of new elements and deprecations of old elements. In this chapter I'm going to focus on the Elements section of the HTML5 specification.

I will start by showing how best practices have contributed to the evolution of HTML. Then I will cover many of the new tags included in the HTML5 specification: tags for creating new sections, grouping content, semantic markup, embedded content, new interactive content, and forms. I'll also cover the new features of web forms: new form properties, field properties, and input types. Finally, I'll cover the elements that have been deprecated in HTML5.

Functionality, Semantics, and the Evolution of HTML

HTML5 represents the latest in the language's evolutionary line. In the beginning of the Web this evolution was largely driven by the browser manufacturers, who all wanted to create their own proprietary spaces on the Web to distinguish themselves from their competitors. Unfortunately, this led to the fracturing of the Web that is now known as the "Browser Wars."

The first standards were born to combat this fracturing. By providing a common ground for all browser manufacturers, they made it possible for developers to code HTML that was platform independent.

More important, as the standards evolved further, a set of best practices evolved along with them to help developers leverage the strengths of complying with the standards. Probably the two most important of these practices were the concepts of separation of functionality and semantic markup.

Separation of functionality dictates that we should use each of our tools according to their strengths. It is often summarized as "separation of presentation from content," but it goes deeper than that: use HTML for content, CSS for presentation, and JavaScript for functionality.

Decoupling HTML from CSS and JavaScript allows the three languages to evolve independently, and also makes it possible for developers to upgrade more easily or even completely change technologies later without having to totally redo the code for all three languages.

The core idea of *semantic markup* is to use the right tag to mark up a given section or piece of data as determined by its content. In a way it's a deeper application of separation of functionality: use the right tag for the job. Thus paragraphs should be marked up with `<p>` tags, unordered (bulleted) lists with `<ul>`, list items with `<li>`, and so forth. Today this best practice has become second nature to web developers, but it wasn't always that way. It was common to see tags used just for the indentation or margins that their default styles provided, which made for pretty confusing markup.

The problem with these best practices is they were of little help outside of building simple informational documents on the Web. If you wanted to do complex layouts, HTML didn't have the necessary semantics. And if you wanted to build functionality with web technologies, HTML really started to show its lack of semantics.

A major symptom of this lack was the proliferation of nonsemantic tags like `<div>` and `<span>`. Because these tags only denote sections (`<div>` denotes a block-level section and `<span>` denotes an inline section) they can safely be used to encompass just about any content. This suffusion of nonsemantic tags caused the coining of the neologism "div-itis" or, more commonly, "divitis."

One of the main purposes of HTML5 was to address these shortcomings. HTML5 specifies several new tags for different sections of documents, new semantic tags, and tags for improving interactivity and expanding the functionality of forms.

Sections

SUPPORT LEVEL
Good

All major browsers support section elements for at least the last two versions.

WHATWG Living Standard: `http://www.whatwg.org/specs/web-apps/current-work/multipage/sections.html#sections`

W3C Candidate Recommendation:
http://www.w3.org/TR/html5/sections.html#sections

HTML5 includes a set of new tags that are designed to address the lack of structural tags in previous versions of HTML. Marking up even moderately complex documents revealed several weaknesses in the original HTML tag set that resulted in the use of nonsemantic tags for many common purposes, such as navigation sections, document headers, and document footers.

The new tags are as follows:

- `<article>` An article is a complete, self-contained set of content within a page. Conceptually an article could be distributed or reused by itself. Examples of valid articles include a single magazine article within a larger magazine, a blog post, a reusable widget in the user interface, or any other self-contained set of content.
- `<aside>` An aside is a way of indicating a sidebar: a set of content that is independent of, and tangential to, the content that surrounds it. Examples include pull quotes, sidebars, or even advertising sections within larger documents.
- `<nav>` A nav section is the section with the major navigation links to other articles, or to other documents. It is not generally meant for collections of minor links, such as the links that are often relegated to a footer (in that specific case, the `<footer>` tag is considered semantically sufficient).
- `<footer>` This well-named tag represents the footer of the containing section element (`<body>`, `<article>`, etc.). Footers typically contain information about the containing section element like copyright information, contact information, and links to supporting documents and site maps.
- `<header>` The `<header>` tag groups together a set of introductory tags for the current containing section element (`<body>`, `<article>`, etc.). Headers can contain navigation, search forms, or even the document's table of contents and internal links.
- `<section>` The `<section>` tag is used to group thematically similar content together, often with a heading of some sort.

Before these tags were introduced, these sections were typically marked up using `<div>` tags with relevant CSS classes, as in Listing 2-1.

Listing 2-1. Old and Busted Markup with Nonsemantic Tag0073

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
    <style>
body {
  margin: 0;
  padding: 0;
}
.page {
  background-color: #C3DBE8;
}
.header {
  background-color: #DDDDDD;
}
.header li {
  display: inline-block;
  border: 1px solid black;
  border-radius: 5px;
  padding: 0 5px;
}
.footer {
  background-color: #DDDDDD;
}
    </style>
  </head>
  <body>
    <div class="page">
      <div class="header">
        <h1>Lorem Ipsum Dolor Sit Amet</h1>
        <div class="navigation">
          <ul>
            <li>one</li>
            <li>two</li>
            <li>three</li>
          </ul>
```

```
      </div>
    </div>
    <div class="section">
      <h2>Section Header</h2
      <p>Lorem ipsum dolor sit amet, consectetur
adipiscing elit.
        Proin congue leo ut nut tincidunt, sed hendrerit
justo
        tincidunt. Mauris vel dui luctus, blandit felis
sit amet,
        mollis enim. Nam tristique cursus urna, id
vestibulum
        tellus condimentum vulputate. Aenean ut lectus
adipiscing,
        molestie nibh vitae, dictum mauris. Donec lacinia
odio
        sit amet odio luctus, non ultrices dui rutrum.
Cras
        volutpat tellus at dolor rutrum, non ornare nisi
        consectetur. Pellentesque sit amet urna
convallis, auctor
        tortor pretium, dictum odio. Mauris aliquet odio
vel
        congue fringilla. Mauris pellentesque egestas
lorem.</p>
    </div>
    <div class="aside">
      <h2>Aside Header</h2>
      <p>Vivamus hendrerit nisl nec imperdiet bibendum.
Nullam
        imperdiet turpis vitae tortor laoreet ultrices.
Etiam
        vel dignissim orci, a faucibus dui. Pellentesque
        tincidunt neque sed sapien consequat dignissim.
</p>
    </div>
    <div class="footer">
      <div class="address">
        Sisko's Creole Kitchen, 127 Main Street,
        New Orleans LA 70112
      </div>
    </div>
```

```
    </div>
  </body>
</html>
```

Listing 2-1 divides your content into a single "page," contained within a `<div>` tag with the class `"page"` applied to it. Within this page you have a header with our navigation, a section, an aside, and a footer. You've also applied some basic styling to the markup to better illustrate the header and footer sections, and make the navigation elements look more like buttons than a simple unordered list.

This is the kind of markup that you're probably used to seeing, and other than the fact that it relies a great deal on nonsemantic `<div>` tags there's nothing wrong with it. With the new HTML5 tags, however, you can do away with all of those `<div>` tags and replace them instead with semantic tags, as you do in Listing 2-2.

Listing 2-2. New Hotness Markup with HTML5 Semantic Tags

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
    <style>
body {
  margin: 0;
  padding: 0;
}
.page, article {
  background-color: #C3DBE8;
}
.header, header {
  background-color: #DDDDDD;
}
.header li, header li {
  display: inline-block;
  border: 1px solid black;
  border-radius: 5px;
  padding: 0 5px;
}
.footer, footer {
  background-color: #DDDDDD;
}
    </style>
```

```
  </head>
  <body>
    <article>
      <header>
        <h1>Lorem Ipsum Dolor Sit Amet</h1>
        <nav>
          <ul>
            <li>one</li>
            <li>two</li>
            <li>three</li>
          </ul>
        </nav>>
      </header>
      <section>
        <h2>Section Header</h2>
        <p>Lorem ipsum dolor sit amet, consectetur
adipiscing elit.
          Proin congue leo ut nut tincidunt, sed hendrerit
justo
          tincidunt. Mauris vel dui luctus, blandit felis
sit amet,
          mollis enim. Nam tristique cursus urna, id
vestibulum
          tellus condimentum vulputate. Aenean ut lectus
adipiscing,
          molestie nibh vitae, dictum mauris. Donec lacinia
odio
          sit amet odio luctus, non ultrices dui rutrum.
Cras
          volutpat tellus at dolor rutrum, non ornare nisi
          consectetur. Pellentesque sit amet urna
convallis, auctor
          tortor pretium, dictum odio. Mauris aliquet odio
vel
          congue fringilla. Mauris pellentesque egestas
lorem.</p>
      </section>
      <aside>
        <h2>Aside Header</h2>
        <p>Vivamus hendrerit nisl nec imperdiet bibendum.
Nullam
          imperdiet turpis vitae tortor laoreet ultrices.
```

```
Etiam
        vel dignissim orci, a faucibus dui. Pellentesque
        tincidunt neque sed sapien consequat dignissim.
</p>
      </aside>
      <footer>
        <address>
          Sisko's Creole Kitchen, 127 Main Street,
          New Orleans LA 70112
        </address>
      </footer>
    </article>
  </body>
</html>
```

You have replaced all of the nonsemantic divs with their associated semantic HTML5 tags. You've also updated the style sheet so the new tags will share the same styles with the old classes that were applied to the `<div>` tags you removed.

Browsers do render the two examples slightly differently. The differences vary from browser to browser: Internet Explorer 10 has the least variation, with the only difference being that text contained within an `<address>` tag is automatically rendered in italics. With Chrome and Firefox, the differences are greater, as seen in Figure 2-1.

Figure 2-1. *Screenshots of Listing 2-1 rendered in Chrome (left) and Listing 2-2 rendered in Firefox (right)*

As you can see, the font size for `<h1>` tags within `<header>` tags is smaller in both browsers, and they both render text within `<address>` tags in italics, as does Internet Explorer. If you are migrating to the new semantic tags, be sure you take these differences into account.

Grouping

SUPPORT LEVEL
Good

All major browsers support `<figure>` and `<figcaption>` features for at least the last two versions. Internet Explorer does not support the `<main>` tag natively but other browsers do.

WHATWG Living Standard: `http://www.whatwg.org/specs/web-`

apps/current-work/multipage/grouping-content.html#grouping-content

W3C Candidate Recommendation: http://www.w3.org/TR/html5/grouping-content.html#grouping-content

---

HTML5 defines a few new tags for grouping content. These tags differ from the HTML5 Section tags in that they define a given group of data as a particular kind of data, while the Section tags provide structure for the document. The new tags are as follows:

- `<figure>` This tag is used to group together a set of content that is self-contained and independent from the main document flow, but is referenced from within the document flow. Examples of figures include illustrations, screenshots, and code snippets.
- `<figcaption>` This tag is used to provide a caption for a `<figure>` tag. Captions are optional.
- `<main>` The definition of the `<main>` tag differs between the W3C and the WHATWG specifications. According to the W3C the `<main>` tag should be used to group together the primary content of the document or application that has to do with the main subject or functionality. According to the WHATWG, the `<main>` tag has no intrinsic meaning and instead represents its contents. The rationale for this difference is explained in detail in Bug 21553 over on the W3C's bugbase: https://www.w3.org/Bugs/Public/show_bug.cgi?id=21553.

The new grouping tags are simple to use, as demonstrated in Listing 2-3.

Listing 2-3. Using the New HTML5 Grouping Tags

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
    <style>
figure figcaption {
  font-style: italic;
```

```
}

figure pre {
  line-height: 1.6em;
  font-size: 11px;
  padding: 1em 0.5em 0.0em 0.9em;
  border: 1px solid #bebab0;
  border-left: 11px solid #ccc;
  margin: 0.3 0 1.7em 0.3em;
  overflow: auto;
  max-height: 500px;
  position: relative;
  background: #faf8f0;
}
    </style>
  </head>
  <body>
    <main>
      <article>
        <h1>Main, Figure and Figcaption</h1>
        <h2>Best Things Ever</h2>
        <p>Vivamus hendrerit nisl nec imperdiet bibendum.
Nullam
        imperdiet turpis vitae tortor laoreet ultrices.
Etiam
        vel dignissim orci, a faucibus dui. Pellentesque
        tincidunt neque sed sapien consequat dignissim.
</p>
        <figure>
          <figcaption>Using Figure and Figcaption for Code
Samples</figcaption>
          <pre>
[sample code here]
          </pre>
        </figure>
        <p>More content about Main, Figure and
Figcaption...</p>
      </article>
    </main>
  </body>
</html>
```

This example uses `<main>` to indicate the main section of the example document, and `<figure>` and `<figcaption>` to define a code sample area. You have also applied some simple CSS styling to the code area and its caption, to make it stand out more from the rest of the document, as shown in Figure 2-2.

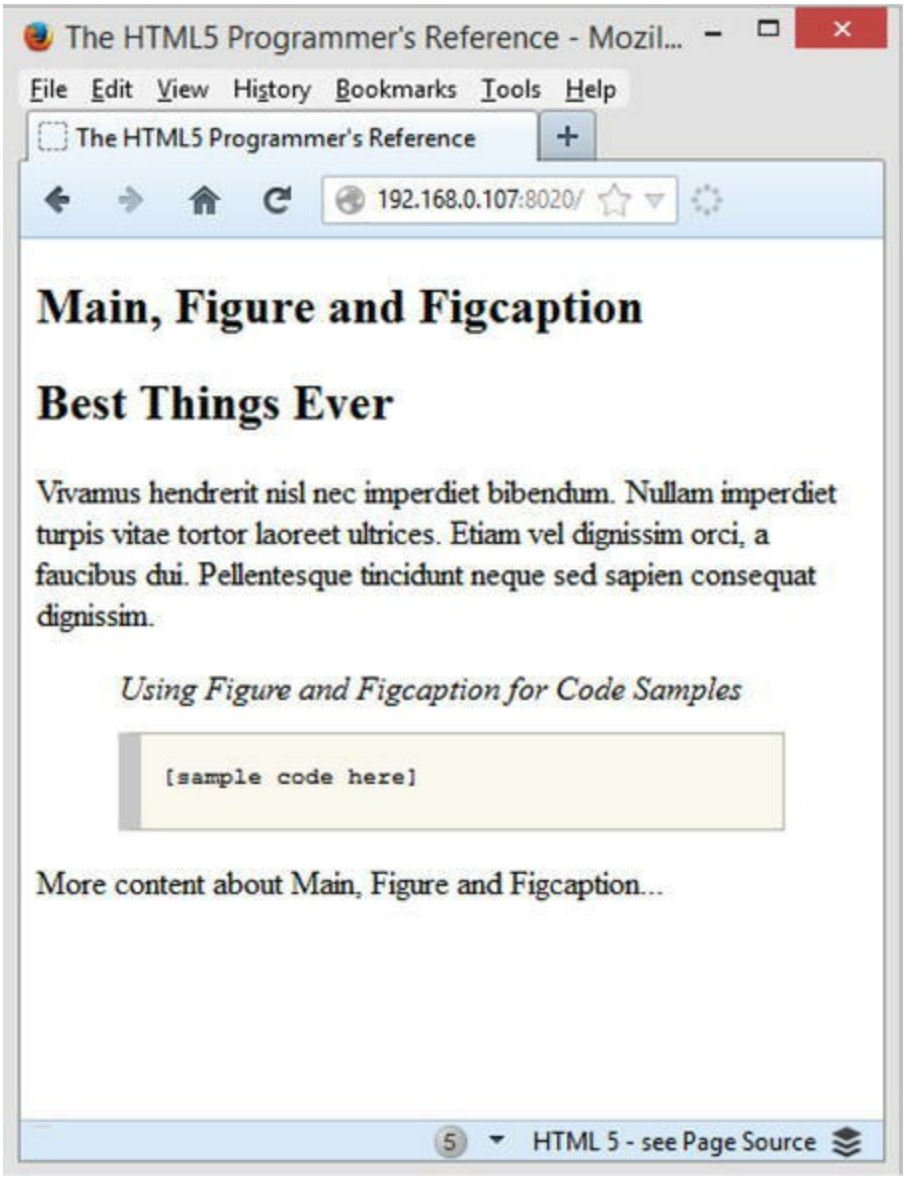


Figure 2-2. *Screenshot of Listing 2-3 rendered in Firefox*

In this screenshot you can see that the browser applies some default margins to the `<figure>` tag, which is fairly consistent across browsers.

Semantics

SUPPORT LEVEL

Mixed

There is little support in any browser for <bdi>, <data>, <ruby>, <rt>, <rp>, or <time>.

Support for <mark> is good (going back at least two versions of the major browsers) and <wbr> is excellent (going back to the very earliest versions of the major browsers).

WHATWG Living Standard: http://www.whatwg.org/specs/web-apps/current-work/multipage/text-level-semantics.html#text-level-semantics

W3C Candidate Recommendation: http://www.w3.org/TR/html5/text-level-semantics.html#text-level-semantics

---

HTML5 includes several new semantic tags designed to help clarify content types.

- <bdi> The Bi-Directional Isolation Element is used to isolate an inline span of text that might be rendered in a different direction than the surrounding text.
- <data> The <data> tag is used to associate machine-readable data with the content it encloses. It provides a semantic way of annotating content with data002E
- <mark> This tag is used to mark occurrences within a document, such as search results.
- <ruby>, <rp>, and <rt> These tags are for Ruby annotations, which are used for showing pronunciation of East Asian characters. For details about Ruby annotations, see http://www.w3.org/TR/ruby/ and http://en.wikipedia.org/wiki/Ruby_character.
- <time> The <time> tag is similar to <data> in that it provides a way to associate data (in this case, specifically date/time data) with the enclosed content.
- <wbr> The Word Break Opportunity tag is used to indicate a

position in the document flow where the browser may initiate a line break though its internal rules might not otherwise do so. It has no effect on bidi-ordering, and if the browser does initiate a break at the tag, a hyphen is not used.

Unfortunately support for these tags is rather poor. The `<data>` and `<time>` tags, along with the tags for Ruby annotations, are not widely supported, even in the most modern browsers.

The `<mark>` tag, though, is quite well supported, and is as easy to use as any other inline tag. Listing 2-4 shows a very simple use of the `<mark>` tag to highlight certain words within a document.

Listing 2-4. Marking Words in a Document

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
    <style>
mark {
  background-color: #E3DA5D;
}
    </style>
  </head>
  <body>
    <article>
      <h1>Using the &lt;mark&gt; tag</h1>
      <p>Vivamus hendrerit nisl nec imperdiet
<mark>bibendum</mark>. Nullam
       imperdiet turpis vitae tortor laoreet ultrices.
Etiam
       vel dignissim orci, a faucibus dui. Pellentesque
       tincidunt <mark>neque</mark> sed sapien consequat
dignissim.</p>
    </article>
  </body>
</html>
```

This example renders the same in all browsers (Figure 2-3).

Figure 2-3. *Screenshot of Listing 2-4 rendered in Firefox*

The `<wbr>` tag is probably one of the most broadly supported of all the HTML5 tags. It was a nonstandard tag available in all browsers that was brought into the standard with HTML5. It's used to provide word break suggestions in long words, which can be situationally useful. Listing 2-5 shows a simple example with long words before inserting `<wbr>` tags:

Listing 2-5. Long Words in a Document

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
    <style>
.larger {
  font-size: 2em;
}
    </style>
  </head>
  <body>
    <article>
      <h1>Using the &lt;wbr&gt; tag</h1>
      <p>Here are some long words in a slightly larger font
size to demonstrate
        how useful the &lt;wbr&gt; tag can be.</p>
      <p class="larger">Supercalifragilisticexpialidocious
and antidisestablishmentarianism,
        also pneumonoultramicroscopicsilicovolcanoconiosis.
</p>
    </article>
  </body>
</html>
```

As shown in Figure 2-4, Listing 2-5 renders as you would expect in all modern browsers.

Figure 2-4. *Rendering of Listing 2-5 in Firefox*

You can easily use some `<wbr>` tags to help the browser decide where to break those long words, as in Listing 2-6.

Listing 2-6. Suggesting Line Breaks in Large Words

```
<!DOCTYPE HTML>
```

```
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
    <style>
.larger {
  font-size: 2em;
}
    </style>
  </head>
  <body>
    <article>
      <h1>Using the &lt;wbr&gt; tag</h1>
      <p>Here are some long words in a slightly larger font
size to demonstrate
        how useful the &lt;wbr&gt; tag can be.</p>
      <p
class="larger">Supercali<wbr>fragilistic<wbr>expialidociou
and
        antidis<wbr>establishment<wbr>arianism.</p>
    </article>
  </body>
</html>
```

The browser can break the words at our suggestions if needed (Figure 2-5).

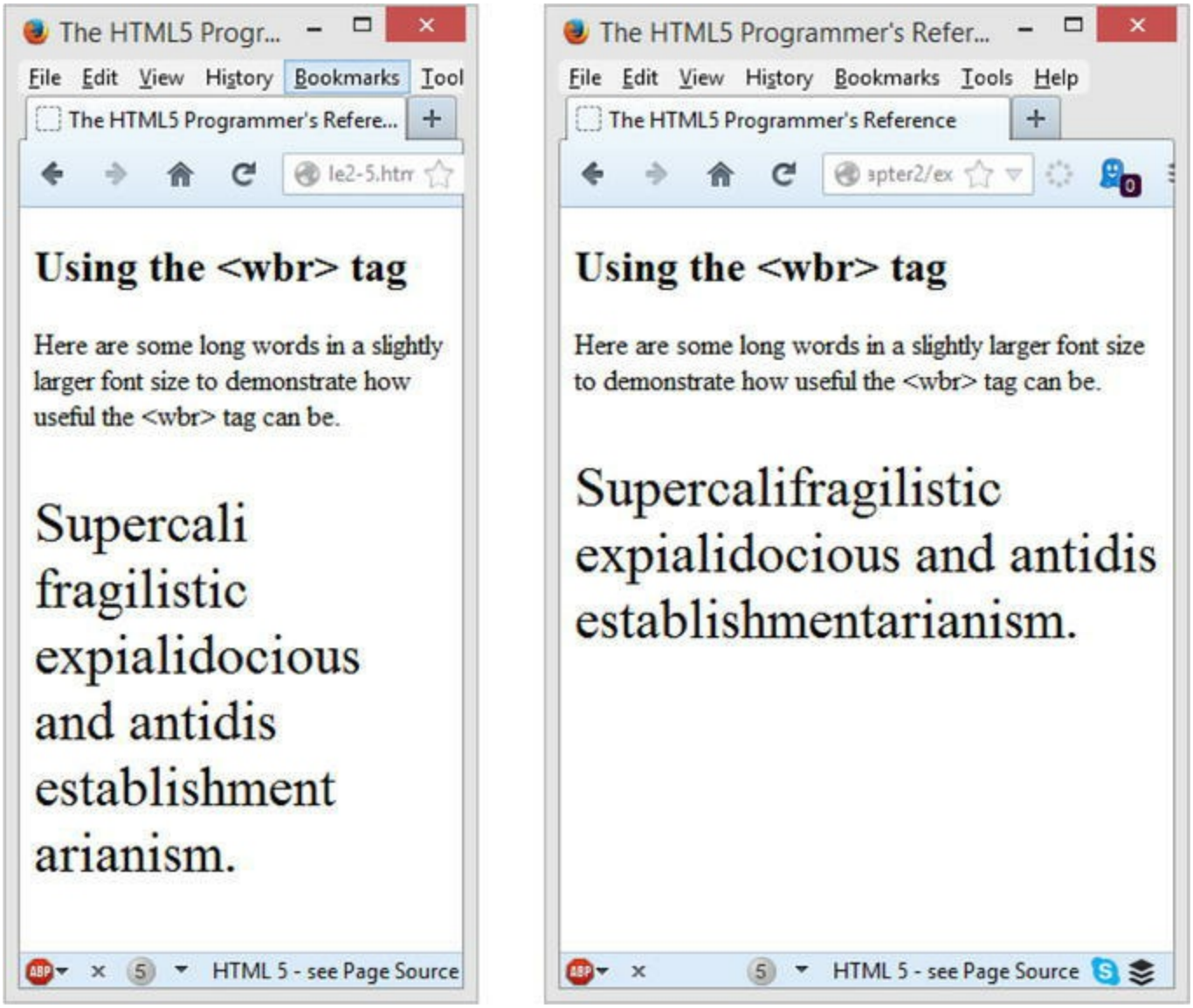


Figure 2-5. *Rendering Listing 2-6 at different browser widths*

As you can see, the browser can now use our word break suggestions if it needs to. This can be particularly useful if you are working on cramped layouts where screen real estate is at a premium, such as on a mobile device.

Audio and Video Content

SUPPORT LEVEL
Good

All modern browsers support audio and video elements for at least the last two versions, but see the following for information about format support.

WHATWG Living Standard: http://www.whatwg.org/specs/web-apps/current-work/multipage/edits.html#embedded-content

W3C Candidate Recommendation: www.w3.org/TR/html5/embedded-content-0.html#embedded-content-0

One of the biggest shortcomings of previous versions of HTML was their inability to easily include multimedia content on web pages. HTML5 has new tags that specifically address that problem. With these new tags, including multimedia content on a web page is as easy as including static images. Even better, all modern browsers support these capabilities very well.

Before HTML5, if you wanted to embed a video into your web page, you needed a third-party plug-in that had the ability to play the desired content, as well as do things like adjust the volume, fast-forward or reverse through the content, and so forth. With HTML5, browser manufacturers have built these capabilities into their software. These capabilities include a user interface for controlling playback and the ability to play various media formats for audio and video encoding.

An unfortunate complication is that audio and video can both be encoded in many different formats, and many of these formats have patent encumbrances that made the browser manufacturers unwilling to support them. So while all modern browsers support multimedia tags, some browsers support different formats than others. For details on what browser supports which formats, see Chapter 7.

Another complication arises from interacting with multimedia. For example, users will often want to skip around in content, going forward or back as desired. Supporting interactive functionality like that requires a server that is capable of reacting to these user interactions and can provide the portions of content as needed. Simple web servers typically don't have this capability, though many of them can be configured to do so. For more information on configuring servers for multimedia, see Chapter 7.

Embedded Audio Content

With the HTML5 `<audio>` tag, you can embed audio content into your web pages as easily as including an image. Like any HTML tag, the `<audio>` tag has several properties that you can set:

- `autoplay`: This is a boolean flag that, when set (to anything, even `false`), will cause the browser to immediately begin playing the audio content as soon as it can without stopping for buffering.
- `controls`: If this attribute is set, the browser will display its default user interface controls for the audio player (volume controls, progress meter/scrub bar, etc.).
- `loop`: If this attribute is set, the browser will loop playback of the specified file.
- `muted`: This attribute specifies that the playback should be muted by default.
- `preload`: This attribute is used to provide to the browser a hint for how to provide the best user experience for the specified content. It can take three values: `none`, `metadata`, and `auto`. The `none` value specifies that the author wants to minimize the download of the audio content, possibly because the content is optional, or because the server resources are limited. The `metadata` value specifies that the author recommends downloading the metadata for the audio content (duration, track list, tags, etc.) and possibly the first few frames of the content. The `auto` value specifies that the browser can put the user's needs first without risk to the server. This means the browser can begin buffering the content, download all the metadata, and so forth. Note that these values can be changed after the page has loaded. For example, if you have a page with many `<audio>` tags each with `preload` set to `none` to prevent swamping the server, when the user makes a choice of which of the `<audio>` tags they want to hear, you can dynamically change its `preload` value to `auto` to provide a better user experience. This enables you to balance user experience with available resources.
- `src`: This attribute specifies the source of the content, just as with an `<img>` tag. If desired, this attribute can be omitted in favor of one or more `<source>` tags contained within the `<audio>` tag.

The `<audio>` tag is not self-closing and thus requires a closing tag. Note that since older browsers do not support the `<audio>` tag, any content contained within one will be displayed in those browsers, thus providing a backward-compatible way of providing alternate content in older browsers.

The audio tag is very easy to use, as shown in Listing 2-7.

Listing 2-7. Embedding Audio Content in a Web Page

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
  </head>
  <body>
    <article>
      <h1>Using the &lt;audio&gt; tag</h1>
      <audio controls="controls" src="../media/windows-
rolled-down.mp3">
      </audio>
    </article>
  </body>
</html>
```

Note The examples in this section use audio files. You should substitute your own files as needed.

Listing 2-7 has opted to show the controls for the native player to demonstrate what they look like by default. Each browser's native player looks slightly different and has somewhat different features (Figure 2-6).

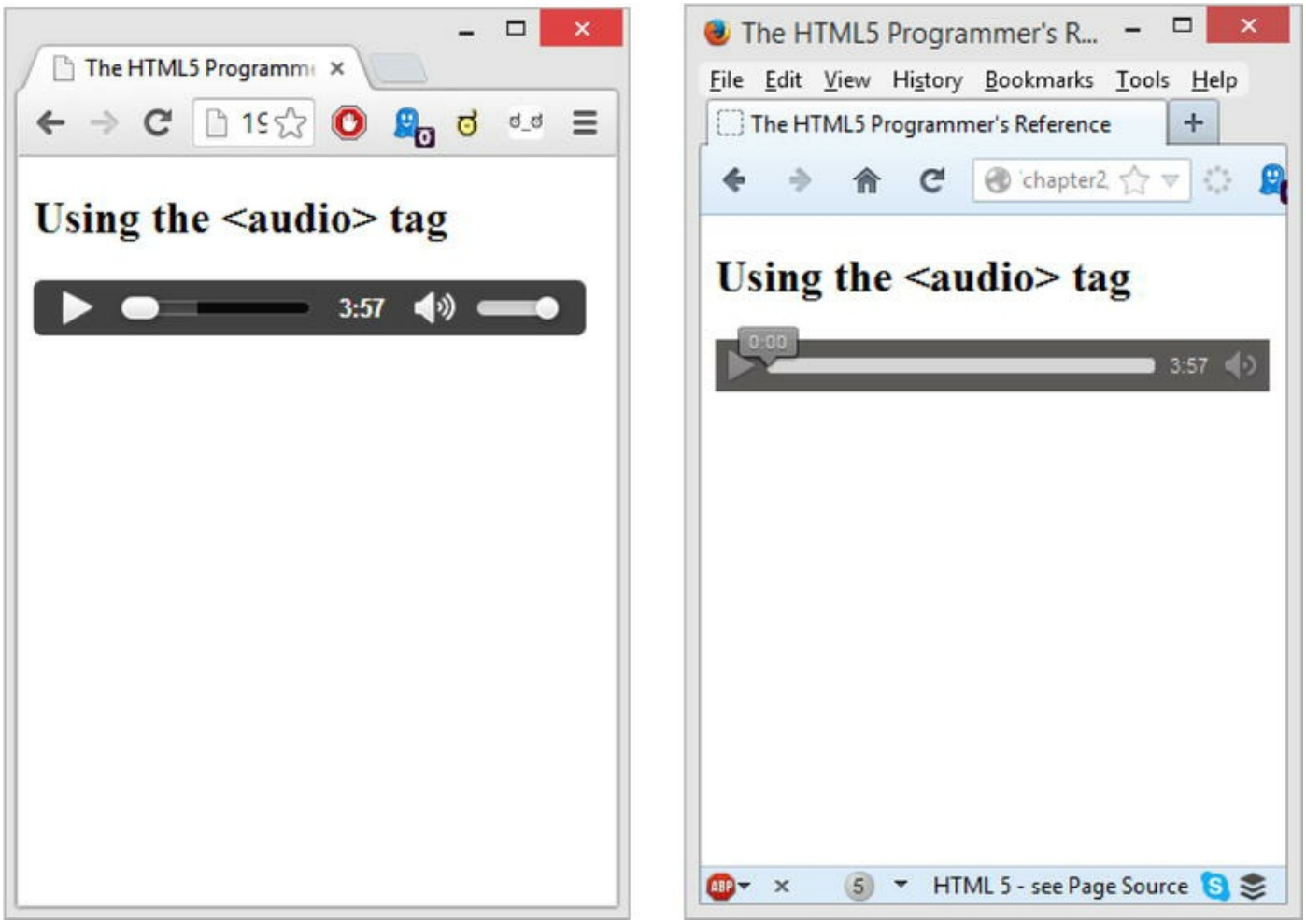


Figure 2-6. *Listing 2-7 rendered in Chrome (left) and Firefox (right)*

Chrome's default audio player has a volume slider, while Firefox's player has a tooltip indicating current play time. It is not possible to style the player, so if you want to have a consistent look and feel across all browsers you will have to build your own player—which is actually quite easy to do because HTML5 also specifies a JavaScript API for working with `<audio>` tags. For details on this API, please see Chapter 7.

Tip In Chrome, the audio and video players are implemented as web components using the new Shadow DOM specification. Using the Shadow DOM APIs it is possible to access and style the components of the player directly. For example, the background of the player is a shadow `<div>`, which can be selected with the CSS selector `audio::-webkit-media-controls-panel` and whose appearance (such as the background color) can be altered as desired. Similarly, the volume bar is an `<input type="range">` tag with the selector `audio::-webkit-media-controls-volume-slider`. Unfortunately, at the time of this writing Chrome and Opera are the only two browsers to support the Shadow DOM specification. Other

browsers may also be implementing their players using Shadow DOM, and when they fully support the specification, their players may become accessible as well, allowing web developers to control the appearance of the players without having to resort to building their own from scratch.

Embedded Video Content

The HTML5 `<video>` tag enables basic video capabilities in browsers. It functions similarly to the `<audio>` tag and has a similar set of properties that can be set:

- `autoplay`: This is a boolean flag that, when set (to anything, even `false`), will cause the browser to immediately begin playing the video content as soon as it can without stopping for buffering.
- `controls`: If this attribute is set, the browser will display its default user interface controls for the video player (volume controls, progress meter/scrub bar, etc.).
- `height`: This attribute can be used to specify the height, in pixels, of the video player.
- `loop`: If this attribute is set, the browser will loop playback of the specified file.
- `muted`: This attribute specifies that the playback should be muted by default.
- `poster`: This attribute can be used to specify a URL for a poster to display before the video is played. If no poster is specified, then the player will show the first frame of the video by default, once it has loaded.
- `preload`: This attribute is used to provide the browser a hint for how to provide the best user experience for the specified content. It can take three values: `none`, `metadata`, and `auto`. The `none` value specifies that the author wants to minimize the download of the video content, possibly because the content is optional, or because the server resources are limited. The `metadata` value specifies that the author recommends downloading the metadata for the video content (duration, track list, tags, etc.) and possibly the first few frames of the content. The `auto` value specifies that the browser can put the user's needs first without risk to the server.

This means the browser can begin buffering the content, download all the metadata, and so forth. Note that these values can be changed after the page has loaded. For example, if you have a page with many `<video>` tags, each with `preload` set to `none` to prevent swamping the server, when the user chooses which of the `<video>` tags they want to view, you can dynamically change its `preload` value to `auto` to provide a better user experience. This enables you to balance user experience with available resources.

- `src`: This attribute specifies the source of the content, just as with an `<img>` tag. If desired, this attribute can be omitted in favor of one or more `<source>` tags contained within the `<video>` tag.
- `width`: This attribute can be used to specify the width of the video player, in pixels.

The `<video>` tag is just as easy as the `<audio>` tag to use, as shown in Listing 2-8.

Listing 2-8. Embedding Video Content in a Web Page

```
<!DOCTYPE html>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
  </head>
  <body>
    <article>
      <h1>Using the &lt;video&gt; tag</h1>
      <video controls="controls"
src="../media/podcast.m4v">
      </video>
    </article>
  </body>
</html>
```

Note The examples in this section use video files. You should substitute your own files as needed.

And as with the `<audio>` tag, each browser provides a slightly different video

player, as shown in Figure 2-7.

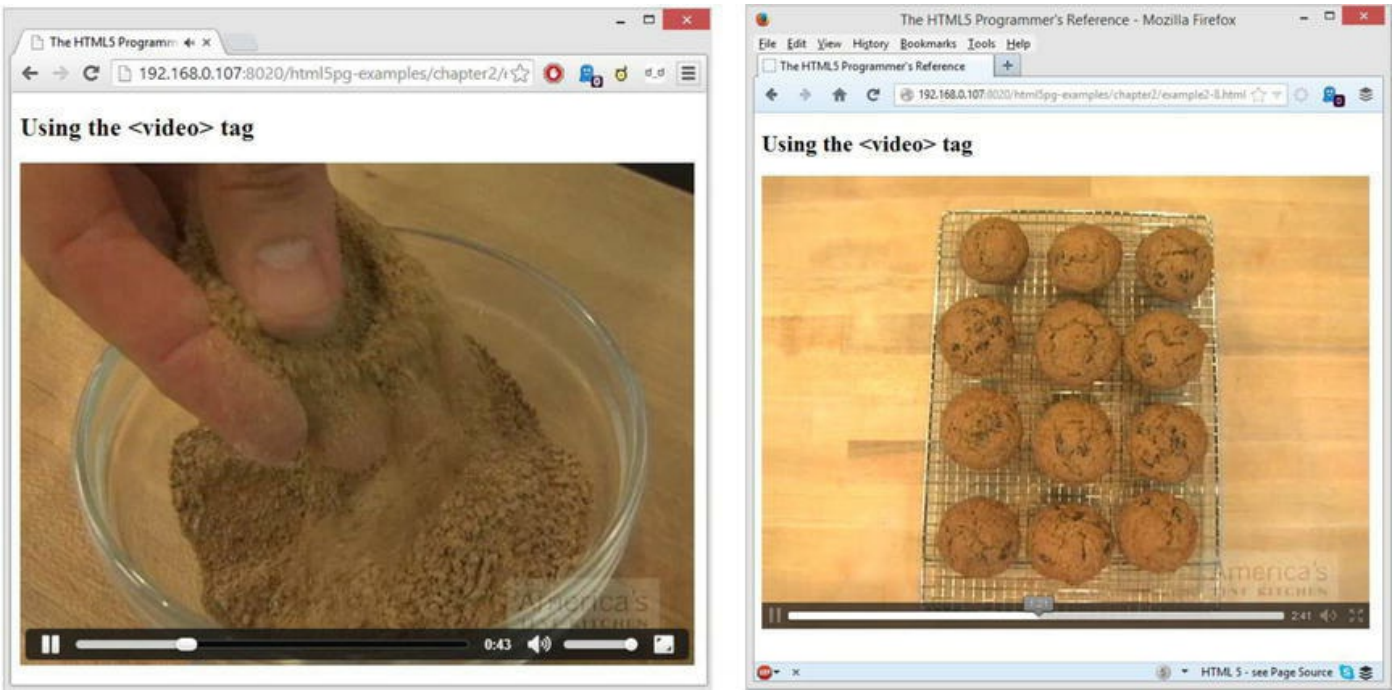


Figure 2-7. *Listing 2-8 rendered in Chrome (left) and Firefox (right)*

As before, the two browsers have slightly different interfaces for their video players. If you want to change the appearance of the player and its controls, you'll have to build your own.

Specifying Multiple Sources

Both the `<audio>` and `<video>` tags have an `src` attribute, but you can forgo that attribute in favor of including a list of one or more `<source>` tags inside of the `<audio>` or `<video>` tag. You can even specify different encodings of the same file, thus working around any limitations that browsers might have with encoding support. The browser will go down the list of `<source>` tags and play the first file that it supports.

The `<source>` tag has two attributes:

- `src`: The URL for the audio file.
- `type`: The MIME type of the audio file, with an optional codecs parameter, specified according to RFC 4281.

As an example, imagine you have a video that we want to serve. You have it in two

different formats: Ogg Vorbis and MP4. Use two `<source>` tags as shown in Listing 2-9.

Listing 2-9. Specifying Multiple Sources for Multimedia

```
<video controls>
  <source src="../media/video-1.mp4" type="video/mp4">
  <source src="../media/video-1.ogv" type="video/ogg">
</video>
```

You can get very precise about the encoding of your audio and video by using the optional codecs parameter in the type attribute. For example, if you have an H.264 video (profile 3) with low-complexity AAC audio all contained in an MP4 container, you could specify the codecs as shown in Listing 2-10:

Listing 2-10. Specifying Audio and Video Codecs for a Video Source

```
<source src="../media/video-1.mp4" type="video/mp4,
codecs=’ avc1.4D401E, mp4a.40.2’">
```

This can be particularly useful for providing the best possible quality encoding for your video while allowing the most browsers to access it regardless of encoding support limitations.

Interactive Elements

SUPPORT LEVEL
Unsupported

Interactive elements are not supported by modern browsers except for experimental versions.

WHATWG Living Standard: `http://www.whatwg.org/specs/web-apps/current-work/multipage/interactive-elements.html#interactive-elements`

W3C Candidate Recommendation: `http://www.w3.org/TR/html5/interactive-elements.html`

HTML5 includes a new set of interactive elements that are intended to provide some prebuilt user interface elements that can be used in web pages and applications. Unfortunately, these features are not yet supported in most browsers, but support probably will improve with time.

Dialogs

One of the most exciting new features is the `<dialog>` tag, which provides the ability to easily create pop-up dialogs. Any content enclosed in a `<dialog>` tag is not rendered in the document until you call one of its display methods:

- `show`: Calling this method will open the dialog as a standard pop-up.
- `showModal`: Calling this method will open the dialog as modal dialog, with the rest of the page grayed out behind the dialog.

In addition, each dialog will dispatch a `close` event when it is closed.

Listing 2-11 is a simple example that demonstrates how to use the `<dialog>` tag. As of this writing, the only browser that supports the `<dialog>` tag is Chrome, and even then you have to activate the Experimental Web Platform Features in `chrome://flags`. If you enable the features, the example will work great.

Listing 2-11. Web Dialogs

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
    <style>
li {
  display: inline-block;
  background-color: #A9DCF5;
  border-radius: 4px;
  padding: 2px 10px;
  cursor: pointer;
}
    </style>
  </head>
```

```
  <body>
    <article>
      <h1>Using the &lt;dialog&gt; tag</h1>
      <ul>
        <li id="open-dialog">Open Dialog</li>
        <li id="open-modal">Open Modal</li>
      </ul>
      <dialog id="dialog">
        <p>Hello World!</p>
        <button id="close-dialog">Okay</button>
      </dialog>
    </article>
    <script>
var myDialog = document.getElementById('dialog'),
    openDialog = document.getElementById('open-dialog'),
    openModal = document.getElementById('open-modal'),
    closeDialog = document.getElementById('close-dialog'),
    status = document.getElementById('status');

closeDialog.addEventListener('click', function(event) {
  myDialog.close();
}, false);

openDialog.addEventListener('click', function(event) {
  myDialog.show();
}, false);

openModal.addEventListener('click', function(event) {
  myDialog.showModal();
}, false);

myDialog.addEventListener('close', function(event) {
  alert('A close event was dispatched.');
}, false);
    </script>
  </body>
</html>
```

This will render a very simple dialog shown in Figure 2-8.

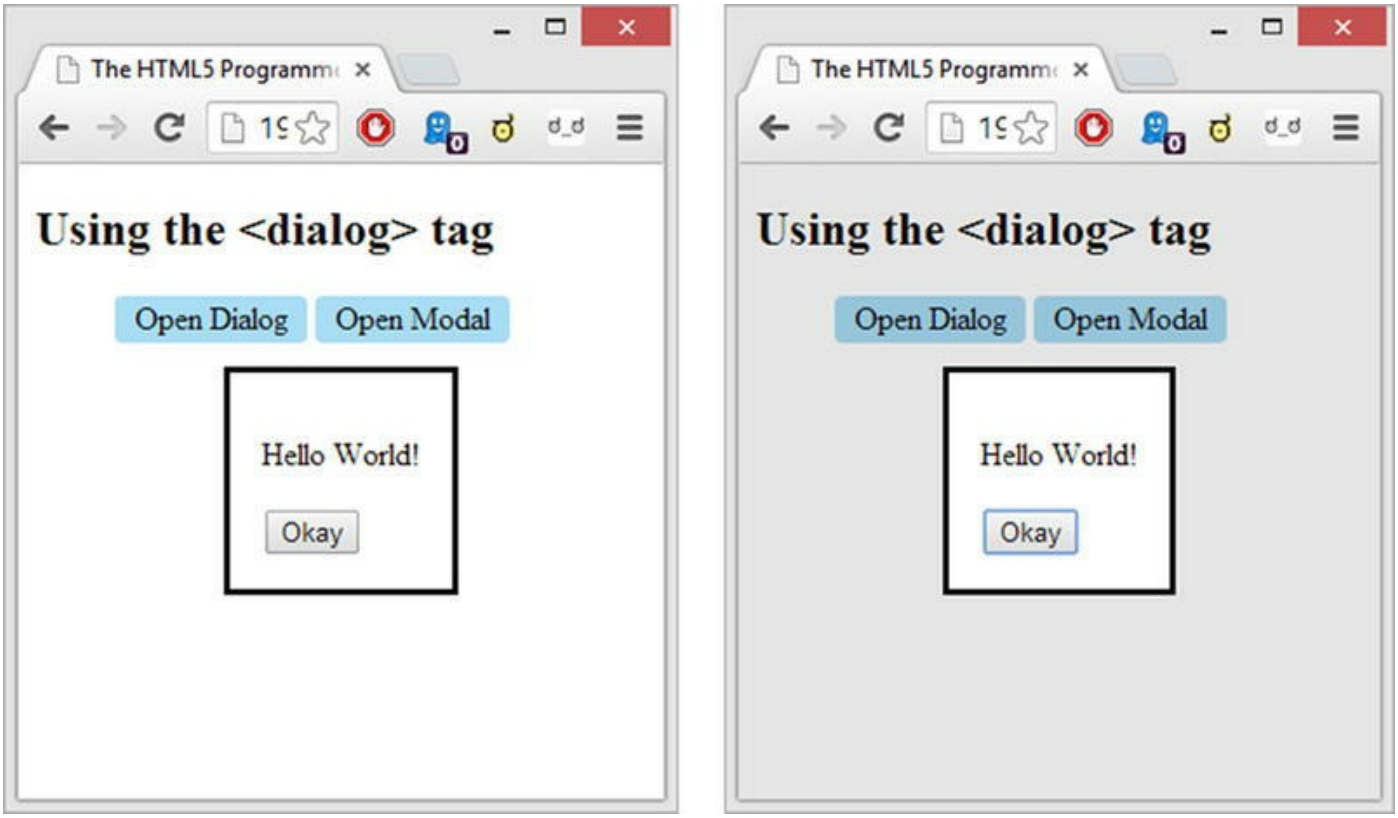


Figure 2-8. *The dialogs that Listing 2-11 produces*

Each time you close one of the dialogs, the close event will fire, producing an alert that reads, "A close event was dispatched."

This is the default appearance of the dialogs. They can easily be styled with CSS to make them more attractive, and of course they can contain any content, including images, form fields, and so forth. You can also style the backdrop for the modal instance; it is a pseudo-element that can be accessed using `::backdrop` on your dialog selector.

For example, in Listing 2-12, if you add a couple of simple CSS directives to your style sheet, you'll have a much more attractive dialog.

Listing 2-12. CSS Styles for Web Dialogs

```
dialog {
  text-align: center;
  padding: 1.5em;
  margin: 1em auto;
  border: 0;
  border-radius: 8px;
  box-shadow: 0 2px 10px #111;
}
```

```
dialog::backdrop {
  background-color: rgba(187, 217, 242, 0.8);
}
```

These styles will change the appearance of both the dialog and the modal backdrop (Figure 2-9).

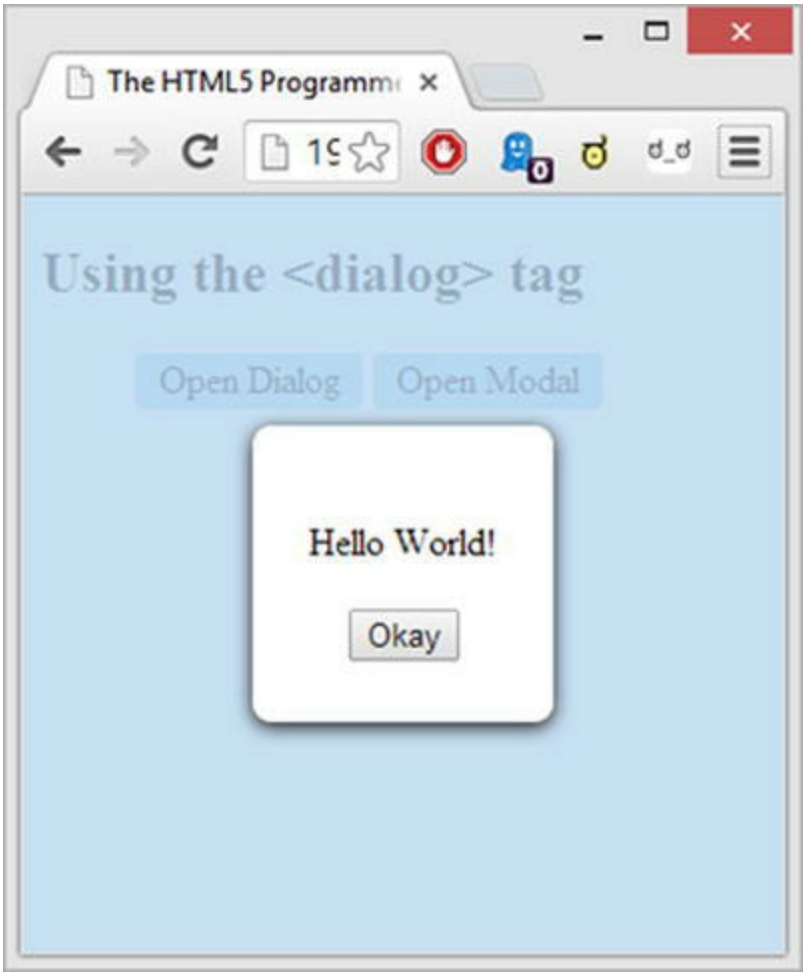


Figure 2-9. *Web dialog (modal state) with CSS styles applied*

Though the `<dialog>` tag is currently only supported in Chrome, there is a polyfill that provides most of the functionality in other browsers available at https://github.com/GoogleChrome/dialog-polyfill.

Tip *Polyfill* is a term for a library that enables or duplicates unsupported features in browsers. Another term for polyfill is *shim*.

Progressive Disclosure

One common UI feature is *progressive disclosure*: you provide a simple list of items, and when the user clicks on one, the space beneath expands, revealing more information. These widgets go by different names depending on the framework used (jQuery UI, for example, refers to them as *accordions*). HTML5 includes a definition for this feature using the `<summary>` and `<details>` tags. The `<details>` tag encloses all of the desired content, including a `<summary>` tag, which should enclose just a brief summary of the content. The default rendering of a `<details>` tag is to show just the contents of the `<summary>` tag preceded by a small triangle. The user can then click anywhere on the summary and the rest of the content will be revealed. You can specify that a given `<details>` tag is to be rendered in the open state by giving it the open attribute.

At the moment, Chrome, Opera, and Safari are the only browsers that support the `<details>` and `<summary>` tags. Firefox will be supporting the tags and you can check the status of their support in bug 591737 at https://bugzilla.mozilla.org/show_bug.cgi?id=591737. The status of support in Internet Explorer is unknown.

The tags are simple to use, as shown in Listing 2-13.

Listing 2-13. Progressive Disclosure

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
  </head>
  <body>
    <article>
      <h1>Using the &lt;summary&gt; and &lt;details&gt;
tags</h1>
      <details>
        <summary>Item 1</summary>
        <p>Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Phasellus
          accumsan orci nec justo rhoncus facilisis.
Integer pellentesque
          ipsum vitae semper lacinia. Quisque non nisl
rutrum, porta est at,
          ultrices neque. Aenean consequat, lacus vulputate
vestibulum
          faucibus, turpis magna mollis quam, a congue
```

```
neque lorem at
        justo.</p>
    </details>
    <details>
      <summary>Item 2</summary>
      <p>Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Phasellus
        accumsan orci nec justo rhoncus facilisis.
Integer pellentesque
        ipsum vitae semper lacinia. Quisque non nisl
rutrum, porta est at,
        ultrices neque. Aenean consequat, lacus vulputate
vestibulum
        faucibus, turpis magna mollis quam, a congue
neque lorem at
        justo.</p>
    </details>
    <details open>
      <summary>Item 3--this one will be open by
default</summary>
      <p>Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Phasellus
        accumsan orci nec justo rhoncus facilisis.
Integer pellentesque
        ipsum vitae semper lacinia. Quisque non nisl
rutrum, porta est at,
        ultrices neque. Aenean consequat, lacus vulputate
vestibulum
        faucibus, turpis magna mollis quam, a congue
neque lorem at
        justo.</p>
    </details>
    <details>
      <summary>Item 3</summary>
      <p>Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Phasellus
        accumsan orci nec justo rhoncus facilisis.
Integer pellentesque
        ipsum vitae semper lacinia. Quisque non nisl
rutrum, porta est at,
        ultrices neque. Aenean consequat, lacus vulputate
vestibulum
```

```
        faucibus, turpis magna mollis quam, a congue
neque lorem at
        justo.</p>
     </details>
   </article>
  </body>
</html>
```

In Chrome, this example renders nicely, as shown in Figure 2-10.

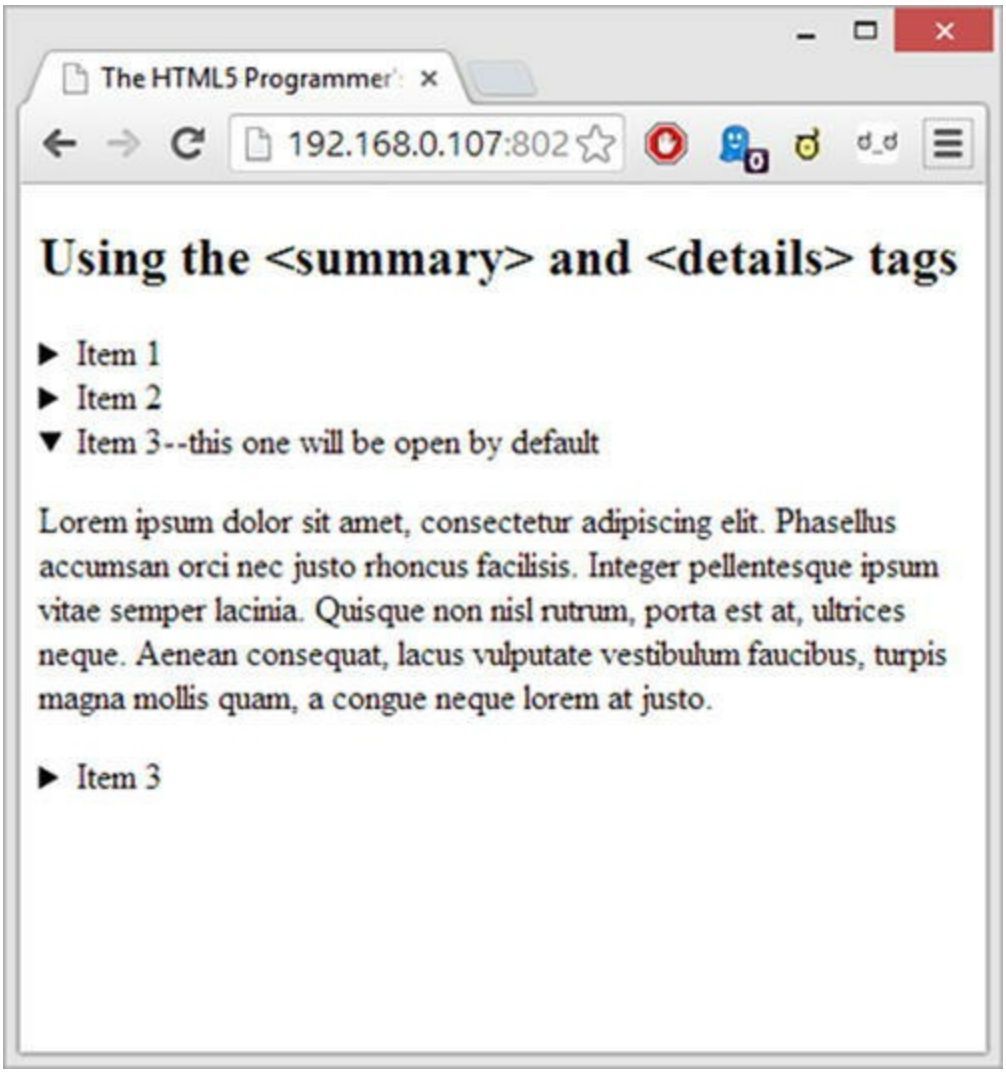


Figure 2-10. *Listing 2-13 rendered in Chrome*

Clicking on a closed item reveals its hidden content, and clicking on an open item hides its content.

Forms

Forms have been significantly improved in HTML5. The specification includes both new tags for forms (such as data lists, progress meters, and date pickers) as well as new attributes for existing form tags. These new features are designed to make forms more interactive for users and easier to build and maintain.

New Form Elements

SUPPORT LEVEL
Mixed

Most of these features are well supported in the major browsers for the last two versions. Internet Explorer does not support the `<meter>` tag, however.

WHATWG Living Standard: `http://www.whatwg.org/specs/web-apps/current-work/multipage/forms.html#forms`

W3C Candidate Recommendation: `http://www.w3.org/TR/html5/forms.html#forms`

HTML5 has a few new form elements that are specifically designed to implement common user interface patterns that have evolved over the last few years. Specifically these new tags implement autocomplete features and progress bars.

Data Lists

The first of the new tags implements a common autocomplete feature: when you begin typing into a form field, a drop-down appears that has a list of options that match what has already been typed. As you continue to type, the list becomes more specific, and at any time you can use the arrow keys to select one of the options. These sort of autocomplete fields are often referred to as *data lists* (and sometimes *combo boxes*) and HTML5 has a new `<datalist>` tag that implements this exact user interface element.

In practice, a `<datalist>` tags contain `<option>` tags, one for each item in the data list. By themselves `<datalist>` elements are not rendered in page, and can go

anywhere in the document structure. Once created, a data list must be associated with an input field in order to use it. Give the `<datalist>` tag a unique `id` attribute. To associate it with an `<input>` element, set that element's list attribute to the unique `id`. That tells the browser to render the specified data list with the `<input>` element, as demonstrated in Listing 2-14.

Listing 2-14. A Data List

```
<!DOCTYPE html>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
  </head>
  <body>
    <!-- Note the datalist can be anywhere -->
    <datalist id="browsers">
      <option value="Chrome">
      <option value="Firefox">
      <option value="Internet Explorer">
      <option value="Opera">
      <option value="Safari">
    </datalist>
    <article>
      <h1>Using the &lt;datalist&gt; tag</h1>
      <input list="browsers" />
    </article>
  </body>
</html>
```

As with other HTML5 user interface elements, each browser renders the data list slightly differently (Figure 2-11).

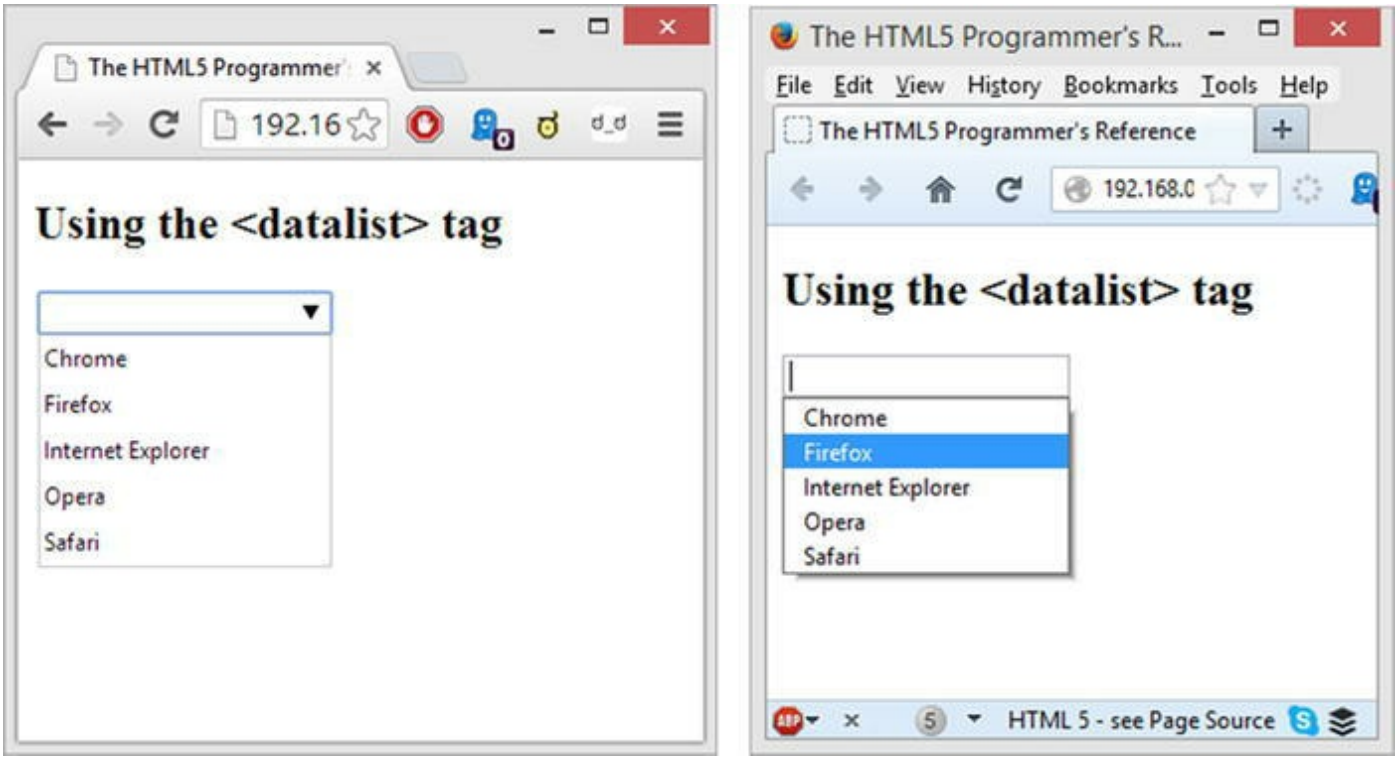


Figure 2-11. *Listing 2-14 rendered in Chrome (left) and Firefox (right)*

As you can see, Chrome provides a drop-down arrow hint on the right side of the input field to indicate that the input field is a data list, and Firefox has a slight drop shadow on the dropdown. The functionality of the list remains the same between browsers.

Meter

The new `<meter>` tag provides a simple meter bar or gauge visual element. This bar is meant to model a measurement within a known range, or a fractional value of a whole (e.g., volume, disk usage, etc.). It should not be used to show progress (e.g., in a download); use the new `<progress>` tag for this.

The `<meter>` tag has the following properties:

- `value`: The current value to be displayed. This value must be within the min and max values, if specified. If no value is set, or if it is malformed, the browser will default to 0. If specified but the value is greater than the max attribute, the value will be set to the value of the max attribute. If the value is less than the min attribute, the value will be set to the value of the min attribute.
- `min`: The minimum value of the range. Defaults to 0 if not

specified.

- `max`: The maximum value of the range. Must be greater than the value of the min attribute (if specified). Defaults to 1.

It is also possible to specify subranges within the measured range. There can be a `low` range, a `high` range, and an `optimum` range. The `low` range goes from the `min` value to a specified value, while the `high` range goes from the `high` value to the `max` value. Either the `low` range or the `high` range can be specified as an `optimum` range by specifying a number within them using the `optimum` attribute.

- `low`: The highest value of the low range. When the value attribute is within the `low` range, the bar will render yellow by default.
- `high`: The lowest value of the `high` range, which ranges from this value to the value of the `max` attribute. When the `value` attribute is within the `high` range, the bar will render yellow by default.
- `optimum`: Indicates an optimum value for the range. The value must be between the `min` and `max` values of the range. If the `low` and `high` ranges are used, specifying an `optimum` value within one of them will indicate which of those ranges is preferred. When the value is within the preferred range, the bar will render green. When it is in the other range, it will render red.

Creating these meters is as simple as adding a `<meter>` tag to your document, as demonstrated in Listing 2-15.

Listing 2-15. Meter Bars

```
<!DOCTYPE html>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
  </head>
  <body>
    <article>
      <h1>Using the &lt;meter&gt; tag</h1>
      <p>Simple meter from 1 to 100, value set to 25:<br>
        <meter min="1" max="100" value="25"></meter>
      </p>
      <p>Simple meter from 1 to 100, low range from 1 to
```

```
25, high range from
        75 to 100, value set to 90:<br>
       <meter min="1" max="100" low="25" high="75"
value="90"></meter>
     </p>
     <p>Simple meter from 1 to 100, low range from 1 to
25, high range from
        75 to 100, value set to 10:<br>
       <meter min="1" max="100" low="25" high="75"
value="10"></meter>
     </p>
     <p>Simple meter from 1 to 100, low range from 1 to
25, high range from
        75 to 100, optimum set to 10, value set to 10:<br>
       <meter min="1" max="100" low="25" high="75"
optimum="10" value="10"></meter>
     </p>
     <p>Simple meter from 1 to 100, low range from 1 to
25, high range from
        75 to 100, optimum set to 10, value set to 10:<br>
       <meter min="1" max="100" low="25" high="75"
optimum="10" value="90"></meter>
     </p>
   </article>
  </body>
</html>
```

The meters render pretty consistently across browsers (Figure 2-12).

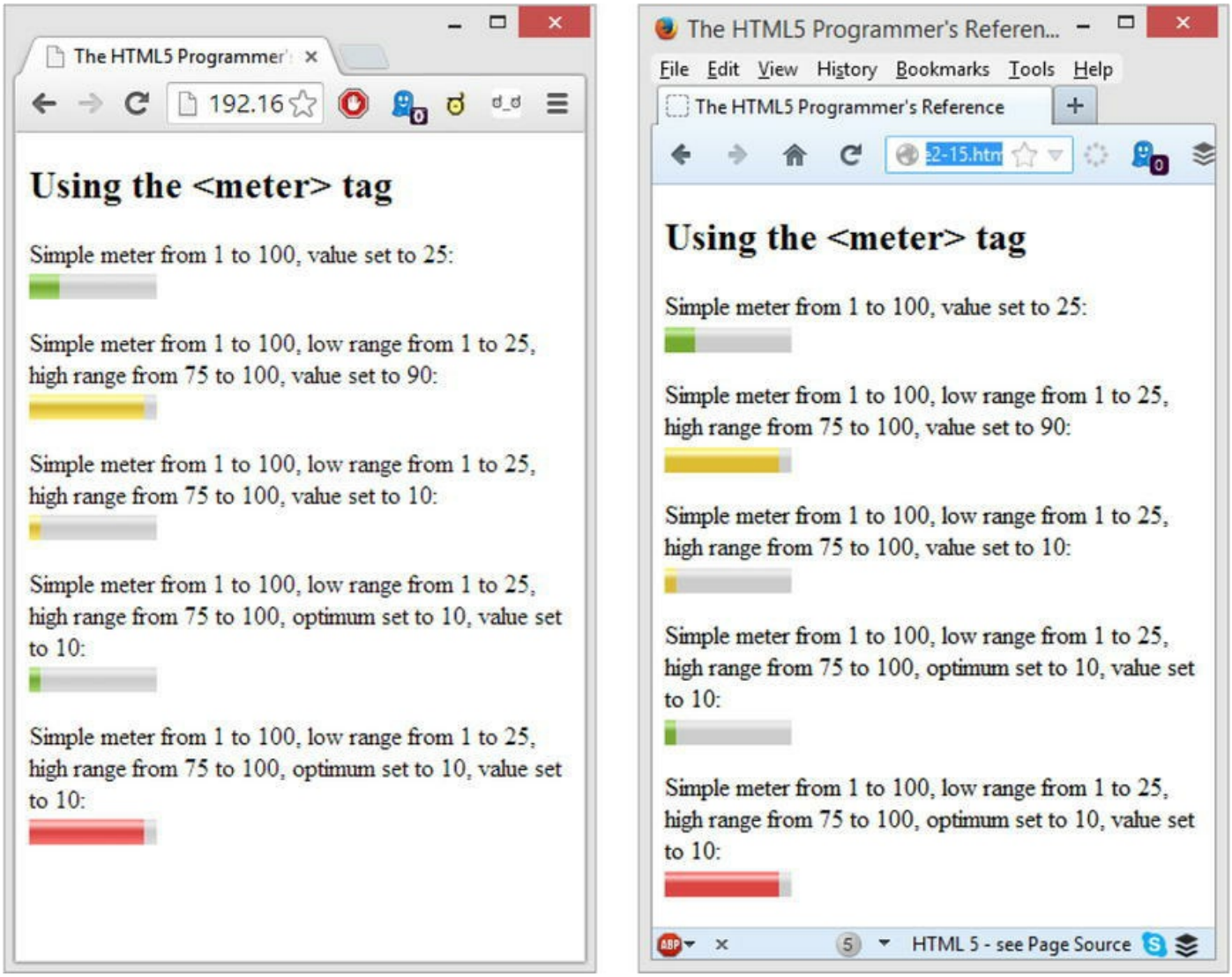


Figure 2-12. *Listing 2-15 rendered in Chrome (left) and Firefox (right)*

Output

The new `<output>` tag provides a way of specifying the output of a calculation or other user action within a form. It doesn't have any special features; instead it provides a semantic tag for marking up this kind of content.

A simple example is shown in Listing 2-16.

Listing 2-16. Calculation Output in a Form

```
<!DOCTYPE html>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
```

```
  </head>
  <body>
    <article>
      <h1>Using the &lt;output&gt; tag</h1>
      <form id="mainform" onsubmit="return false">
        <label for="input-number">Temperature</label>
        <input name="input-number" id="input-number"
type="number" step="any"><br>
        <input type="radio" name="convert-choice"
id="radio-ftoc" checked value="ftoc">
        <label for="radio-ftoc">Convert Fahrenheit to
Celcius</label><br>
        <input type="radio" name="convert-choice"
id="radio-ctof" value="ctof">
        <label for="radio-ctof">Convert Celcius to
Fahrenheit</label><br>
        Result:
        <output name="output-target" for="input-number"
id="output-target"></output>
      </form>
    </article>
    <script>
var myForm = document.getElementById('mainform');
var converter = {
  ctof: function(degreesC) {
    return (((degreesC * 9) / 5) + 32);
  },
  ftoc: function(degreesF) {
    return (((degreesF - 32) * 5) / 9);
  }
};
myForm.addEventListener('input', function() {
  var inputNumber = document.getElementById('input-
number'),
      outputTarget = document.getElementById('output-
target');
  var sel = document.querySelector('input[name=convert-
choice]:checked').value;
  outputTarget.value = converter[sel]
(parseInt(inputNumber.value));
}, false);
    </script>
```

```
  </body>
</html>.
```

This is a simple example, but you've used a couple of nifty tricks.

- You created a converter object, which has two methods, `ctof` (for converting Celsius to Fahrenheit) and `ftoc` (for converting Fahrenheit to Celsius).
- You set one of the radio button's value properties to `ctof`, and the other to `ftoc`.
- You used the selector `input[name=convert-choice]:checked` to get whichever radio button is checked and then fetch its value (either "ctof" or "ftoc").
- Then you can directly access the correct method on the converter object just by using the result of your query.

Tip JavaScript is also governed by a standard—ECMA-262—which specifically defines two ways to access object members: dot notation or bracket notation. So `objectName.identifierName` is functionally equivalent to `objectName[<identifierName string>]` even if the object in question is not an array. For details, see Section 11.2.1, "Property Accessors," in ECMA-262 at http://www.ecma-international.org/ecma-262/5.1/.

Figure 2-13 shows Listing 2-16 as rendered in Chrome.

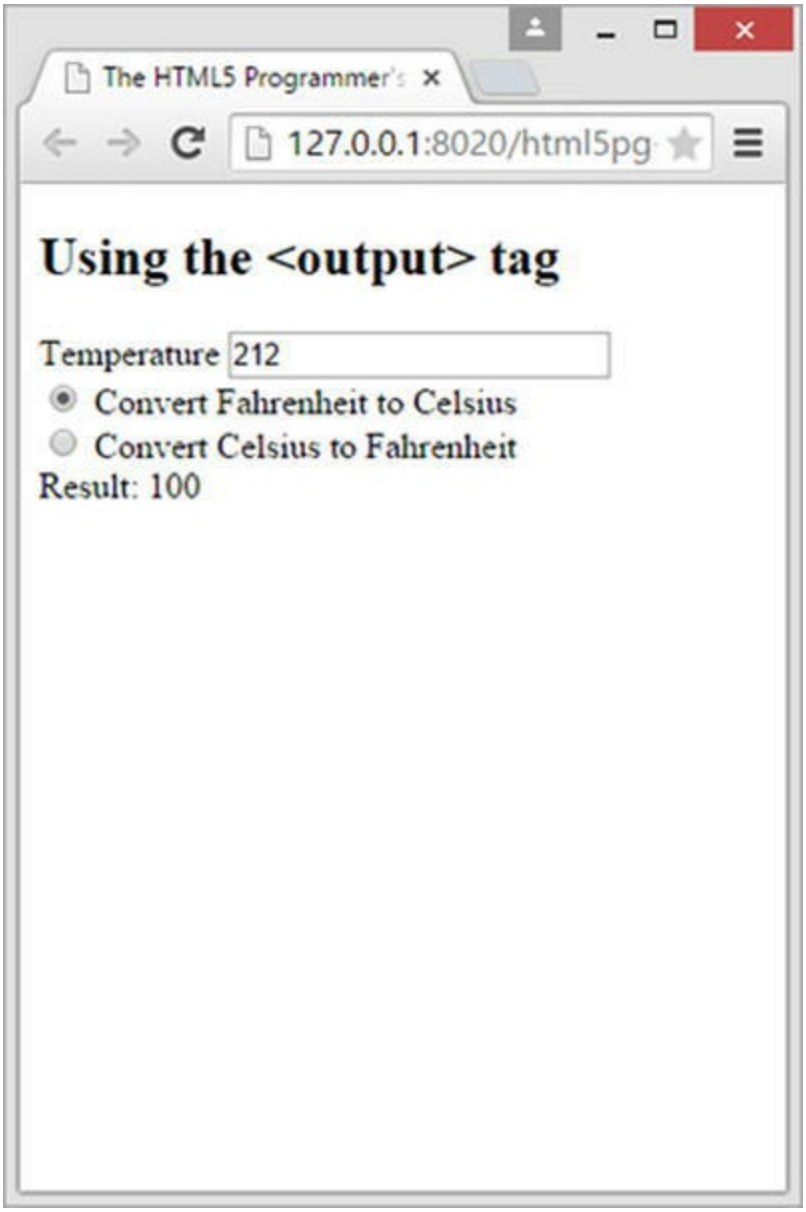


Figure 2-13. *Listing 2-16 rendered in Chrome*

Progress

HTML5 defines a new `<progress>` tag, which renders as a progress meter in the document. It is used to indicate progression or completion of a task, and provides the user with an idea of how much has been done and what still remains. It should not be used for visualizing a measurement within a known range—for that, use the `<meter>` tag.

The `<progress>` tag takes the following attributes:

- `max`: The maximum value of the activity. This value must be a valid positive floating-point number. If `max` is not specified, the

maximum value defaults to 1.

- `value`: The current value of the progress. This value must be a valid floating-point number between 0 and `max` (if specified) or 1 (if `max` is not specified). If `value` is not specified, then the progress bar is considered indeterminate, meaning the activity it is modeling is ongoing but gives no indication of how much longer it will take to complete.

Listing 2-17 provides a simple demonstration of progress bars.

Listing 2-17. Progress Bars

```
<!DOCTYPE html>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
  </head>
  <body>
    <article>
      <h1>Using the &lt;progress&gt; tag</h1>
      <p>Downloading file1<br>
        <progress max="100" value="10">10/100</progress>
10%</p>
      <p>Downloading file2<br>
        <progress max="100" value="50"
orient="vertical">50/100</progress> 50%</p>
    </article>
  </body>
</html>
```

As shown in Figure 2-14, the progress bar renders differently in the various browsers.

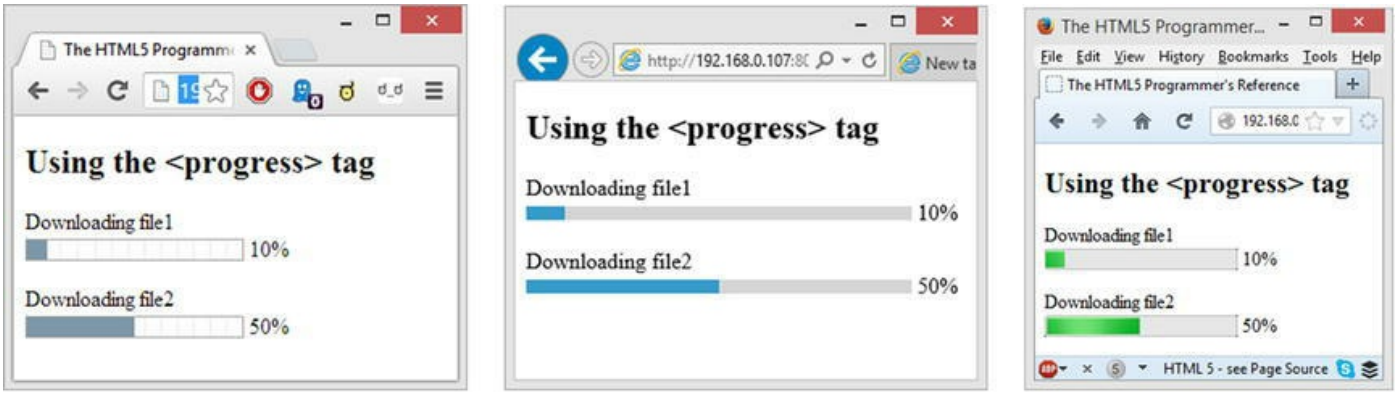


Figure 2-14. *Listing 2-17 rendered in Chrome (left), Internet Explorer (middle), and Firefox on Windows 8 (right)*

The examples would look different when rendered on MacOS as well. Fortunately, the bars are easy to style. Firefox and Internet Explorer give direct access to the element's styling, while in Chrome you have to select the pseudo-elements to change them. By adding a few simple directives to your CSS, as shown in Listing 2-18, you can make the bar look the same in all browsers.

Listing 2-18. CSS Rules for Progress Bars

```
progress {
  color: #0063a6;
  font-size: .6em;
  line-height: 1.5em;
  text-indent: .5em;
  width: 15em;
  height: 1.8em;
  border: 1px solid #0063a6;
  background-color: #fff;
}
::-webkit-progress-bar {
  background-color: #fff;
}
::-webkit-progress-value {
  background-color: #0063a6;
}
```

As you can see in Figure 2-15 the bars now render the same across browsers.

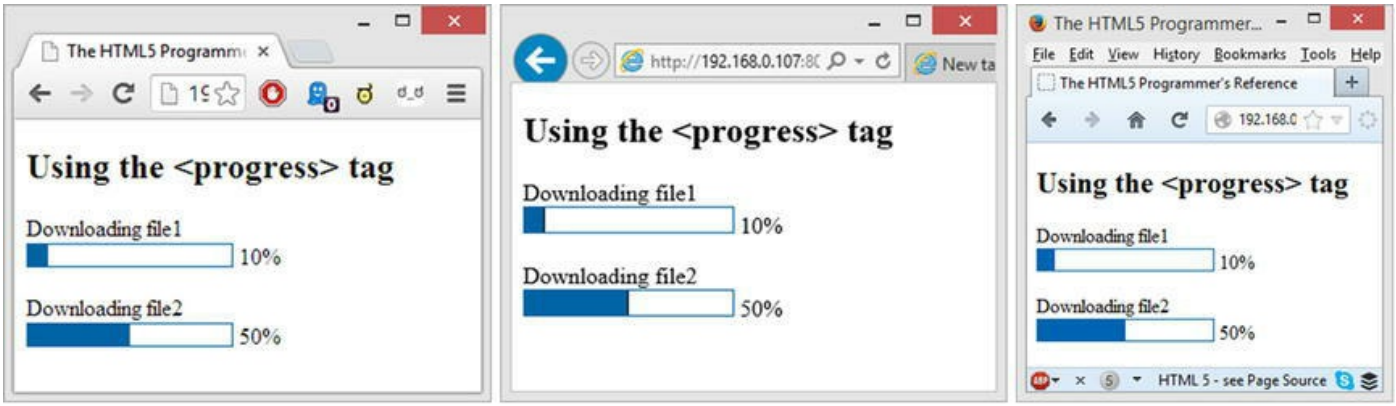


Figure 2-15. *Progress bars with CSS rules applied*

A slightly more practical example would be a timer. Using the `<progress>` tag you can indicate that some allotted time—ten seconds, for example—is passing, as shown in Listing 2-19.

Listing 2-19. A Ten-Second Timer

```
<!DOCTYPE html>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
    <style>
progress {
  color: #0063a6;
  font-size: .6em;
  line-height: 1.5em;
  text-indent: .5em;
  width: 15em;
  height: 1.8em;
  border: 1px solid #0063a6;
  background-color: #fff;
}
::-webkit-progress-bar {
  background-color: #fff;
}
::-webkit-progress-value {
  background-color: #0063a6;
}
    </style>
  </head>
  <body>
```

```
    <article>
      <h1>Using the &lt;progress&gt; tag</h1>
      <h2>Ten Second Timer</h2>
      <p><progress max="10" value="0"
id="myProgress">0</progress></p>
    </article>
    <script>
var progress = 0;
var myProgress = document.getElementById("myProgress");
var myTimer = setInterval(function() {
  myProgress.value = ++progress;
  if (progress > 10) {
    clearInterval(myTimer);
  }
}, 1000);
    </script>
  </body>
</html>
```

This example uses the DOM method `setInterval()` to run a function every second that updates the value of the progress bar. When the progress bar is full, it cancels the timer with the `clearInterval()` method.

New Form Element Attributes

The HTML5 specification includes a few useful new attributes for form elements. Again, these new attributes were designed specifically to address shortcomings in previous versions of HTML forms and to add commonly needed functionality that, until now, had to be built using JavaScript.

Autocomplete

All browsers offer the capability of storing form data for later reuse. This is of particular help with mobile devices because it reduces typing. The `autocomplete` attribute allows you to specify which `<input>` elements can be autocompleted and which should always be filled in manually. The `autocomplete` attribute can take two values: `on` (autocomplete is allowed; this is the default) or `off` (autocomplete is not allowed).

Listing 2-20 is a simple example with two form fields, one with autocomplete

allowed and the other with it disallowed.

Listing 2-20. Controlling Autocomplete in Forms

```
<!DOCTYPE html>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
  </head>
  <body>
    <article>
      <h1>Using the autocomplete attribute</h1>
      <form id="test-form" action="#" method="post">
        <p><label for="input-auto">This input allows
autocomplete:</label><br>
        <input autocomplete="on" id="input-auto"
name="input-auto"></p>
        <p><label for="input-noauto">This input does not
allow autocomplete:</label><br>
        <input autocomplete="off" id="input-noauto"
name="input-noauto"></p>
        <p><input type="submit"></p>
      </form>
    </article>
  </body>
</html>
```

In just about every browser, you should be able to fill out the form fields and click the submit button. Then, reload the page. Double-click in the first form field, and you should be presented with a drop-down containing the value you entered earlier (Figure 2-16).

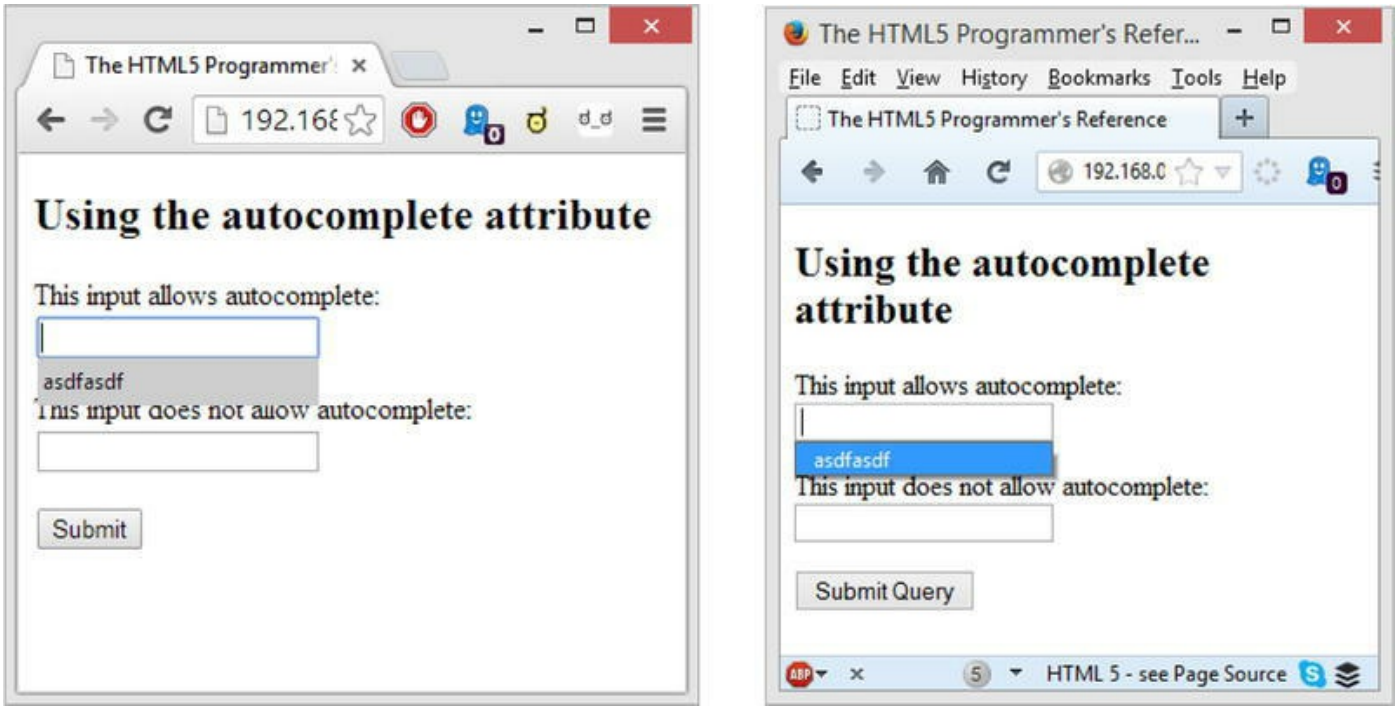


Figure 2-16. *Listing 2-20 rendered in Chrome (left) and Firefox (right)*

Note that you will have to enable the autofill feature in your browser. Most browsers will enable it by default, but many people turn it off for security purposes. If the user has disabled the feature in their browser, the autocomplete attribute will have no effect.

Browsers use a number of cues to determine which form fields should be autocompleted with what data: the name and ID for the fields, the `action` and `method` attributes of the `<form>` tag, and so forth. The process is fairly nonstandard and edges into the realm of "magic." In 2012, Google proposed an extension to the `autocomplete` property to help standardize the process. In this proposal they suggested an `autocompletetype` attribute with an extensive set of values ranging from `address-line1` to `postal-code` to `url`. You can read their full proposal at `http://wiki.whatwg.org/wiki/Autocompletetype`. That proposal was never fully adopted, but sections from it eventually went into the new Autofill specification, which you can view at `https://html.spec.whatwg.org/multipage/forms.html#autofill`

Autofocus

The `autofocus` attribute allows you to specify what form field should have focus when the page loads. Because it is exclusive, you can only set `autofocus` on one form field on a given page, and the focus will go to that element when the page is done

loading. You cannot set `autofocus` on a form element of type `hidden`. Autofocus can be set to any `<input>`, `<button>`, or `<textarea>` field.

If you add an `autofocus` attribute to the second field in Listing 2-20 on page load it will be focused, as in Listing 2-21.

Listing 2-21. Automatically Focusing an Input Field

```
<!DOCTYPE html>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
  </head>
  <body>
    <article>
      <h1>Using the autofocus attribute</h1>
      <form id="test-form" action="#" method="post">
        <p><label for="input-auto">This input allows
autocomplete:</label><br>
        <input autocomplete="on" id="input-auto"
name="input-auto"></p>
        <p><label for="input-noauto">This input does not
allow autocomplete:</label><br>
        <input autocomplete="off" id="input-noauto"
name="input-noauto" autofocus="autofocus"></p>
        <p><input type="submit"></p>
      </form>
    </article>
  </body>
</html>
```

When the page finishes loading, the second input field will be selected and ready to receive input, as shown in Figure 2-17.

Figure 2-17. *Listing 2-21 rendered in Chrome*

Of course, as soon as the user clicks anywhere else in the browser, the field will loose focus, and it will not return unless the user clicks on the field again. The autofocus only happens on page load.

Placeholder

Another commonly designed feature of forms is placeholder text inside of an `<input>` field. Placeholder text helps provide more information about what the field is for, and it disappears when the user starts typing. HTML5 includes a new `placeholder` attribute that can be applied to both `<input>` and `<textarea>` fields. The value specified for the attribute is used as placeholder text inside the field.

Listing 2-22 is a simple example of a form in which the user can compose and send an e-mail.

Listing 2-22. Placeholder Text

```
<!DOCTYPE html>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
  </head>
  <body>
    <article>
      <h1>Using the placeholder attribute</h1>
      <p><label for="message-title">Title:</label><br>
        <input placeholder="title of message" id="message-
title"></p>
      <p><label for="message-body">Body:</label><br>
        <textarea placeholder="body of message"
id="message-body"></textarea></p>
      <p><input type="submit" value="Send Email"></p>
    </article>
  </body>
</html>
```

In modern browsers the placeholder text will be visible as grayed-out text within the form fields (Figure 2-18).

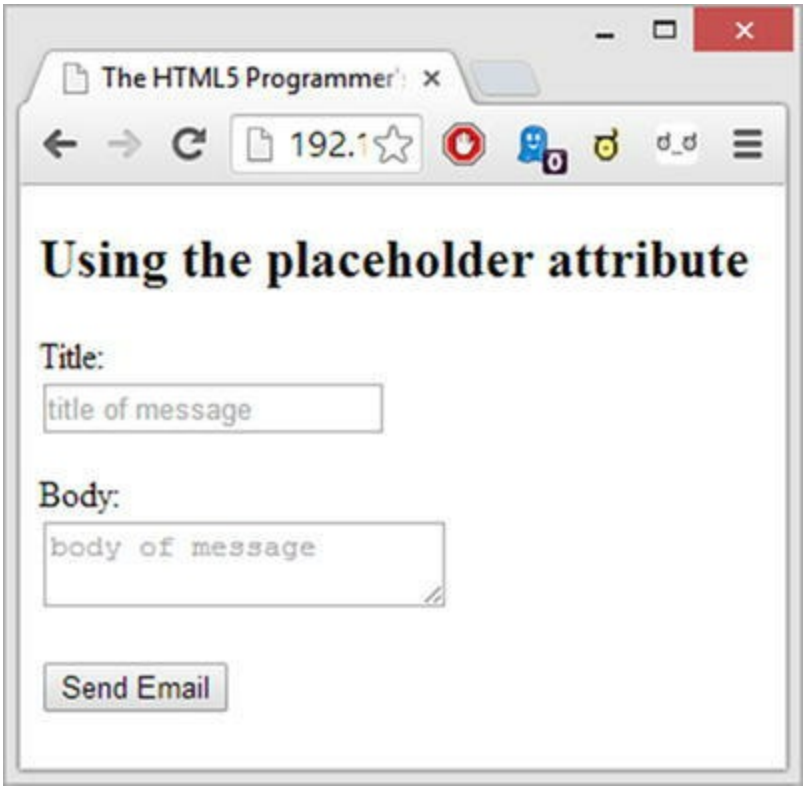


Figure 2-18. *Listing 2-22 rendered in Chrome*

As soon as the user begins to type in a field, the placeholder text will disappear, as shown in Figure 2-19.

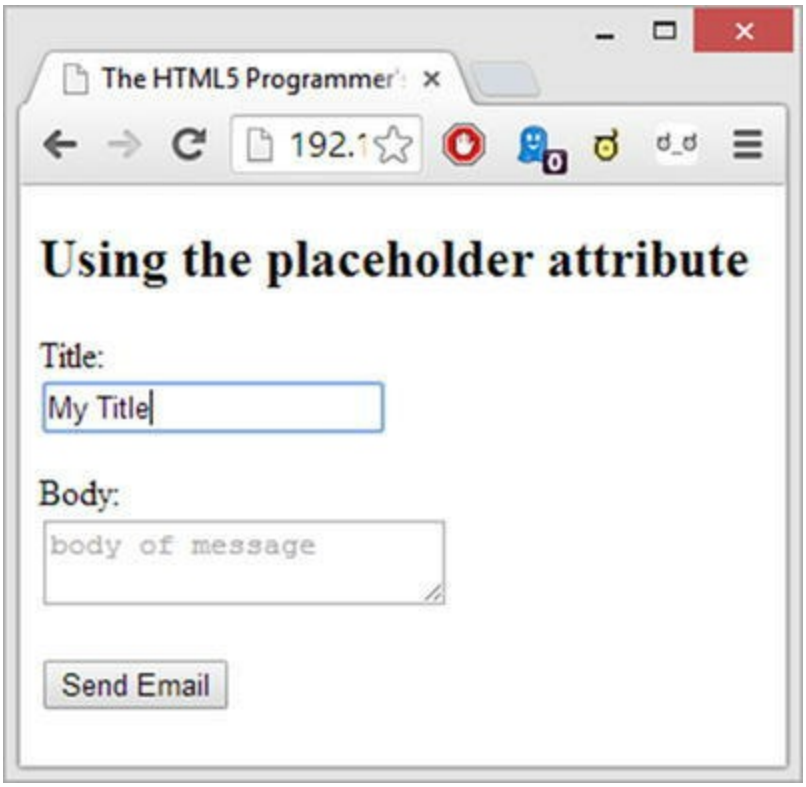


Figure 2-19. *Listing 2-22 after user input*

Note that this example still includes `<label>` tags. Placeholder text is a nice design concept, but it should not replace `<label>` tags, which are an important part of form accessibility. You don't typically need both—as you can see in the example, the labels are somewhat redundant. In this case, you can simply hide the `<label>` tags with CSS, as per Listing 2-23.

Listing 2-23. Hiding Labels with CSS

```
label: {
  display: none;
}
```

This simple bit of CSS makes the form look much nicer in Figure 2-20.

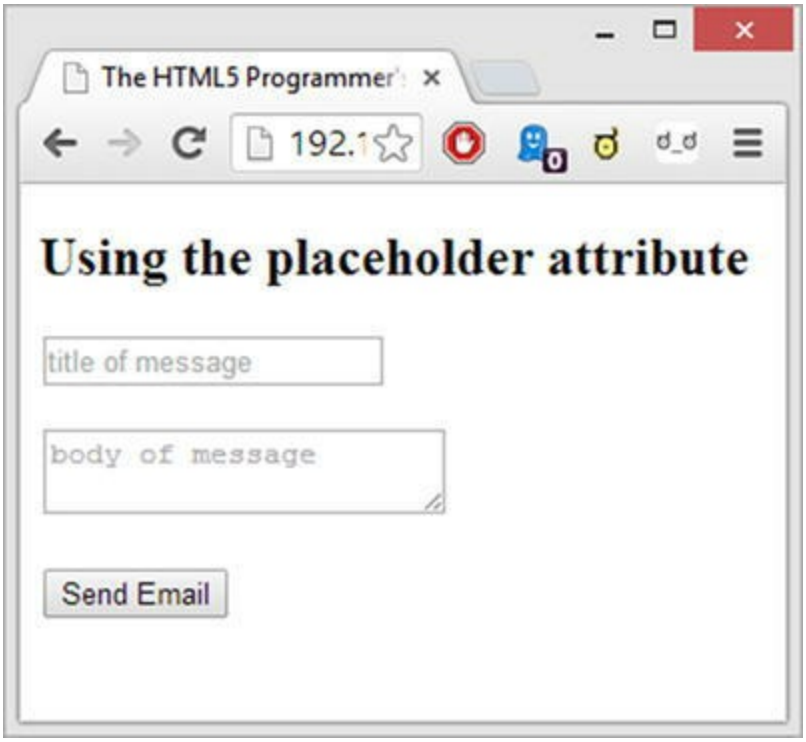


Figure 2-20. *Listing 2-23 rendered in Chrome*

New Input Types

The HTML5 specification also includes new values for the `input` element's `type` attribute. You're probably familiar with using this attribute to create checkboxes and radio buttons:

```
<input type="checkbox">
<input type="radio">
```

HTML5 adds several new types that add new user interface capabilities to the input field, from color and date pickers to search boxes. Unfortunately, support for these new input types varies widely from browser to browser.

For desktop browsers, Chrome has the best support, with Firefox and Internet Explorer both far behind. This limits their usefulness for desktop applications, unfortunately.

On mobile browsers the support is better. Most of the new input types will use the device's special keyboards and input widgets. For example, when an input field with the type of `tel` is active in Safari Mobile, the phone will display the telephone keyboard. This makes it easier for mobile users to enter a phone number.

Even though these new types aren't widely supported now, support is growing for them, especially on mobile devices. Given the benefits of using specific keyboards for mobile input, it's a good practice to use these input types even if they're not widely supported.

The new input types are as follows:

- `color`: Allows the user to select a color. In Chrome desktop this displays a color chip that, when clicked, shows the host operating system's color-picker user interface widget. No other browser supports this element.
- `date`, `datetime`, `datetime-local`, `month`, `time`, and `week`: These input types allow users to input dates and times. In Chrome desktop these display calendar and time selection widgets that are built into Chrome. On mobile devices these display date and time selectors (on iOS these take the form of spinners).
- `email`: Indicates an input field that will be used for collecting e-mail addresses. On mobile devices this type will display an Internet address–friendly keyboard when active. On iOS this keyboard takes the form of a regular keyboard with the @ key easily available, and a .com key.
- `number`: This input type specifies that the user will be entering a number. On Chrome and Firefox desktop, this input type will display a simple increase/decrease widget. On mobile devices this input type will display the numeric page of the alphanumeric keyboard.

- `range`: Displays a slider widget. This is the only input type that is widely supported by all browsers on both desktop and mobile.
- `search`: Indicates that the field is a search field. The primary difference between a search field and a regular input field is that search fields include a "clear" functionality, typically implemented as an × button at the edge of the input field. On Chrome and Internet Explorer desktop this displays a simple search field, with a clear button on the right.
- `tel`: Indicates that the field will be used to enter a telephone number. On mobile devices, an input field with this type will display the telephone number keyboard when active.
- `url`: Indicates the field will be used to enter a URL, most probably a web address. On mobile devices this will display an Internet address–friendly keyboard while active.

If you would like to test these input fields, Listing 2-24 has a full set of them that you can load into any browser.

Listing 2-24. New Input Types Demonstrated

```
<!DOCTYPE html>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
    <meta name="viewport" content="width=device-width,
user-scalable=no">
    <style>
input {
  display: block;
  margin-bottom: 20px;
}
    </style>
  </head>
  <body>
    <article>
      <h1>New HTML5 Input Types</h1>
      <form id="mainform" onsubmit="return false">
        <label>type=color</label>
        <input type="color">
```

```
        <label>type=date</label>
        <input type="date">
        <label>type=datetime</label>
        <input type="datetime">
        <label>type=datetime-local</label>
        <input type="datetime-local">
        <label>type=email</label>
        <input type="email">
        <label>type=month</label>
        <input type="month">
        <label>type=number</label>
        <input type="number">
        <label>type=range</label>
        <input type="range">
        <label>type=search</label>
        <input type="search">
        <label>type=tel</label>
        <input type="tel">
        <label>type=time</label>
        <input type="time">
        <label>type=url</label>
        <input type="url">
        <label>type=week</label>
        <input type="week">
      </form>
    </article>
  </body>
</html>
```

Chrome is the most interesting browser for this example. It produces several highly useful widgets for several of the types, as shown in Figure 2-21.

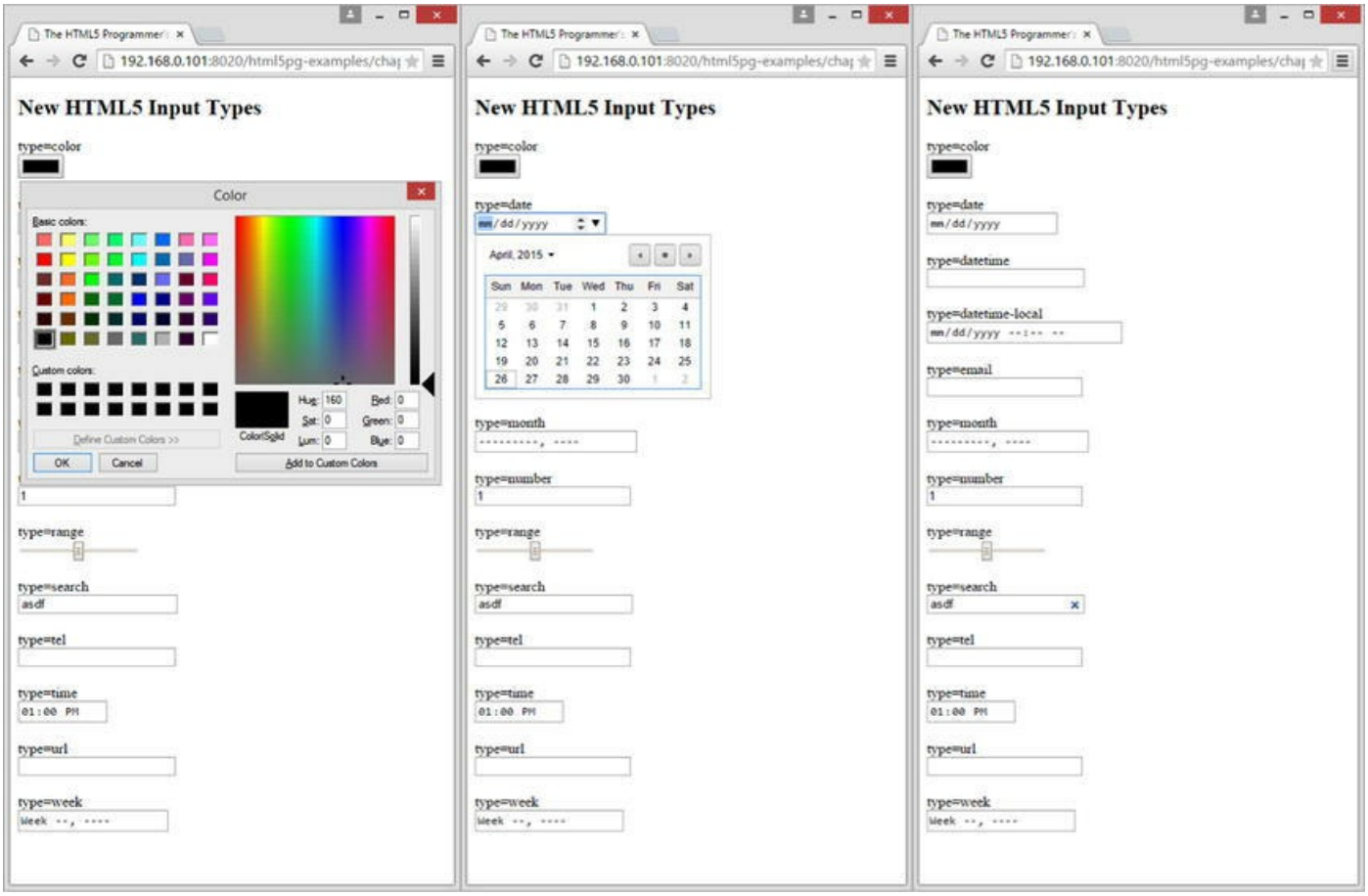


Figure 2-21. Color, date, and search input types rendered in Chrome desktop

Deprecated Elements and Obsolete Parameters

HTML5 has officially deprecated several elements. Some of these have been replaced with new tags or superseded by CSS features, while others are just no longer needed.

- `<applet>`: Use `<embed>` or `<object>` instead.
- `<acronym>`: Use `<abbr>` instead.
- `<frame>`, `<frameset>`, and `<noframes>`: Frame sets have been completely deprecated in HTML5. Instead, consider using iframes or a server-side technology.
- `<strike>`: Use `<s>`, unless the markup is for an edit, in which case use `<del>`.

- <basefont>, <big>, <blink>, , <font>, <marquee>, <multicol>, <nobr>, <spacer>, and <tt>: Use appropriate elements or CSS instead. For the <tt> element use <kbd> (to denote keyboard input), <var> (for variables), <code> (for computer code), or <samp> (for sample output). In the case of the <big> element, use header tags if the content is a heading, the element for denoting emphasis or importance, or the <mark> element for highlighting references.

In addition, HTML5 has rendered obsolete many parameters on existing elements. Again, many of these parameters have been replaced by CSS or other features, while others were holdovers from earlier versions of HTML or XHTML.

- The properties background, datasrc, datafld, and dataformats have been deprecated from all applicable tags. In the case of background, use CSS to apply backgrounds to elements.
- <a>: charset, coords, methods, name, rev (use rel with an opposite term), shape (use area for image maps), and urn.
- <body>: alink, bgcolor, link, marginbottom, marginheight, marginleft, marginright, margintop, marginwidth, text, and vlink.
-
: clear.
- <caption>: align.
- <col>: align, char, charoff, valign, and width.
- : align.
- <dl>: compact.
- <hr>: align, color, noshade, size, and width.
- All header tags: align.
- <iframe>: align, allowtransparency, frameborder, hspace, longdesc, marginheight, marginwidth, scrolling, and vspace.
- <img>: align, border, datasrc, hspace, longdesc, lowsrc, name, vspace.
- <input>: align, hspace, ismap, usemap, vspace.
- <legend>: align.

- <link>: charset, methods, rev, target, and urn.
- <menu>: compact.
- <object>: align, archive, border, classid, code, codebase, codetype (use data and type attributes and the <param> element), declare, hspace, standby, and vspace.
- : compact.
- <option>: name (use id instead).
- <p>: align.
- <param>: type and valuetype (use name and value attributes).
- <pre>: width.
- <script>: event, for, and language.
- <table>: align, bgcolor, border, bordercolor, cellpadding, cellspacing, frame, rules, summary, and width.
- <tbody>: align, char, charoff, and valign.
- <td>: abbr, align, axis, bgcolor, char, charoff, height, nowrap, scope, valign, and width.
- <tfoot>: align, char, charoff, and valign.
- <th>: align, bgcolor, char, charoff, height, nowrap, valign, and width.
- <thead>: align, char, charoff, and valign.
- <tr>: align, bgcolor, char, charoff, and valign.
- : compact and type.

Though browsers may still render these tags and recognize these properties in an HTML5 document, you should not use them. Any validator should throw errors on them as well.

Summary

In this chapter, I have covered the highlights of the Elements section of the HTML5 specification:

- I discussed how the evolution of HTML has been influenced by semantics.
- I took you on a brief tour of the new tags that HTML5 introduces for sections, grouping and semantics. These tags further expand the languages capabilities to handle complex documents and layouts.
- I explored the new multimedia features of HTML5, and demonstrated the basic use of the `<audio>` and `<video>` tags.
- I then gave you a look at the new interactive elements specified by HTML5: dialogs and progressive disclosure. Unfortunately, these features are not yet well supported, but that should change as time passes.
- I reviewed the changes to forms that HTML5 has introduced, some of which are especially useful in a mobile context.
- And finally, I went over the tags and attributes that have been deprecated in HTML5.

In the next chapter, you'll dive into the JavaScript APIs that HTML5 specifies.

CHAPTER 3

HTML5 APIs

As mentioned in Chapter 1, the HTML5 standard differs from previous HTML standards in that it is more than just the definition for a markup language. Since the standard was designed to be a platform for creating web applications as well as web pages, it introduces a host of new features designed to make building applications easier: new ways for the browser to communicate with the server, new ways to store and retrieve data, support for common user interactions like drag and drop, and so forth.

Like the new `audio` and `video` tags, many of these new web application features have been implemented in the past using extensive JavaScript programs or even browser plugins. Now, with HTML5, the browser manufacturers implement them directly in their browsers.

All of these new features can be used in JavaScript programs, so browser manufacturers provide interfaces for accessing them with your scripts. These interfaces typically take the form of JavaScript objects and methods. The HTML5 standard defines these interfaces as well, so the JavaScript objects and methods appear and function the same way in all browsers that implement the standard (assuming they implement the standard completely and correctly).

These interfaces are called Application Programming Interfaces (APIs). If you've done any scripting at all in a browser you're probably already familiar with one of the most common APIs: the Document Object Model (DOM). The DOM is an API for accessing the document currently rendered in the browser. Any method for fetching a reference to an element in the browser (like `getElementById`) is a part of the DOM API, as is any method for accessing the event model (like `addEventListener`). The browser also publishes the `Navigator` API for accessing internal browser features like browsing history. Another commonly used API is the `XMLHttpRequest` constructor, which is an interface to the browser's network communications system that allows you to communicate asynchronously with the server.

Note The term *API* has fairly broad application, from libraries and frameworks to network services (including many web services that you can access asynchronously with the browser) and even internal interfaces between objects in a single application.

In this chapter I'll cover the new APIs that are defined in the HTML5 specification. I'll take a practical approach to the APIs, focusing on how well the feature is supported and what you can do with it, and provide plenty of examples that you can use as a starting point for your own web applications.

Server-sent Events

SUPPORT LEVEL
Mixed

Internet Explorer does not support this API. All other browsers have full support.

WHATWG Living Standard: `http://www.whatwg.org/specs/web-apps/current-work/multipage/comms.html#server-sent-events`

W3C Draft: `http://dev.w3.org/html5/eventsource/`

Imagine you're building a simple stock ticker application. You have a server resource that publishes the APIs you need for getting stock values, so it's quite easy to get things started. But how do you get updates to the stock values? How does the server let your client application know that a stock's value has changed?

This is probably the canonical use case for Server-sent Events: a situation in which the server needs to inform the client that something has happened. Because HTTP as a standard only defines stateless communication, and thus only clients can initiate requests to servers, there was no way for a server to send a message to a client without the client first asking for one. Server-sent Events is one of the ways that HTML5 addresses this issue, in the specific case of one-way communication from the server to the client.

In the past, people wrote simple polling timers into their scripts that would essentially ask the server "Is there anything new?" on a timer, as shown in Listing 3-1.

Note The examples in this section will need to be run from a server, rather than loaded directly from your filesystem.

Listing 3-1. An Example Polling Script

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
  </head>
  <body>
  <script>

// This is a mock API that simply returns the same JSON
structure
// every time the URL is requested. This JSON structure
has a single
// property, isChanged, which is set to false.
var strUri = './example3-1.json';

// This is how often we'll query the mock API, in
milliseconds.
var timerLength = 1000;

/**
 * Fetch an update from a web service at the specified
URL. Initiates an
 * XMLHttpRequest to the service and attaches an event
handler for the success
 * state.
 * @param {string} strUrl The URL of the target web
service.
 */
function fetchUpdates(strUrl) {
  // Create and configure a new XMLHttpRequest object.
  var myXhr = new XMLHttpRequest();
  myXhr.open("GET", strUrl);
  // Register an event handler for the readyStateChange
event published by
  // XMLHttpRequest.
  myXhr.onreadystatechange = function() {
```

```
    // If readyState is 4, request is complete.
    if (myXhr.readyState === 4) {
      handleUpdates(myXhr.responseText);
    }
  };
  // Everything is configured. Send the request.
  myXhr.send();
}

/**
 * Handle an update from the mock API. Parses the JSON
returned by the API and
 * looks for changes.
 * @param {string} jsonString A JSON-formatted string.
 */
function handleUpdates(jsonString) {
  // Parse the JSON string.
  var jsonResponse = JSON.parse(jsonString);
  if (jsonResponse.isChanged) {
    // Handle changes here, probably by checking the
structure to determine
    // what changed and then route the change to the
approprate part of the app.
    console.log('change reported.');
  } else {
    console.log('no changes reported.');
  }
}

// Everything is all set up: we have a function for
fetching an update and a
// function to handle the results of an update query. Now
we just have to kick
// off everything. Using setInterval, we will call our
fetchUpdates method every
// timerLength milliseconds. We can cancel the timer by
calling the
// cancelInterval(pollTimer) method.
var pollTimer = setInterval(fetchUpdates.bind(this,
strUri), timerLength);
  </script>
  </body>
```

```
</html>
```

This example does a lot of setup before it starts the polling process. Begin by defining the URL for the mock API, which is just a file you have created on the filesystem. You also define how often you want to poll your mock API. Then create a function for fetching an update. This is the method you will call with the timer, and it initiates the `XMLHttpRequest` (XHR for short) to the mock service. The XHR will publish `readyStateChange` events as it communicates with the server. By looking at the `readyState` property of the XHR object, you can tell what state the query is in: still talking with the server, or done talking, or even if an error has occurred. So add an event handler to the XHR object that will be called each time a `readyStateChange` event occurs. In this case you're using an inline function to keep the code simple, though you could have defined it outside of this code block and referred to it by name. The event handler checks the `readyState` property and, if it is in the correct state, it calls the `handleResponse` function. That function takes the JSON-formatted string that you fetched from your mock API and processes it accordingly.

This is a pretty unsophisticated example, but it demonstrates the basics of using a timer to regularly poll a web service. Using the methods `setInterval` and `cancelInterval` it's easy to start and stop timers. If your application will need multiple timers, you can build a constructor with convenience methods for creating, starting, stopping, pausing, and disposing of them. You can rapidly build up a lot of code just around creating and managing your timers.

And if you think about it, simple timers aren't really a good way of handling this problem. What if there is a temporary network problem and one of the calls to `fetchUpdates` takes longer than a second? In that case, another call to `fetchUpdates` would be executed before the first has returned, resulting in two active and pending calls to the server. Depending on the network conditions, the second call could return before the first. This situation is referred to as a *race condition,* because the two pending calls are essentially in a race to see which one completes first.

Fortunately this race condition is fairly easy to fix: instead of having a timer fire off requests regardless of external limitations, alter the `handleUpdates` method so that the last thing it does is schedule the next call to `fetchUpdates`. That way you won't ever have more than one call happening and the race condition is eliminated. The script would change as shown in Listing 3-2 (the surrounding HTML remains the same).

Listing 3-2. Eliminating the Race Condition from Example in Listing 3-1

```
// This is a mock API that simply returns the same JSON
structure
// every time the URL is requested. This JSON structure
```

```
has a single
// property, isChanged, which is set to false.
var strUri = ’./example3-1.json’;

// This is how often we’ll query the mock API, in
milliseconds.
var timerLength = 1000;

// Reference to the currently active timer.
var pollTimeout;

/**
 * Fetch an update from a web service at the specified
URL. Initiates an
 * XMLHttpRequest to the service and attaches an event
handler for the success
 * state.
 * @param {string} strUrl The URL of the target web
service.
 */
function fetchUpdates(strUrl) {
  // Create and configure a new XMLHttpRequest object.
  var myXhr = new XMLHttpRequest();
  myXhr.open("GET", strUrl);
  // Register an event handler for the readyStateChange
event published by
  // XMLHttpRequest.
  myXhr.onreadystatechange = function() {
    // If readyState is 4, request is complete.
    if (myXhr.readyState === 4) {
      handleUpdates(myXhr.responseText);
    }
  };
  // Everything is configured. Send the request.
  myXhr.send();
}

/**
 * Handle an update from the mock API. Parses the JSON
returned by the API and
 * looks for changes, and then initiates the next query to
the mock service.
 * @param {string} jsonString A JSON-formatted string.
```

```
 */
function handleUpdates(jsonString) {
  // Parse the JSON string.
  var jsonResponse = JSON.parse(jsonString);
  if (jsonResponse.isChanged) {
    // Handle changes here, probably by checking the
structure to determine
    // what changed and then route the change to the
approprate part of the app.
    console.log('change reported.');
  } else {
    console.log('no changes reported.');
  }
  // Schedule the next polling call.
  pollTimeout = setTimeout(fetchUpdates.bind(this, strUri),
timerLength);
}

// Initiate polling process.
fetchUpdates(strUri);
```

The changes from the previous version of the script have been bolded. This version of the code eliminates the race condition, because only one call to `fetchUpdates` will be active at any given time. However, you could now be polling the server at an unpredictable rate.

It is possible to program around these problems as well, but handling all of the edge cases well can be difficult, and you will end up with a significant amount of code, all just to intelligently handle polling a web service.

Ideally, this sort of communication should be a feature of the browser, and that's what the new Server-sent Events feature does. Server-sent Events has the browser handle all of the details of connecting to the server and polling it for events, and lets you leave behind timer-based polling scripts and all of the problems they entail. Server-sent Events provide a way for you to open a channel to the server, and then attach event listeners to that channel to handle various event types that the server will publish. The browser handles everything else for you.

To use Server-sent Events, you need not only a client that can open a channel to a web service and handle the events that occur on that channel; you also need a server that will handle incoming channel subscriptions correctly and publish correctly formatted events.

Client Setup

The Server-sent Events specification defines a new constructor, `EventSource`, in the global context, which you can use to create new connections to the server:

```
var serverConnection = new EventSource(targetUrl);
```

The constructor returns an `EventSource` object that is an interface to a connection that the browser will maintain to the server resource specified by `targetUrl`. The browser will handle all of the connection maintenance and polling for you—all you have to do is listen for events from the server.

As the server publishes events to the connection, the server resource will publish events that will be dispatched from the `EventSource` object. Like any DOM event, you can attach event handlers to the `EventSource` object using the `addEventListener` method.

By default, the `EventSource` interface will publish three event types:

- `open`: Published when the connection is first opened and network communication has been initialized. Useful for initializing the connection. Fires at most once, and if the browser fails to establish a connection to the specified service, it won't fire at all.
- `message`: Published when the server sends a new message.
- `error`: Published if an error occurs in the connection (e.g., the connection times out).

When an event is dispatched, the `EventSource` will call the event handling function registered for that event type. The function will be called with an `event` object as a parameter, and that `event` object will have a `data` attribute that will contain the data that was sent from the server. Listing 3-3 shows how to create a new `EventSource` object and attach event listeners to it.

Listing 3-3. Stubbed EventSource Event Handlers and Subscriptions

```
/**
 * Handles message events published by the EventSource.
 * @param {EventSourceEvent} event
 */
function handleMessage(event) {
  // Handle message.
  console.log('A message was sent from the server: ',
event.data);
}
```

```
/**
 * Handles error events published by the EventSource.
 * @param {EventSourceEvent} event
 */
function handleError(event) {
  // Handle an error.
  console.error('An error happened on the EventSource: ',
event.data);
}

/**
 * Handles an open event published by the EventSource.
 * @param {EventSourceEvent} event
 */
Function handleOpen(event) {
  // Handle the open event.
  console.log('The connection is now open.');
}

// Create a new connection to the server.
var serverConnection = new EventSource(targetUrl);

// Attach event handlers.
serverConnection.addEventListener('message',
handleMessage);
serverConnection.addEventListener('error', handleError);
serverConnection.addEventListener('open', handleOpen);
```

Now whenever the resource specified by `strUrl` publishes an event, the `handleMessage` event handler will be called. If an error arises in the connection the browser will publish an `error` event and the `handleError` event handler will be called. Note that you can configure your server to publish custom event types, and you can add event handlers for them in exactly the same way (see next section, "Sending Events from the Server").

To close the connection to the server, call the `close` method on the `EventSource` object:

```
serverConnection.close();
```

This will signal the browser to stop polling the server and close the connection. You can then set the EventSource object to `null` to eliminate it from memory. There is no way to reopen a connection once it has been closed.

Sending Events from the Server

For Server-sent Events to work, you need a resource on a server that knows how to handle the incoming polling requests from the browser and how to correctly publish events as needed. The server resource can be written in any language—Java, PHP, JavaScript, C++, and so forth. The resource must respond to polling requests with the `text/event-stream` MIME type. Responses consist of multiline `key: value` pairs, and are terminated by a double linefeed. Valid keys are as follows:

- `data`: This specifies a line of arbitrary data to be sent to the client, which will receive it as the data property of the `event` object.
- `event`: Specifies an arbitrary event type associated with this Server-sent Event. This will cause an event of the same name to be dispatched from the active `EventSource` object, thus enabling arbitrary events beyond `open`, `message`, and `error` to be fired from the server. If no event type is specified, the event will just trigger a `message` event on the `EventSource`.
- `id`: This specifies an arbitrary ID to associate with the event sequence. Setting an ID on an event stream enables the browser to keep track of the last event fired, and if the connection is dropped it will send a `last-event-ID` HTTP header to the server.
- `retry`: Specifies the number of milliseconds before the browser should re-query the server for the next event. By default this is set to `3000` (three seconds). This enables the server resource to throttle browser queries and prevent itself from being swamped.

For example, a basic Server-sent Event would look like this:

```
data: arbitrary line of text\n\n
```

This event would trigger a `message` event on the associated `EventSource` object and call the `message` event handler (assuming one was registered). The `event` object received by the message event handler will have a `data` attribute, which will contain the text sent by the server (in the preceding case, it would be "arbitrary line of text").

You can send multiple line events as well—just terminate the event with a double-linefeed:

```
data: arbitrary line of text\n
```

```
data: another arbitrary line of text\n\n
```

In this case, the `event.data` attribute would be set to "arbitrary line of text\nanother arbitrary line of text". The event data can be any text: HTML, CSS, XML, JSON, and so forth.

Multiple event types can be included in a single Server-sent Event as well. Going back to the original example of a stock ticker, here is an event that shows updates on two different stocks:

```
event: update\n
data: {\n
data: "ticker":"GOOG",\n
data: "newval":"559.89",\n
data: "change":"+0.05"\n
data: }\n
event: update\n
data: {\n
data: "ticker":"GOOGL"\n
data: "newval":"571.65"\n
data: "change":"+1.09"\n
data: }\n\n
```

This single Server-sent Event would trigger two `update` events on the `EventSource` object. Each time the `update` event handler would be called, with an `event` object containing the data for that event. The data for the first event would be the following JSON-formatted text:

```
{
  "ticker":"GOOG",
  "newval":"559.89",
  "change":"+0.05"
}
```

And the data for the second event would be the following JSON-formatted text:

```
{
  "ticker":"GOOGL",
  "newval":"571.65",
  "change":"+1.09"
}
```

Origin Limitations

Server-sent Events are subject to the Same Origin Policy, so a page from one origin cannot subscribe to an event stream from another. In particular, event streams are often published on different TCP ports than standard web pages, so it's not possible for a web page published on a standard port like port 80 to subscribe to an event stream published on a different port, even if it is from the same server. Only clients served from the same origin as the event stream can access that event stream.

If you want to use Server-sent Events, you will need to have a web server that is flexible enough to serve the HTML-based client (and all of its dependent resources like CSS, JavaScript, images, etc.) and publish event streams. This makes server-integrated scripting languages like PHP, JSP, or ColdFusion prime candidates for building application servers that rely on Server-sent Events, because you can write the event streams in the integrated scripting language and serve the clients using the same web server. It's also quite easy to configure most web servers to route requests to special URLs to different resources, making it possible to publish both regular web content and event streams from the same server. The details of such implementations are beyond the scope of this book, but this is an important limitation to Server-sent Events.

In the example that follows, you'll opt for a simpler solution: building a server that can serve the client HTML file while acting as an event stream. Since both the HTML client file and the event stream are from the same origin, there will be no problems with the subscription.

Security

Just as with any technique for handling network communication, it's a good idea to be conscious of security when you're building applications with Server-sent Events. Never send sensitive information (e.g., passwords) on unencrypted connections, because server events are sent in plain text. If you need to send sensitive information you should at the very least use HTTPS.

As mentioned, Server-sent Events are limited by the Same Origin Policy, so a script cannot subscribe to an event stream from a network resource different than its own. In addition, the event object received by EventSource event handlers will have an origin property that you can check to verify that the server event is coming from the source you expect, as shown in Listing 3-4.

Listing 3-4. Checking Event Origins from Server-sent Events

```
// The EventSource object.
var serverConnection;
```

```
/**
 * Handle an event published on an EventSource object.
 * @param {EventSourceEvent} event
 */
function messageHandler(event) {
  if (event.origin !== 'https://www.myexample.com') {
    // Something bad has happened, stop listening for
events and surface a warning to the user.
    serverConnection.close();
    alert('Warning: Server event received from wrong
network resource.');
    return;
  }
  // Handle event here.
}

// Initiate subscription to event stream and register
event handler.
serverConnection = new EventSource(targetUrl);
serverConnection.addEventListener('message',
messageHandler);
```

In this snippet you're checking the origin of the event as reported by the browser. This is not a foolproof check, however, as it can be spoofed, but it's one more layer of security you can add to your application.

An Example Application

To build a functional example application, you'll need a server resource that can send the events in the expected format and can also serve the client that will subscribe to the event stream. As mentioned, you could use any language to build this server resource, but to stay consistent with the other examples in the book, use JavaScript. You're probably used to using JavaScript in the browser. You can also use it on a server, just like any other scripting engine. The most popular implementation of JavaScript as a standalone scripting engine is the Node.js framework, which has been ported to multiple operating systems. The Node.js framework provides a fast JavaScript interpreter and a framework of APIs for accessing the filesystem, network stack, and other resources on the server.

Tip If you've never used Node.js, you can learn more about it at

http://nodejs.org.

To run this example, you'll need a server with Node.js installed. You'll build a simple script that will both act as the event streamer and serve the event client. As you can see in Listing 3-5, it's quite easy to build a simple HTTP server with Node.js.

Listing 3-5. A Simple Event Stream Server Written in JavaScript

```
// Include the modules needed to build the server.
var http = require('http');
var sys = require('sys');
var fs = require('fs');

// Use the http.createServer method to create a server
listening on port 8030.
// The server will call the handleRequest method each time
a request is
// received.
http.createServer(handleRequest).listen(8030);

/**
 * Handle an incoming request from the server.
 * @param {Object} request The request headers.
 * @param {Object} resource A reference to the server
resource that received
 * the request.
 */
function handleRequest(request, resource) {
  // Incoming requests to our server will be to one of two
URLs.
  // If the request is for /example3-5-events we should
send our SSE.
  // If the request is for /example3-5.html, we should
serve the example client.
  if (request.url == '/example3-5-events') {
    // Initialize an event stream.
    sendSSE(request, resource);
  } else if (request.url == '/example3-6.html'){
    // Send the client.
    resource.writeHead(200, {'Content-Type':
'text/html'});
```

```
    resource.write(fs.readFileSync('example3-6.html'));
    resource.end();
  }
}

/**
 * Initializes an event stream and starts sending an event
every 5 seconds.
 * @param {Object} request
 * @param {Object} resource
 */
function sendSSE(request, resource) {
  // Initialize the event stream.
  resource.writeHead(200, {
    'Content-Type': 'text/event-stream',
    'Cache-Control': 'no-cache',
    'Connection': 'keep-alive'
  });

  // Send an event every 5 seconds.
  setInterval(function() {
    // Randomly generate either 0 or 1.
    var randNumber = Math.floor(Math.random() * 2);
    // If the random number is 1, set isChanged to true.
Otherwise, set it to
    // false.
    var isChanged = (randNumber === 1) ? true : false;
    resource.write('data: ' + '{"isChanged":' + isChanged
+ '}\n\n');
  }, 5000);
}
```

If you request `example3-6.html` it will serve the HTML client (which you'll define in Listing 3-6), and if you request `example3-5-events` it will initiate an event stream that will push an event to the client every five seconds. The event will be a simple JSON-formatted string with an `isChanged` property that will be set randomly to `true` or `false`. To run this server, use the following command:

```
node example3-5server.js
```

The HTML client for this server just has to initiate the `EventSource` to the correct URL, as shown in Listing 3-6.

Listing 3-6. A Server-sent Event Client

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
    <style>
#changeme {
  width: 300px;
  height: 300px;
  border: 1px solid black;
  overflow: auto;
}
    </style>
  </head>
  <body>
    <h1>Server-sent Events Demonstration</h1>
    <div id="changeme"></div>
  <script>
// The URL for our event stream. Note that we are not
specifying a domain or
// port, so they will default to the same ones used by the
host script.
var strUri = '/example3-5-events';
// Get a reference to the DOM element we want to update.
var changeMe = document.getElementById('changeme');

// Create a new server-side event connection and register
an event handler for
// the 'message' event.
var serverConnection = new EventSource(strUri);
serverConnection.addEventListener('message', handleSSE,
false);

/**
 * Handles a server-sent event by parsing the JSON in the
data and handling
 * any changes.
 * @param {EventSourceEvent} event The event object from
the event source.
 */
function handleSSE(event) {
```

```
  // Parse the JSON string.
  var jsonResponse = JSON.parse(event.data);
  // Create a new element to append to the DOM.
  var newEl = document.createElement('div');
  if (jsonResponse.isChanged) {
    newEl.innerHTML = 'Change reported.';
  } else {
    newEl.innerHTML = 'No changes reported.';
  }
  // Append the new element to the DOM.
  changeMe.appendChild(newEl);
}
  </script>
  </body>
</html>
```

This client initiates a new `EventSource` for the event stream's URL and then attaches a message event handler to it. Every time the server publishes an event, the message event handler is called, the event data is parsed, and the results are appended to the DOM. This client is a lot simpler than your previous polling example because all of the details are handled by the browser now. There's no need to initiate an `XMLHttpRequest` object, no need to manage your own timers—all you have to do is initialize an `EventSource` object and register event handlers.

Server-sent Events only allow for one-way communication from the server to the browser. For full duplex communication, see the section on WebSockets.

WebSockets

SUPPORT LEVEL
Good

All modern browsers support WebSockets.

WHATWG Living Standard: https://html.spec.whatwg.org/multipage/comms.html#networ

W3C Draft: http://www.w3.org/TR/websockets/

WebSockets build on Server-sent Events by providing full duplex communication between client and server: not only can the server send arbitrary information to the client, but the client can transmit arbitrary information back to the server. In addition, WebSockets are not beholden to the Same Origin Policy, which prevents scripts from one origin from interacting with pages from a different origin.

The WebSocket JavaScript API in the browser manifests as a new WebSocket constructor in the global context, which returns a WebSocket object:

```
var mySocket = new WebSocket(url, protocols);
```

The `url` parameter specifies a valid URL to a WebSocket compliant service. WebSockets have their own communication protocol, which is different than the hypertext transfer protocol (HTTP) we see all the time. The WebSocket protocol is specified by either `ws://` (for a standard WebSocket connection) or `wss://` (for a secure WebSocket connection). Any attempt to specify a different protocol (such as `http` or `https`) when constructing a WebSocket will result in an error.

The optional `protocols` parameter is either a single protocol string or an array of protocol strings specifying one or more sub-protocols implemented by the server. (A *protocol* in this context is a set of rules for how data is transmitted between the client and the server.) This parameter does not change the overall network protocol used by the browser to create and maintain the connection with the server; instead it allows you to specify acceptable data formats for sending and receiving information through that connection. This enables a single server to implement multiple ways of transmitting data to and receiving data from clients. For example, you can implement a server that implements both a Server-sent Event protocol (using the key/value pairs specified for that API) and a protocol that sends JSON-formatted strings as text. Your client can then specify which protocol it expects. If a protocol is not specified, the server will have to choose a default. If the specified protocols are not available on the server, the server will refuse the connection.

Connecting to the Server: Inside the WebSocket Handshake

A web browser creates a two-way connection with a service using a handshake process. When you create a new WebSocket using the constructor, the browser will immediately send a simple `GET` query to the host specified in the URL. The query will contain all the headers needed to specify that the browser is attempting to create a WebSocket connection (as opposed to making a simple HTTP request). For example, assume you went to the web site at `example.com` and they were advertising their new

WebSocket-based chat service. (A chat service is a common example use case for WebSockets, since it involves the need to both send and receive messages.) You click the link to sign on to the chat service, and the application starts.

Behind the scenes, the browser will attempt to connect to the chat service. Assuming the service was also hosted on the same domain `example.com` at the URL `ws://www.example.com/chat`, the JavaScript socket creation might look something like:

```
var mySocket = new WebSocket('ws://www.example.com/chat',
['chat', 'json']);
```

and the resulting request and its headers would look something like:

```
GET /chat HTTP/1.1
Host: example.com:8000
Upgrade: websocket
Connection: upgrade
Origin: http://www.example.com
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Protocol: chat, json
Sec-WebSocket-Version: 13
```

The first line is a simple `GET` request. The "chat" part of the request is optional, but allows a single server to publish many WebSocket services.

The headers contain the information needed to establish a WebSocket connection with the server. Specifically, the `Sec-WebSocket-Key` header contains a unique identifier that the client will expect the server to use in a specific way in its response. The `Sec-WebSocket-Protocol` header contains the sub-protocols specified in the constructor. In this case the client is specifying that it knows the `'chat'` and `'json'` protocols.

The server will take the information in the headers and formulate a response, which could look something like the following:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

The value of the `Sec-WebSocket-Accept` header depends on the value of the `Sec-WebSocket-Key` header sent by the client. The client knows to expect a specific value from the server, and if a different one is specified (or if one is not

specified at all), the client knows that the server can't handle a WebSocket connection. The `Sec-WebSocket-Protocol` header indicates that the service has chosen the `'chat'` protocol for communication.

Note The WebSocket Protocol is defined in RFC 6455, "The WebSocket Protocol," which you can read at `https://tools.ietf.org/html/rfc6455`. That document specifies the entire protocol, including the details of handling the `Sec-WebSocket-Key` header to create the `Sec-WebSocket-Accept` value.

Once the handshake is complete, the client and server will communicate with one another using a special data frame protocol. This protocol allows the client and server to send arbitrary information to one another easily.

Receiving Information from the Server

On the client, interactions with the server are event driven: when communication happens with the server, specific events are published on the associated WebSocket connection object.

- `error`: An `error` event is published on the connection object when the WebSocket fails to connect to the server, or loses the connection.
- `open`: The `open` event is published on the connection object when the WebSocket first succeeds in connecting with the specified service. This event indicates that the socket is ready to send and receive data.
- `close`: When a WebSocket closes, either due to an error or because the client deliberately closed the connection, a `close` event is published on the connection object.
- `message`: When the server sends information through the connection, the browser will publish a message event on the connection object. The event will contain the data that was transmitted from the server.

Listing 3-7 demonstrates some stubbed event handlers for these events using the hypothetical chat service.

Listing 3-7. Stubbed Event Handlers for a Web Socket

```
// Create a new WebSocket connection to the chat service.
var chatUrl = 'ws://www.fgjkjk4994sdjk.com/chat';
var validProtocols = ['chat', 'json'];
var chatSocket = new WebSocket(chatUrl, validProtocols);

/**
 * Handles an error event on the chat socket object.
 */
function handleError() {
  console.log('An error occurred on the chat
connection.');
}

/**
 * Handles a close event on the chat socket object.
 * @param {CloseEvent} event The close event object.
 */
function handleClose(event) {
  console.log('The chat connection was closed because ',
event.reason);
}

/**
 * Handles an open event on the chat socket object.
 * @param {OpenEvent} event The open event object.
 */
function handleOpen(event) {
  console.log('The chat connection is open.');
}

/**
 * Handles a message event on the chat socket object.
 * @param {MessageEvent} event The message event object.
 */
function handleMessage(event) {
  console.log('A message event has been sent.');

  // The event object contains the data that was
transmitted from the server.
  // That data is encoded either using the chat protocol
or the json protocol,
```

```
  // so we need to determine which protocol is being used.
  if (chatSocket.protocol === validProtocols[0]) {
    console.log('The chat protocol is active.');
    console.log('The data the server transmitted is: ',
event.data);
    // etc...
  } else {
    console.log('The json protocol is active.');
    console.log('The data the server transmitted is: ',
event.data);
    // etc...
  }

// Register the event handlers on the chat socket.
chatSocket.addEventListener('error', handleError);
chatSocket.addEventListener('close', handleClose);
chatSocket.addEventListener('open', handleOpen);
chatSocket.addEventListener('message', handleMessage);
```

In this example you create a set of simple event handlers, one for each of the event types, then register those handlers on the connection object. You can actually run this example if you want. The domain `fgjkjk4994sdjk.com` doesn't exist, so first the browser will publish an `error` event on the connection, followed by a `close` event. In the console, you'll see something similar to Figure 3-1.

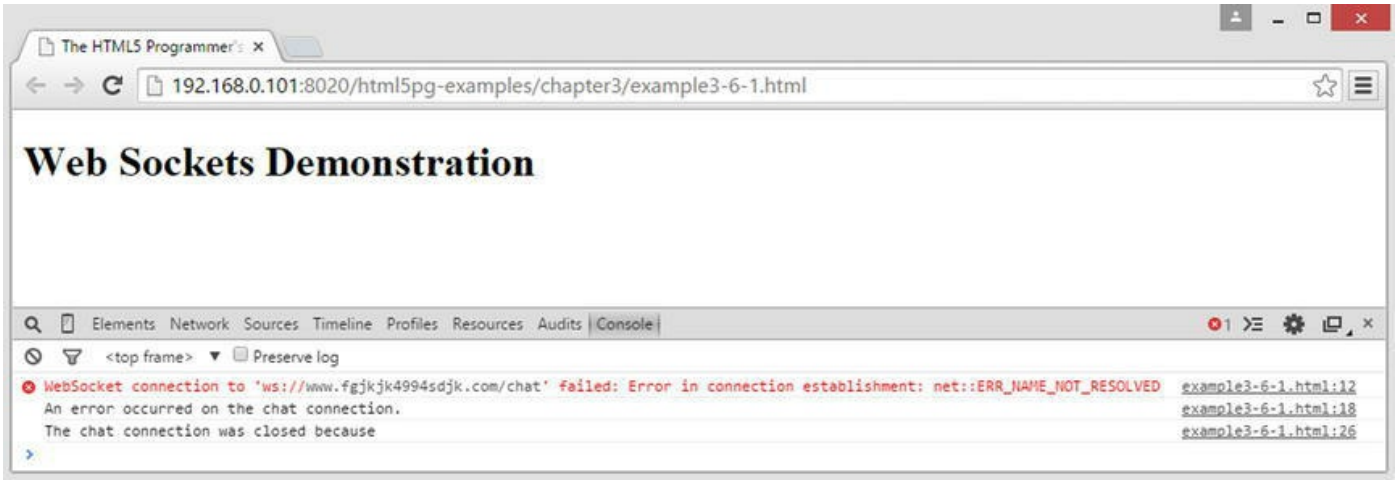


Figure 3-1. *The results of running Listing 3-7 in Chrome*

In the closed event handler `handleClose` you check the `reason` property on the `event` object to see if a reason for the closing was specified. This property may or may not be present, depending on the error that occurred and the subprotocol that was specified for the connection.

The `handleMessage` event handler is very simple, but demonstrates how to check the active subprotocol, and how to access the data that was transmitted by the server. We're used to simple text-based communication through HTTP (as with Server-sent Events), but WebSockets can transmit binary data as well. You could transmit any arbitrary binary data through a WebSocket; for example, you can send and receive images.

In JavaScript, binary data is represented using either *binary large objects* (also known as *blobs*) or *array buffers*. Both of these are valid data types in JavaScript: `Blob` represents blobs and `ArrayBuffer` represents array buffers. The difference between the two types is how the data is being used. If you are working with a single chunk of raw data that never has to be changed (like an image) it is best represented by a `Blob`. If you need to process the data (look at parts of it, or even change it), it's probably best represented using an `ArrayBuffer`. Both of these are data types in JavaScript, so it's easy to check to see if the information transmitted from the server is in that format. Here's an update to the `handleMessage` event handler that demonstrates checking for `Blobs` and `ArrayBuffers`:

```
/**
 * Handles a message event on the chat socket object.
 * @param {MessageEvent} event The message event object.
 */
function handleMessage(event) {
  console.log('A message event has been sent.');

  // The event object contains the data that was
transmitted from the server.
  // That data is encoded either using the chat protocol
or the json protocol,
  // so we need to determine which protocol is being used.
  if (chatSocket.protocol === validProtocols[0]) {
    console.log('The chat protocol is active.');

    // Check the data type of the incoming data.
    if (event.data instanceof Blob) {
      console.log('The data is a Blob.');
    }
    if (event.data instanceof ArrayBuffer) {
      console.log('The data is an ArrayBuffer.');
    }

    console.log('The data the server transmitted is: ',
event.data);
```

```
    // etc...
  } else {
    console.log('The json protocol is active.');
    console.log('The data the server transmitted is: ',
event.data);
    // etc...
  }

}
```

Here you've added checks in the `chat` subprotocol section to determine if the data is a Blob or an `ArrayBuffer`. You could do a similar check in the `json` subprotocol section as well; it is possible to encode both `Blobs` and `ArrayBuffers` in JSON format (typically using Base64 encoding).

Note For more details on handling Blobs, see https://developer.mozilla.org/en-US/docs/Web/API/Blob, and for more details on working with Array Buffers, see https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ArrayBuffe

---

Sending Information to the Server

You can transmit information to the server using a WebSocket as well. Each WebSocket connection object has a `send` method that you can use to immediately transmit information to the server. You should make sure only to send information after the `open` event has fired, otherwise the information likely will be lost. As a result, you will often see a pattern in WebSocket applications in which the `open` event handler kicks off the initialization of an application to guarantee nothing happens until the connection is available.

The `send` method takes only one parameter: the data to send, as shown next:

```
var mySocket = new WebSocket(url);
mySocket.send('hello world!');
```

Valid data types are `Strings`, `Blobs`, and `ArrayBuffers`.

Closing the Connection

The connection object has a `close` method that can be used to close the connection to the server when you're done with it. Calling this method will immediately cause the client to send a `close` request to the server, which in turn will close the connection. The client will then dispatch a `close` event on the connection object. There is no way to reopen a closed WebSocket connection object.

An Example WebSocket Application

As with Server-sent Events, you will need a server capable of handling WebSocket connections in order to build a functioning example. Building such a server from scratch is a moderately difficult task, as the server has to know how to upgrade the HTTP connection to a WebSocket connection, and also how to send and receive data according to the WebSocket protocol. Fortunately you probably won't have to build one from scratch. There are many open source projects devoted to creating WebSocket servers that you can use in your projects. For the example WebSocket application you will build a simple WebSocket server using Node.js, a JavaScript framework for servers. Node.js provides a fast JavaScript interpreter as well as libraries for accessing the filesystem, network stack, and other server technologies.

Rather than build the server from scratch, you'll use WebSocket-Node, an open source implementation of the WebSocket protocols in Node.js. The home for this project is at `https://github.com/theturtle32/WebSocket-Node`. To install the module, use the node package manager npm:

```
npm install websocket
```

This should install the module for you. If it doesn't, see the Installation notes at the project home site.

Once WebSocket-Node is installed, you can use it to build a server for your example. The simplest WebSocket example is a server that simply echoes back anything that the client sends. Listing 3-8 shows how simple it is to build a server using the WebSocket-Node library.

Listing 3-8. A Simple WebSocket Server

```
// Include the modules needed to build the server.
var WebSocketServer = require('websocket').server;
var http = require('http');
var currentConnection;

// Define the subprotocol name for the WebSocket
```

```
connection.
var subProtocol = 'echo';

/**
 * Handles a request event on the WebSocket server.
 * @param {Object} request
 */
function handleRequest(request) {
  currentConnection = request.accept(subProtocol,
request.origin);
  currentConnection.on('message', handleMessage);
}

/**
 * Handles a message event on a socket connection.
 * @param {Object} message The message event object.
 */
function handleMessage(message) {
  // Echo back whatever was received.
  if (message.type === 'utf8') {
    currentConnection.sendUTF(message.utf8Data);
  } else if (message.type === 'binary') {
    currentConnection.sendBytes(message.binaryData);
  }
}

// Create a simple server that always returns 404 (not
found) to any request.
// (We're only going to use it to upgrade to the WebSocket
protocol.)
var simpleServer = http.createServer(function(request,
response) {
    response.writeHead(404);
    response.end();
});
simpleServer.listen(8080);

// Create a WebSocket server based on the simple server.
var socketServer = new WebSocketServer({
    httpServer: simpleServer,
    autoAcceptConnections: false
});
```

```
// Register the request event handler.
socketServer.on('request', handleRequest);
```

This example creates a WebSocket server based on a simple HTTP server. Whenever a connection request comes in on the socket server, a `request` event is dispatched on that object. In the `handleRequest` event handler you establish a WebSocket connection by accepting the request, and then register an event handler for `message` events. Whenever the client sends a message to the server, a `message` event is dispatched on the connection object. Your `handleMessage` event handler simply echoes back whatever data was received.

Save this script in a file called `example3-8-server.js`. To run it, type `node example3-8-server.js`.

Tip If you don't want to build a server yourself, there is a simple echo server running at `ws://echo.websocket.org`. Use that as the URL and do not specify a protocol. Note that if you do run the example this way, it will demonstrate that WebSockets are not bound to the Single Origin Policy, since the original page will be served from a local server but the WebSocket server is on an entirely different domain.

Next you need to build a client that can make use of the server. Your client should try sending various types of data and display anything that is echoed back from the server. Listing 3-9 shows a client that will connect to your server and run a set of tests.

Listing 3-9. A WebSocket Demonstration Class

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
  </head>
  <body>
    <h1>WebSockets Demonstration</h1>
    <div id="display"></div>
    <script>
function WebSocketDemo() {

  /**
   * The URL for the WebSocket server. There is a simple
echo server running at
   * ws://echo.websocket.org/ if you don't have a local
```

```
server running.
   * @private {string}
   */
  this.demoUrl_ = 'ws://localhost:8080/';

  /**
   * The protocol used by the server. If using the server
at echo.websocket.org
   * set this to null, as it does not have a protocol.
   * @private {string|Array<string>}
   */
  this.subProtocol_ = 'echo';

  /**
   * @private {WebSocket}
   */
  this.demoSocket_ = null;

  /**
   * A reference to the DOM element to use for displaying
messages.
   * @private {HtmlElement}
   */
  this.display = document.getElementById('display');

  /**
   * Displays a message on the page.
   * @param {string} messageText A simple string of text.
   * @param {*=} opt_messageData An optional set of data
to append to the text
   * string.
   * @private
   */
  this.displayMessage_ = function(messageText,
opt_messageData) {
    var messageData = opt_messageData ? opt_messageData
: '';
    var newParagraph = document.createElement('p');
    newParagraph.innerText = messageText + messageData;
    this.display.appendChild(newParagraph);
  };

  /**
```

```
   * Handles an error event on the demo socket object.
   * @private
   */
  this.handleError_ = function() {
    this.displayMessage_('An error occurred on the demo
connection.');
  };

  /**
   * Handles a close event on the demo socket object.
   * @private
   */
  this.handleClose_ = function() {
    this.displayMessage_('The demo connection was
closed.');
  };

  /**
   * Handles an open event on the demo socket object.
   * @private
   */
  this.handleOpen_ = function() {
    this.displayMessage_('The demo connection is open.');

    // Now that the socket is open, we can send data.
    this.sendDataAndClose_();
  };

  /**
   * Handles a message event on the demo socket object.
   * @param {MessageEvent} event The message event object.
   * @private
   */
  this.handleMessage_ = function(event) {
    this.displayMessage_('A message event has been
received from the server.');
      // Check the data type of the incoming data.
      if (event.data instanceof Blob) {
        this.displayMessage_('The data is a blob.');
      }
      if (event.data instanceof ArrayBuffer) {
        this.displayMessage_('The data is an
ArrayBuffer.');
```

```
      }
      this.displayMessage_('The data the server transmitted
is: ', event.data);
  };

  /**
   * Initializes the demo by creating a new connection and
registering event
   * handlers.
   * @private
   */
  this.initDemo_ = function() {
    // Open the socket.
    this.demoSocket_ = new WebSocket(this.demoUrl_,
this.subProtocol_);

    // Register the event handlers on the demo socket.
    this.demoSocket_.addEventListener('error',
this.handleError_.bind(this));
    this.demoSocket_.addEventListener('open',
this.handleOpen_.bind(this));
    this.demoSocket_.addEventListener('message',
 this.handleMessage_.bind(this));
    this.demoSocket_.addEventListener('close',
this.handleClose_.bind(this));
  };

  /**
   * Sends data to the server, then closes the socket.
   * @private
   */
  this.sendDataAndClose_ = function() {
    // Send a text string.
    this.demoSocket_.send('Hello world!');

    // Send a JSON-formatted string.
    var testObject = {
      message: 'hello world',
      active: true
    };
    var testObjectString = JSON.stringify(testObject);
    this.demoSocket_.send(testObjectString);
```

```
    // Send a Blob.
    var testBlob = new Blob(['some data']);
    this.demoSocket_.send(testBlob);

    // Done! Demo over. Close the socket after waiting for
a few seconds for
    // all of the messages to be sent and received. You
might need to adjust
    // this depending on the speed of your connection.
    setTimeout(function() {
      this.demoSocket_.close();
    }.bind(this), 5000);
  };

  /**
   * Runs the demonstration.
   */
  this.run = function() {
    this.initDemo_();
  };
}

// Create the demo and run it.
var myDemo = new WebSocketDemo();
myDemo.run();

    </script>
  </body>
</html>
```

In this example you've encapsulated the demonstration in a class constructor. Although you're only running the demonstration once, this is a good pattern to follow when building complex connections like this because it helps encapsulate their functionality. It also means you can easily instantiate more than one connection at once if you want.

Everything that was previously a function or variable in the global scope has been moved into the class. You've also added a new method, `sendDataAndClose_`, which demonstrates sending various types of data, and then closes the connection after a five-second delay. The `open` event handler calls `sendDataAndClose_`, so no data would be sent unless the connection was ready. Anything that the server sends will be displayed on the page.

Run this example and it will produce a result similar to the screenshot in Figure 3-2.

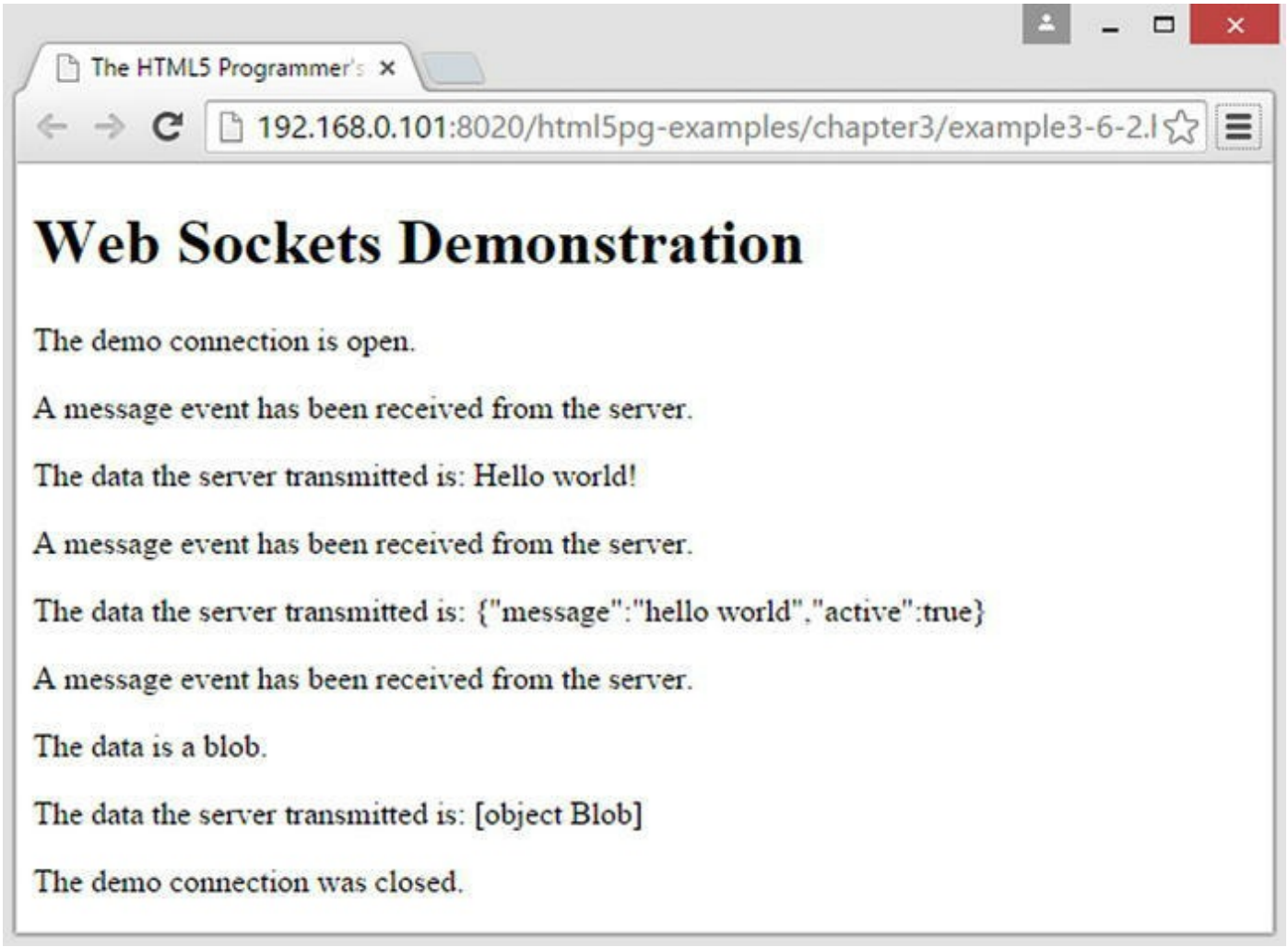


Figure 3-2. *The result of running Listing 3-9*

You can also induce an error in the WebSocket connection by specifying an invalid subprotocol value for `this.subProtocol_`:

```
/**
 * The protocol used by the server. If using the server at
echo.websocket.org
 * set this to null, as it does not have a protocol.
 * @private {string|Array<string>}
 */
this.subProtocol_ = 'invalid protocol';
```

The WebSocket server will not perform the upgrade and the socket connection will fail.

Cross Document Messaging/Web Messaging

SUPPORT LEVEL
Good

All modern browsers support these features.

WHATWG Living Standard: http://www.whatwg.org/specs/web-apps/current-work/multipage/web-messaging.html#crossDocumentMessages

W3C Draft: http://www.w3.org/TR/webmessaging/

When web browsers manufacturers started adopting JavaScript, it quickly became clear that security was going to be an important issue. Early on, Netscape introduced the Same Origin Policy, which dictates that a script can only access DOM content from the same origin as itself. If it weren't for this policy, malicious scripts from any domain could run on your browser, then read—and modify—all of the data to which the browser had access: rendered pages, history, cookies, even saved passwords.

There is no explicit standard for the Same Origin Policy, but it is based largely on RFC 6454, "The Web Origin Concept." (You can read this RFC at http://tools.ietf.org/html/rfc6454.) Roughly speaking, two resources have the same origin if their protocol (HTTP, HTTPS), host (e.g., `www.example.com`), and port (the default is port 80) all match.

Note Internet Explorer does not include the port in its origin determinations. Instead it uses the Security Zone that the resource falls within.

The Same Origin Policy is a cornerstone of web application security, and it is strictly enforced by browsers. Unfortunately it creates difficulties in building web applications that utilize multiple resources on different domains or even subdomains (e.g., `www.example.com` will have a different origin than `services.example.com`, even though they both have the same root domain of `example.com`). As a result, web developers have created many different

workarounds, some more hackish than others.

Web Messaging, also known as Cross Document Messaging, is one of the ways HTML5 provides a secure method of working within the Same Origin Policy, while allowing safe communication between resources from different origins. Specifically, the feature allows scripts in one frame to communicate with scripts in another frame using events that can be triggered at will.

The specification creates the new method `postMessage` on the browser's `window` object. You use this method to send a message to a target frame, which in turn causes a `message` event to be fired in that `window`. An event handler in the target frame can capture the event and receive the message.

The `postMessage` method takes two parameters:

- `message`: The message you want to transmit to the target frame.
- `origin`: The origin you expect the resources in the target frame to have. If the resources in the target frame do not have the specified origin, the event will not be dispatched.

The target frame will dispatch a `message` event when it receives a message. The resulting `event` object will have two important attributes:

- `Event.data`: This attribute will contain the message that was sent from the other window.
- `Event.source`: This attribute will contain the origin of the sending window. You should always double-check the origin of message sources to prevent accidentally capturing and processing events from unexpected (and possibly malicious) origins.

To create an example, you'll need two pages, which you should call the Main Page (Listing 3-10) and the Target Page (Listing 3-11). The Main Page will contain an iframe that loads the Target Page.

Listing 3-10. Cross-Domain Messaging, Main Page

```
<!DOCTYPE html>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
  </head>
  <body>
```

```
    <h1>Cross-Domain Messaging</h1>
    <iframe src="example3-8.html" id="targetFrame">
</iframe>
    <p><button id="clickme">Click to send a message to the
iframe.</button></p>
    <script>
// Message to send to the target window.
var strMessage = "This is a message sent from the main
window.";

// Reference to the button.
var clickme = document.getElementById("clickme");

// Reference to the target frame.
var targetFrame = document.getElementById("targetFrame");

// Add a click event handler to the button.
clickme.addEventListener("click", function() {
  // Send a message to the target frame.
  targetFrame.contentWindow.postMessage(strMessage, "*");
});

/**
 * Handle a cross domain message.
 * @param {Event} event The event object from the cross
domain message.
 */
function handleMessage(event) {
  // Create a message and show it to the user using an
alert.
  var strAlert = "Message event in the main window!\nThe
message was:\n";
  strAlert += event.data;
  alert(strAlert);
}

// Register the handleMessage event handler on the window.
window.addEventListener("message", handleMessage, false);
    </script>
  </body>
</html>
```

Listing 3-11. Cross-Domain Messaging, Target Page

```
<!DOCTYPE html>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
  </head>
  <body>
    <h1>Target iframe</h1>
    <script>
/**
 * Handles message events dispatched to this window.
 * @param {Event} event The event object from the cross
domain message.
 */
function handleMessage(event) {
  // Create a message and show it to the user using an
alert.
  var strAlert = "Message event in target iframe!\nThe
message was:\n";
  strAlert += event.data;
  alert(strAlert);

  // Post a message back to the parent frame.
  window.top.postMessage("This is a message from the
target iframe.", "*");
}

// Register the message event handler on the window.
window.addEventListener("message", handleMessage, false);
    </script>
  </body>
</html>
```

To run this example, save both pages in the same directory. Save the Target Page under the name "example3-8.html." When you load the Main Page into your browser, it will load the Target Page into the iframe. Run the example by clicking the text "Click to send a message to the iframe."

Upon load, both documents bind `message` event handlers to their `window` objects. When you click the button, the parent document sends a message to the iframe using `postMessage`. This triggers a `message` event in the iframe, which is handled there by the `handleMessage` event handler. This alerts the event data, and then posts

a message back to the parent document. This in turn triggers a `message` event in the parent window, which invokes the `handleMessage` event handler there and causes the second alert to occur.

In this example you are not taking advantage of the main purpose for this feature, which is to send messages from resources from different origins. If you have access to more than one origin, I encourage you to experiment with this example. Upload the pages to different origins (be sure to alter the URL of the iframe accordingly) and see if the results work as expected.

Note that in this example, you are not specifying the target origin in your calls to `postMessage`, nor are you checking the origin in your event handlers. This is inherently insecure and I strongly recommend against doing this in production code. You are only doing it here because this is an example, and the specific domain information will vary depending on how you are serving the files. I encourage you to modify these scripts so that they specify the target origin correctly and check the source origin according to your specific environment. Also, try setting them to different values to induce origin violations so you can see the results for yourself.

Web Storage

SUPPORT LEVEL
Excellent

All modern browsers support these features and have for the last three versions.

WHATWG Living Standard: `http://www.whatwg.org/specs/web-apps/current-work/multipage/webstorage.html`

W3C Draft: `http://www.w3.org/TR/webstorage/`

I've mentioned already that Hypertext Transfer Protocol is stateless. Partly this means that the server treats each request from a client independently from every other request. As a result there is no built-in mechanism for maintaining data across web page loads or reloads. For example, if the first page of an application is a login form and the user successfully logs in, there is no mechanism to maintain that session as the user navigates through the rest of the site. Or if you were building a shopping cart application, there is no way to carry the user's choices from one page to another.

Of course that makes for terrible user interaction, so in 1995—quite early in the

history of the Web—Netscape employee Lou Montulli created a specification for allowing small pieces of data to be communicated between the browser and the server using special HTTP requests. These pieces would be stored in the client, but the server could request them as needed. Montulli called these small pieces of data "magic cookies," and that is the origin of the term *HTTP Cookie*. Cookies enabled stateful communication on the web and quickly became a cornerstone for web applications. However, HTTP Cookies are a bit clunky. They are quite limited in size (4KB in most browsers) and are rather difficult to manage directly with JavaScript.

HTML5 introduced the concept of Web Storage as an alternative to HTTP Cookies. Web Storage allows for significantly more data to be stored in the browser (up to 5MB per origin in all browsers except Internet Explorer, which allows 10MB per origin). Web Storage also has a very simple key/value API, making it quite easy to use with JavaScript. Unlike HTTP Cookies, Web Storage is controlled entirely by the browser, and the server cannot access the contents directly. If you want to share Web Storage data with the server, your script will specifically have to transmit the data to the server.

Web Storage defines two different forms of storage: Session Storage and Local Storage. Session Storage, as its name suggests, only stores data for the current browser session. When the user closes their browser, the contents of Session Storage are cleared. Local Storage, on the other hand, is permanent. Once you store data in Local Storage it will stay there until you remove it, even if the user closes their browser or reboots their computer or device.

Like HTTP Cookies, Web Storage is limited by origin. Scripts from a given origin can only access Web Storage for that origin. Cross-origin access is not permitted. Unlike HTTP Cookies, you cannot set an expiration date or specify a path for Web Storage data.

Methods and Syntax

Web Storage defines two new objects in the global context: `sessionStorage` and `localStorage`. They both have the same methods:

- `getItem(key)`: Returns the data associated with the specified key.
- `removeItem(key)`: Removes the data associated with the specified key.
- `setItem(key, data)`: Stores the data in storage with the specified key.
- `clear()`: Clears the storage of all contents.

Using Web Storage is quite simple, as shown in Listing 3-12.

Listing 3-12. Using Web Storage

```
<!DOCTYPE html>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
  </head>
  <body>
    <h1>localStorage Example</h1>
    <script>
// Check to see if we've visited this page before.
var myValue = localStorage.getItem("test");
if (myValue == null) {
  alert('This is the first time you loaded this page! Now
reload this page.');
  localStorage.setItem("test", "true");
} else {
  alert('You have loaded this page before!');
}
    </script>
  </body>
</html>
```

This example first tests to see if you have visited this page before. If you haven't, it stores some data in Local Storage. When you reload the page, the data should still be present there.

One of the main limitations of Web Storage is that it can only store primitives (text, numbers, and booleans). More complex data like arrays or objects cannot be stored in Web Storage. However, if the desired data can be formatted as a JSON string, it can be serialized and then stored. Upon retrieval, the JSON string can be parsed and the data structure restored for use. It's not hard to write functions to handle this for you:

```
/**
 * Serializes and stores an object in session storage
under the specified key.
 * @param {string} key The key to store the data under.
 * @param {Object} value The object to serialize and
store.
 */
```

```
function setSessionObject(key, value) {
  sessionStorage.setItem(key, JSON.stringify(value));
}

/**
 * Retrieves, deserializes, and returns the object stored
in session
 * storage under the specified key.
 * @param {string} key The key that the object was stored
under.
 * @return {Object} The restored object.
 * /
function getSessionObject(key) {
  var value = sessionStorage.getItem(key);
  return value && JSON.parse(value);
}
```

Here the `setSessionObject` wraps the `sessionStorage.setItem` method and the `getSessionObject` method wraps the `sessionStorage.getItem` method. You could easily create similar functions for `localStorage` as well. But wouldn't it be great if both `sessionStorage` and `localStorage` had `getObject` and `setObject` methods, without having to use separate functions? Fortunately, that's quite easy to do thanks to the extendable nature of JavaScript and its inheritance model.

Without going into prototypal inheritance in detail, here's the secret: both the `sessionStorage` and `localStorage` objects inherit from the `Storage` abstract object. That means that any method available on the prototype object for `Storage` will be available to both `sessionStorage` and `localStorage`. So all you have to do is add your new methods to `Storage`:

```
/**
 * Serializes and stores an object in web storage under
the specified key.
 * @param {string} key The key to store the data under.
 * @param {Object} value The object to serialize and
store.
 */
Storage.prototype.setObject = function(key, value) {
  this.setItem(key, JSON.stringify(value));
};

/**
```

```
 * Retrieves, deserializes, and returns the object stored
in web storage under
 * the specified key.
 * @param {string} key The key that the object was stored
under.
 * @return {Object} The restored object.
 */
Storage.prototype.getObject = function(key) {
  var value = this.getItem(key);
  return value && JSON.parse(value);
};
```

Again, don't get too hung up on the syntax, just remember that both `sessionStorage` and `localStorage` are "children" of `Storage`, so any improvements you make to `Storage` will also be available to its children.

Listing 3-13 demonstrates using the new methods.

Listing 3-13. Using Custom Storage Methods

```
<!DOCTYPE html>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
  </head>
  <body>
    <h1>Web Storage Example</h1>
    <script>
/**
 * Serializes and stores an object in web storage under
the specified key.
 * @param {string} key The key to store the data under.
 * @param {Object} value The object to serialize and
store.
 */
Storage.prototype.setObject = function(key, value) {
  this.setItem(key, JSON.stringify(value));
};

/**
 * Retrieves, deserializes, and returns the object stored
in web storage under
 * the specified key.
```

```
 * @param {string} key The key that the object was stored
under.
 * @return {Object} The restored object.
 */
Storage.prototype.getObject = function(key) {
  var value = this.getItem(key);
  return value && JSON.parse(value);
};

// Create a simple test object.
var myObject = {
  test: true
};

// Create a simple test array.
var myArray = [1, 'two', true];

// Check session storage for the stored data.
if (sessionStorage.getItem('myObject') == null) {
  // First time here, so store the data.
  sessionStorage.setObject('myObject', myObject);
  sessionStorage.setObject('myArray', myArray);
  alert('Data stored. Reload the page to validate.');
} else {
  // We have been here before. Get values from storage and
test them.
  var newObject = sessionStorage.getObject('myObject');
  var newArray = sessionStorage.getObject('myArray');
  alert(myObject.test === newObject.test); // should alert
true.
  alert(myArray[1] === newArray[1]); // should alert true.
}
    </script>
  </body>
</html>
```

The first step in this example is to extend `Storage` with your new methods. Next, create an object and an array for testing the new methods. When you load the page it checks for the stored value, and if it isn't there it stores the object and array using your new methods. If it is there, it gets them using your new methods and then tests to see that the values are the same.

Privacy and Web Storage

The possibility of extensive, permanent data storage within a browser raises serious privacy concerns. Though Web Storage is limited by the Single Origin Policy, the same techniques that can be used for setting third-party HTTP Cookies can also be used for setting third-party Web Storage data. And while all browsers offer users a great deal of control over the HTTP Cookies that they store, most browsers have not yet extended those features to Web Storage. In fact, Web Storage is one of the methods used to create so-called Evercookies (you can read about Evercookies at `http://samy.pl/evercookie/`).

As with HTTP Cookies, you should not regard Web Storage as secure. Do not store any sensitive information, such as passwords, in Web Storage.

Most browsers implement some form of private browsing. There's no standard governing what "private browsing" entails, but in most cases it means that all client-side storage is limited to the current session. This includes Web Storage. If the user is using the browser's private browsing feature, any data stored using Web Storage will be wiped out when the user closes that tab, even if you stored it using `localStorage`. As a result there is no guarantee that what you store in `localStorage` will be there the next time the user returns to your application, so bear that in mind if your application relies heavily on `localStorage`.

Drag and Drop

SUPPORT LEVEL
Excellent

All modern browsers support these features and have for at least the last three versions.

WHATWG Living Standard: `http://www.whatwg.org/specs/web-apps/current-work/multipage/dnd.html`

W3C Draft: `http://www.w3.org/TR/html5/editing.html#dnd`

One of the most common interactions in a graphical interface is dragging elements from one place and dropping them in another. Unfortunately there was no easy way to achieve

this basic interaction with HTML and JavaScript. You could do it, but it was quite difficult and required extensive scripting, or the use of an existing library such as jQuery UI (see `http://jqueryui.com/draggable/` for an example).

Now the HTML5 specification brings native drag-and-drop interactions to the browser. The process is event driven, and follows these simple steps:

- Declare one or more objects as draggable, and attach desired event handlers.
- Attach `drop` event handlers to target elements.
- As the user drags items and drops them on targets, the various events are fired and your handlers are called.

The specification includes several new events, a `draggable` property for HTML elements, and a `dataTransfer` object for safely communicating data between events.

There are typically two reasons why you would want to build a drag-and-drop interaction into your application. The simplest (and probably the most common) is when the drag-and-drop operation is a representation of another process that will be performed by the application. In this situation, the items being dragged and the targets upon which they're dropped aren't themselves important—they're just elements in the user interface. Based on the results of the drag-and-drop operation, something else will happen behind the scenes. An example of this is an interface for a shopping cart system. You drag items from the page into your shopping cart, but the things you're dragging, and the target cart, are just arbitrary HTML elements that have been styled in such a way that the user recognizes them. Behind the scenes, the data is being manipulated based on the drag-and-drop operations: the data structure for the cart is being changed as items are added or removed, and so forth.

The other situation is where the items being dragged and/or the targets on which they're being dropped are themselves important. In this situation what is being dragged actually matters, because it will be processed itself. An example of this is a visual clipboard, where you can highlight text in a document and then drag it to the clipboard. The text is actually transferred from one place to another in the DOM via the drag-and-drop process.

The HTML5 drag-and-drop specification handles both situations easily, and I will show you both in my examples. Before diving into these, take a look at the process in detail.

The `draggable` Property

The first component of the Drag and Drop specification is the new `draggable` property. This property is set on any HTML element in the DOM you wish to be draggable. If an element has the `draggable` property set to `true`, the browser will initiate a drag-and-drop sequence from that element if the user holds down the mouse button while the pointer is over the element and then moves the pointer.

The `draggable` property can be set to `true` (indicating the item is draggable), `false` (indicating the item cannot be used to initiate a drag sequence), or `auto` (indicating the browser's default rules apply).

The exception is selected text anywhere in the DOM, including form fields such as `input` and `textarea` fields. Selected text can always initiate a drag sequence.

Drag-and-Drop Events

There are several new drag-and-drop events:

- `dragstart`: Dispatched from the element being dragged.
- `dragenter`: Dispatched from any element when a draggable item is dragged into it.
- `dragover`: Dispatched continuously from any element as long as a draggable item is over it. Note that this event fires continuously regardless of whether or not the draggable item is moving.
- `dragleave`: Dispatched from an element when a draggable item leaves its boundary.
- `drag`: Dispatched from the element being dragged throughout the drag sequence. Like `dragover`, this event is fired continuously regardless of whether the pointer is being moved.
- `dragend`: Dispatched from the element being dragged when the mouse button is released.
- `drop`: Dispatched from an element when the user drops a draggable item on it by releasing the mouse button.

Like other DOM events, you can add event handlers for drag-and-drop events to any desired element. Remember, though, that drag-and-drop events will only fire while a drag sequence is underway.

One important quirk about drag-and-drop events is how you specify drop targets. The HTML5 specification defines a `dropzone` attribute as a counterpart to the

draggable attribute. The `dropzone` attribute is supposed to indicate which elements are valid drop targets. The `dropzone` attribute is not widely implemented, so instead you have to indicate valid drop targets by manipulating events.

Generally speaking, the majority of elements in the DOM should not be valid drop targets, so the default action of the `dragover` event is to cancel drops. As a result, to indicate a valid drop target you have to cancel the default action of the `dragover` event by calling the `preventDefault()` method on the `event` object within the event handler.

The `dataTransfer` Object

The final piece of the drag-and-drop puzzle is the `dataTransfer` object. All of the drag-and-drop events can be handled with standard event handlers, and those event handlers will receive an `event` object as a parameter. One of the properties on drag-and-drop `event` objects is the `dataTransfer` object. This object is used to control the appearance of the drag-and-drop helper (the ghosted visual element that follows the cursor during the drag-and-drop operation), to indicate what the drag-and-drop process is doing, and to easily transfer data from the `dragstart` event to the `drop` event.

The `dataTransfer` object has the following methods:

- `Event.dataTransfer.addElement(HtmlElement)`: Specify the source element of the drag sequence. This affects where the `drag` and `dragend` events are fired from. Ordinarily you probably won't need to change this.
- `Event.dataTransfer.clearData(opt_DataType)`: Clear the data associated with a specific `DataType` (see `setData` in this list). If the `DataType` is not specified, all data is cleared.
- `Event.dataTransfer.getData(DataType)`: Get the data associated with a specific `DataType` (see `setData`, next).
- `Event.dataTransfer.setData(DataType, data)`: Associates the specified `data` with the `DataType`. Valid `DataTypes` depend on the browser. Internet Explorer only supports `DataTypes` of `text` and `url`. Other browsers support standard MIME types and even arbitrary types. The `data` has to be a simple string but could conceivably be a JSON-formatted serialized object.

- `Event.dataTransfer.setDragImage(HtmlElement, opt_offsetX, opt_offsetY)`: Sets the drag helper image to the specified HTML element. By default the upper left corner of the helper image is placed under the mouse pointer, but that can be offset by specifying the optional parameters `opt_offsetX` and `opt_offsetY`, in pixels. This method is not available in Internet Explorer and apparently never will be—see http://connect.microsoft.com/IE/feedback/details/8datatransfer-prototype-setdragimage-method.

The `dataTransfer` object also has the following properties:

- `Event.dataTransfer.dropEffect`: The drop effect that is being performed by the drag-and-drop sequence. Valid values are `copy`, `move`, `link`, and `none`. This value is automatically initialized in the `dragenter` and `dragover` events based on what interaction the user has requested through a combination of mouse actions and modifier keys (e.g., Ctrl-drag, Shift-drag, Option-drag, etc.). These are platform dependent. Only values specified by `effectAllowed` (see next) will actually initiate drag-and-drop sequences.
- `Event.dataTransfer.effectAllowed`: Which `dropEffects` are permitted for this drag-and-drop sequence. Valid values and the effects they permit are as follows:
    - `copy`: Allow a copy `dropEffect`.
    - `move`: Allow a move `dropEffect`.
    - `link`: Allow a link `dropEffect`.
    - `copyLink`: Allow both a copy and a link `dropEffect`.
    - `copyMove`: Allow both a copy and a move `dropEffect`.
    - `linkMove`: Allow both a link and a move `dropEffect`.
    - `all`: All `dropEffects` are permitted. This is the default value.
    - `none`: No `dropEffects` are permitted (the item cannot be dropped).
- `Event.dataTransfer.files`: Contains a list of all the files

available on the data transfer. Will only have values if files are being dragged from the desktop to the browser.

- `Event.dataTransfer.types`: Contains a list of all the `DataTypes` that have been added to the `dataTransfer` object, in the order in which they were added.

Drag-and-Drop API Examples

Listing 3-14 produces the simplest thing imaginable: a set of draggable boxes that can be dropped on a single target. As the boxes are dropped on the target, a counter increases to show the number of times a drop has happened.

Listing 3-14. A Simple Drag-and-Drop Interface

```
<!DOCTYPE html>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
    <style>
.draggable {
  margin: 5px;
  width: 100px;
  height: 100px;
  background-color: #ccc;
  border: 1px solid #000;
  display: inline-block;
}

.target {
  border: 10px solid #000;
  width: 315px;
  height: 100px;
  margin-left: 5px;
  margin-top: 50px;
}

.target.over {
  border: 10px solid green;
}
     </style>
```

```
  </head>
  <body>
    <div class="draggable" draggable='true'></div>
    <div class="draggable" draggable='true'></div>
    <div class="draggable" draggable='true'></div>
    <div class="target"></div>
    <script>

// Get a reference to the drop target.
var dropTarget = document.querySelector('.target');

// Add a dragenter event handler to the drop target.
dropTarget.addEventListener('dragenter', function(event) {
  // Add the 'over' CSS class to the drop target. This
lets the user know that
  // they have dragged something over a valid drop target.
  this.classList.add('over');
}, false);

// Add a dragleave event handler to the drop target.
dropTarget.addEventListener('dragleave', function(event) {
  // Remove the 'over' CSS class.
  this.classList.remove('over');
}, false);

// Add a dragover event handler to the drop target.
dropTarget.addEventListener('dragover', function(event) {
  // Prevent the default event action.
  event.preventDefault();
}, false);

// A counter that indicates how many times something has
been dropped onto the
// drop target.
var counter = 1;

// Add a drop event handler to the drop target.
dropTarget.addEventListener('drop', function(event) {
  // Update the counter and remove the 'over' CSS class.
  this.innerHTML = counter;
  this.classList.remove('over');
  counter++;
}, false);
```

```
    </script>
  </body>
</html>
```

In the script, all you did was register `dragenter`, `dragleave`, `dragover`, and `drop` event handlers on the drop target. On `dragenter` you add a CSS class to the element, and on `dragleave` you remove the CSS class. This gives visual feedback that the user has successfully dragged the element over a target that can receive it. Then you prevent the default action of the `dragover` event, to prevent drop events. In the drop event handler you update the `innerHTML` of the target element and increment the counter. You also remove the visual feedback CSS class, since at this point you'll be terminating the drag-and-drop sequence and a `dragleave` event will not fire.

This example works great in Internet Explorer and Chrome. It does not work at all in Firefox. This is because Firefox requires that the `dataTransfer` object be initialized on `dragstart` by specifying some data—any data. To update your script, you have to add a `dragstart` event handler to each of your draggable elements and set some arbitrary data within them so Firefox will initiate drag sequences from them. Listing 3-15 has the necessary changes to the script (the surrounding HTML and CSS remain the same as in Listing 3-14).

Listing 3-15. Drag-and-Drop Script Updated to Work in Firefox

```
// Get a reference to all of the draggable objects.
var draggables = document.querySelectorAll('.draggable');

// On each draggable element intialize the dataTransfer object on
dragstart so
// Firefox will initiate drag events with them.
for (var i = 0; i < draggables.length; i++) {
  currEl = draggables[i];
  currEl.addEventListener('dragstart', function(event) {
    event.dataTransfer.setData('text', 'anything');
  }, false);
};

// Get a reference to the drop target.
var dropTarget = document.querySelector('.target');

// Add a dragenter event handler to the drop target.
dropTarget.addEventListener('dragenter', function(event) {
  // Add the 'over' CSS class to the drop target. This
lets the user know that
  // they have dragged something over a valid drop target.
```

```
  this.classList.add('over');
}, false);

// Add a dragleave event handler to the drop target.
dropTarget.addEventListener('dragleave', function(event) {
  // Remove the 'over' CSS class.
  this.classList.remove('over');
}, false);

// Add a dragover event handler to the drop target.
dropTarget.addEventListener('dragover', function(event) {
  // Prevent the default event action.
  event.preventDefault();
}, false);

// A counter that indicates how many times something has
been dropped onto the
// drop target.
var counter = 1;

// Add a drop event handler to the drop target.
dropTarget.addEventListener('drop', function(event) {
  // Update the counter and remove the 'over' CSS class.
  this.innerHTML = counter;
  this.classList.remove('over');
  counter++;
}, false);
```

You'll notice that the code uses `querySelectorAll` to get a reference to all of your draggable elements. Then it loops through each of those elements in a `for` loop and applies the `dragstart` event listener to each one. (Another way to do this would have been to delegate the dragstart event handler to a containing element.) Now the elements are draggable in Firefox, and the example works in that browser the same as in the others.

In practice, you'll probably be initializing the data for a drag-and-drop sequence anyway. I mentioned earlier an example of a visual clipboard, which you can see in Listing 3-16.

Listing 3-16. A Visual Clipboard

```
<!DOCTYPE html>
<html>
```

```
    <head>
       <title>The HTML5 Programmer's Reference</title>
       <style>
p {
  margin-bottom: 0;
}
div#dropTarget {
  width: 300px;
  height: 300px;
  border: 10px solid black;
}
div#dropTarget.over {
  border: 10px solid green;
}
.draggable {
  width:100px;
  height: 100px;
  background-color: #ccc;
}
       </style>
    </head>
    <body>
      <p>Type some text here, then highlight it and drag it
to the clipboard below.</p>
      <textarea id="dragSource"></textarea>
      <p>Clipboard</p>
      <div id="dropTarget"></div>
      <script>
// Get references to our drag source and drop target.
var dragSource = document.getElementById('dragSource');
var dropTarget = document.getElementById('dropTarget');

// Add a dragstart event handler to the dragsource
element.
dragSource.addEventListener('dragstart', function(event) {
  // Initialize the dataTransfer object with the current
text.
  event.dataTransfer.setData('text', this.value);
}, false);

// Add a dragenter event handler to the target.
dropTarget.addEventListener('dragenter', function(event) {
```

```
  // Add the 'over' CSS class to the element.
  this.classList.add('over');
}, false);

// Add a dragleave event handler to the target.
dropTarget.addEventListener('dragleave', function(event) {
  // Remove the 'over' CSS class from the element.
  this.classList.remove('over');
}, false);

// Add a dragover event handler to the target.
dropTarget.addEventListener('dragover', function(event) {
  // Prevent the default action of the dragover event.
  event.preventDefault();
}, false);

// Finally, add a drop event handler to the target.
dropTarget.addEventListener('drop', function(event) {
  // Append the text in the dataTransfer object to the
clipboard.
  this.innerHTML = event.dataTransfer.getData('text');
  // Remove the 'over' CSS class from the element.
  this.classList.remove('over');
}, false);
    </script>
  </body>
</html>
```

In this example you've created a simple `textarea` where you can enter some text. You can then highlight the text and drag it to the clipboard. Behind the scenes, the code sets the text as data on the `dataTransfer` object during the `dragstart` event, and then gets the text from the `dataTransfer` object on the `drop` event.

This example works great in almost all browsers, except once again Firefox has a small problem. When you drop text onto the clipboard area, Firefox fires a default action on `drop` that tries to update the URL of the page to the text that was dropped. To prevent this, you will need to call `event.preventDefault()` in the `drop` event handler. By adding that line, the example will work the same in all browsers.

For a final example, consider the need to restrict the movement of the draggable item to a specific area. You don't want to allow it to leave a containing element. Or perhaps you want to limit the movement to a single axis. Unfortunately, the HTML5 Drag and Drop API doesn't provide a ready-made solution for this fairly common use case, but you can build one with the tools it does provide.

The core of the problem is you have no way to limit the mouse pointer with JavaScript. This makes sense; being able to manipulate mouse pointer actions with JavaScript would pose a large security risk. The way the drag-and-drop sequence is set up, wherever the pointer goes, the helper image follows, so if you can't limit the pointer, you can't limit the location of the helper image.

This means the first thing you'll have to do is remove the default helper image using `dataTransfer.setDragImage()`, and that means this example won't work in Internet Explorer because it doesn't implement that method. But the example does work in other browsers, and it's a worthwhile example to demonstrate some more complex interactions with the API.

The next step is to build your own helper image that can be manipulated as needed. Once that's done, it's just a matter of listening to events.

Listing 3-17 provides the full example.

Listing 3-17. Limiting Drag and Drop to a Specific Region

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
.container {
  width:200px;
  height:500px;
  border: 1px solid #000;
  position: relative;
}
#dragTarget {
  height: 20px;
  background-color: #ccc;
}
.drag-helper {
  opacity: 0.5;
  position: absolute;
  width: 200px;
  height: 20px;
  background-color: #ccc;
}
.hidden {
  display: none;
}
```

```
      </style>
    </head>
    <body>
      <div class="container">
        <div id="dragTarget" draggable="true">Drag me!
</div>
      </div>
      <div id="helper" style="width: 1px;height: 1px;">
</div>
      <script>
// Get references to the drag target and container.
var dragTarget = document.getElementById('dragTarget');
var dragContainer = document.querySelector('.container');

// This variable will hold the new helper when we build
it.
var dragHelper;

// Add a dragstart event handler to the drag target.
dragTarget.addEventListener('dragstart', function(event) {
  // Initialize the dataTransfer object for Firefox.
  event.dataTransfer.setData('text', 'Fix for Firefox');

  // Replace the default drag image with a small,
transparent DIV.
  var dragImage = document.getElementById('helper');
  event.dataTransfer.setDragImage(dragImage, 0, 0);

  if (dragHelper == null) {
    // Create our own drag helper by cloning the target
element. Note that when
    // we clone the element we need to do some cleanup,
like removing the clone's
    // id attribute (so we do not introduce duplicate ids
even temporarily) and
    // making not draggable.
    dragHelper = this.cloneNode(true);
    dragHelper.id = '';
    dragHelper.classList.add('drag-helper');
    dragHelper.draggable = false;
    dragContainer.appendChild(dragHelper);
  } else {
    // We've already created a clone, so let's just use
```

```
it.
    dragHelper.classList.remove('hidden');
  }
}, false);

// Add a dragover event handler to the drag container.
dragContainer.addEventListener('dragover', function(event)
{
  // Prevent the default action of the event.
  event.preventDefault();

  // Move the helper to the desired location.
  if (event.clientY < 485) {
    dragHelper.style.top = event.clientY + 'px';
  }
}, false);

// Add a dragend event handler to the target.
dragTarget.addEventListener('dragend', function(event) {
  // Dragging is done, so hide the clone.
  dragHelper.classList.add('hidden');
}, false);

// Add a drop event to the drag container.
dragContainer.addEventListener('drop', function(event) {
  event.preventDefault();
}, false);
    </script>
  </body>
</html>
```

You'll do all of your setup in the `dragstart` event handler. First is the fix for Firefox, otherwise the example wouldn't work in that browser. Then you hide the default helper image by getting a reference to a small transparent `div` and using that as the new helper image. Next you clone the target element and set the clone up to be your new helper, and append it to the DOM.

In your `dragover` event handler, all you have to do is move the helper to the desired position. Remember that since the `dragover` event fires continuously throughout the drag process, you should keep its event handler as lightweight as possible. That's why you're using a cached reference to the drag helper, so you don't have to query for it every time the event fires. As you use the example you'll notice that because you're only listening for `dragover` events in the containing element, when

you move the pointer outside of that element the helper will stop moving. You could have delegated the `dragover` event handler to the `body` element, which would allow the user to move the element from anywhere on the page.

You hide the custom helper in your `dragend` event handler. This is done so the helper will be hidden regardless of where the user releases the mouse button. This is a common use of the `dragend` event.

As mentioned, this example won't work in Internet Explorer, so while this example is interesting it's not very practical for the majority of situations. Unfortunately there's no way to polyfill the `setDragImage` method, either. So if this is the sort of interaction you need and you have to support Internet Explorer, you probably need to look for another solution besides the HTML5 native Drag and Drop API.

Web Workers

SUPPORT LEVEL
Good

All modern browsers support these features and have for at least the last two versions.

WHATWG Living Standard: `http://www.whatwg.org/specs/web-apps/current-work/multipage/workers.html`

W3C Draft: `http://dev.w3.org/html5/workers/`

One of the biggest criticisms leveled at web-based applications is speed: they're just not fast enough, particularly on mobile devices. And as much as I'm a fan of web technologies (and building applications with those technologies), speed is admittedly a legitimate concern.

There are many reasons why web applications can run slowly, but one of the main issues is that scripts running in the web browser can only do one thing at a time. For example, consider the common example of fetching and parsing (or otherwise manipulating) a large file from the server. Using `XMLHttpRequest` it's easy to set up an asynchronous request that will load the file and then execute a callback function when ready. Up until the callback function is called, your scripts can perform other tasks. But once the callback function begins execution, everything has to wait while it handles the data in the way you specified. Your scripts can't update the UI or respond to

user interaction or anything. This can be very frustrating for users.

The asynchronous environment of web browsers helped us create web applications that were more responsive, but still didn't allow us to perform multiple tasks at once. The HTML5 specification addresses this limitation with Web Workers, which give us the ability to create and manipulate multiple separate tasks simultaneously.

Under the hood, Web Workers give us access to the multithreading capabilities of modern browsers and their host systems. A *thread* is a combination of operating system and program resources needed to perform a specific task and manage the status of that task by starting, pausing, and stopping it and handling its completion. Modern operating systems running on modern hardware with multiple processor cores can handle many threads simultaneously. When you create a new Web Worker, the browser spawns a separate thread for that task, and it executes at the same time as the main browser thread.

Multithreading is a powerful tool, and like all powerful tools it has some dangers. Web Workers have some important restrictions designed to reduce those dangers:

- A Web Worker runs in its own independent JavaScript context. It has no direct access to anything in any of the other execution contexts like other Web Workers, or the main JavaScript thread.
- Communication between Web Worker contexts and the main JavaScript thread is done via a `postMessage` interface similar to that used by Web Messaging. This enables you to pass data into and out of Web Worker contexts, but because all contexts are independent, any data passed between contexts is copied, not shared.
- A Web Worker cannot access the DOM. The only DOM methods available to a Web Worker are `atob`, `btoa`, `clearInterval`, `clearTimeout`, `dump`, `setInterval`, and `setTimeout`.
- Web Workers are bound by the Same Origin Policy, so you cannot load a worker script from a different origin than the original script.

These are strong restrictions (particularly the lack of access to the DOM), but they help make Web Workers a safer tool for you to use. If you've ever built multithreaded applications in other languages, you're probably familiar with all of the dangers inherent in that capability: concurrency, security, and so forth. Because of their restrictions, Web Workers are mostly free of those dangers.

Another important feature of Web Workers is the fact that you have to manage your workers yourself. You are responsible for creating them, starting them, stopping them, and disposing of them when their tasks are done. Because Web Workers consume host

system resources, it's important that you manage them correctly to avoid impacting the performance of the entire system.

Creating Web Workers

Creating and managing Web Workers follows three basic steps:

1. Create the new Web Worker.
2. Attach a `message` event handler to the new worker, assuming you are expecting it to communicate results. You should also attach an `error` event handler so that your script can react to any errors that happen during the worker's execution.
3. Start the worker instance by posting a message to it. This will cause the worker to start running, and will also trigger a `message` event within it so the worker can process the message you just posted.

Step 1 is easy. The Web Workers specification creates a new `Worker` constructor in the global context. You create a new instance of a `Worker` by specifying a JavaScript program for it to load and execute:

```
var myNewWorker = new Worker('my-new-worker.js');
```

The file `'my-new-worker.js'` must be a valid JavaScript file, and it will be loaded as soon as the worker is created.

The Worker instance `myNewWorker` will publish `message` and `error` events as it goes about its business, so you can attach event handlers for those events. Listing 3-18 shows the basic pattern with stubbed functions.

Listing 3-18. Stubbed Error and Message Event Handlers for a Web Worker

```
/**
 * Handles an error event from web worker.
 * @param {WorkerErrorEvent} event The error event object.
 */
function handleWorkerError(event) {
  // Handle the error here.
  console.warn('Error in web worker: ', event.message);
}
```

```
/**
 * Handles a message event from a web worker.
 * @param {WorkerMessageEvent} event The message event
object.
 */
function handleWorkerMessage(event) {
  // Handle the message here.
  console.log('Message from worker: ', event.data);
}

// Create a new worker.
var myNewWorker = new Worker('my-new-worker.js');

// Register error and message event handlers.
myNewWorker.addEventListener('error', handleWorkerError);
myNewWorker.addEventListener('message',
handleWorkerMessage);
```

In this basic example you have simple functions for handling the `error` and `message` events that simply display the results in the JavaScript console. You register them as handlers using the `addEventListener` method on the `Worker` instance `myNewWorker`. Note that you should always register your event handlers before starting the worker. If you start the worker and then register the event handlers, something could happen in the few milliseconds it takes to complete the registration and you could miss a message or an error.

To start the worker, simply post a message to it using the `postMessage` method:

```
myNewWorker.postMessage('start');
```

When you post the message to the worker instance, it will begin executing the script, and will also trigger a `message` event within the worker's execution context for the start message. (Note that while the `Worker.postMessage` method is similar to the `window.postMessage` method you would use for Web Messaging, it does not have the optional domain parameter of the latter.)

Inside a Web Worker

Inside a Web Worker, the environment for your script is a little different than in the main execution context. As mentioned, the Web Worker has no access to the DOM, so any attempt to access the `window` object or its children (such as the `document` object

or any element in the DOM) will fail. Web Workers do have access to the following standard properties and methods:

- The DOM methods `atob`, `btoa`, `clearInterval`, `clearTimeout`, `dump`, `setInterval`, and `setTimeout`.
- The `XMLHttpRequest` constructor, so Web Workers can perform asynchronous network tasks.
- The `WebSocket` constructor, so Web Workers can create and manage WebSockets (as of this writing, Firefox does not enable `WebSocket` for Web Workers; however, this feature is being implemented and you can track its status at https://bugzilla.mozilla.org/show_bug.cgi?id=504553)
- The `Worker` constructor, so Web Workers can spawn their own workers (which are referred to as *subworkers*). As of this writing, Chrome and Safari do not implement the `Worker` constructor for Web Workers. There is a bug filed for Chrome at https://code.google.com/p/chromium/issues/detail?id=31666 and for Safari's WebKit at https://bugs.webkit.org/show_bug.cgi?id=22723. Internet Explorer does support subworkers as of version 10.
- The `EventSource` constructor, so Web Workers can subscribe to Server-sent Event streams. This appears to be a nonstandard feature, but seems to be available in all major browsers as of this writing.
- A special subset of the `Navigator` properties, available through the `navigator` object:
    - `navigator.language`: Returns the current language the browser is using.
    - `navigator.onLine`: Returns a boolean indicating whether or not the browser is online.
    - `navigator.platform`: Returns a string indicating the platform of the host system.
    - `navigator.product`: Returns a string with the name of the current browser.

        - `navigator.userAgent`: Returns the user agent string for the browser.

          The implementation of these properties varies from browser to browser, so it might be better to pass needed `Navigator` information into the Web Worker from the main thread.
    - A special subset of `Location` properties, available on the `location` object:
        - `location.href`: The full URL of the script being executed by the Web Worker.
        - `location.protocol`: The protocol scheme of the URL of the script being executed by the Web Worker, including the final ":".
        - `location.host`: The host part of the URL (the hostname and port) of the script being executed by the Web Worker.
        - `location.hostname`: The hostname part of the URL of the script being executed by the Web Worker.
        - `location.port`: The port part of the URL of the script being executed by the Web Worker.
        - `location.pathname`: The initial '`/`' followed by the path of the script being executed by the Web Worker.
        - `location.search`: The initial '`?`' followed by the parameters (if any) of the URL of the script being executed by the Web Worker.
        - `location.hash`: The initial '`#`' followed by the fragment identifier (if any) of the URL of the script being executed by the Web Worker.
    - There is also a method `location.toString()` that simply returns `location.href`.

The Web Worker execution context also has one new method available to it: `importScripts`. The `importScripts` method takes a comma-separated list of one or more JavaScript file names that will be loaded and executed in order. For example, this line

```
importScripts('script1.js', 'script2.js',
'subdirectory/script3.js');
```

will load and execute the three scripts specified, in order. Relative URLs are resolved as relative to the URL of the script that was specified when you created the Web Worker instance. The `importScripts` method is also bound by the Same Origin Policy, so you cannot import scripts from a different origin than the origin that served the parent script for the Web Worker instance.

The `importScripts` method is a blocking method, meaning each script will be loaded and executed, in order, and the worker will not continue to the next line until the last script has finished loading and executing. If one of the scripts fails to load due to a network problem, or it loads but fails to run due to an internal error, the Web Worker will stop executing and publish an `error` event.

Scripts loaded with `importScripts` are executed in the same context as the Web Worker. They cannot access the DOM, but they do have access to all of the standard properties and methods listed above as well as the `importScripts` method, so it is possible for imported scripts to import other scripts.

When a Web Worker is started, it follows these steps:

- It executes the script from start to finish, including any asynchronous tasks (such as `XMLHttpRequest` calls).
- If part of its execution was to register a `message` event handler, it then goes into a wait loop for incoming messages. The first message it receives will be the message that was posted to start the worker. The worker will remain in wait mode until you manually terminate it, or it terminates itself.
- If no `message` event handlers were registered, the worker thread will terminate automatically.

This is an important point: if your Web Worker registers a `message` event handler, it will remain in wait mode forever unless you terminate it. Again, because Web Workers consume system resources, you should be sure to terminate any unneeded workers. You can terminate a worker in one of two ways:

- You can call the `terminate` method on the Worker instance: `myWebWorker.terminate();`.
- The Web Worker can terminate itself by calling its `close` method:`self.close();`.

Either method stops the worker immediately.

A Simple Example of a Web Worker

Listing 3-19 expands on the stubbed example in Listing 3-18.

Note The examples in this section will need to be run from a server.

Listing 3-19. Creating A Web Worker

```
<!DOCTYPE html>
<html>
    <head>
       <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
      <h1>Web Workers</h1>
      <div id="message-box"></div>
      <script>
// Get a reference to the target element.
var messageBox = document.getElementById('message-box');

/**
 * Handles an error event from web worker.
 * @param {WorkerErrorEvent} event The error event object.
 */
function handleWorkerError(event) {
  console.warn('Error in web worker: ', event.message);
}

/**
 * Handles a message event from a web worker.
 * @param {WorkerMessageEvent} event The message event
object.
 */
function handleWorkerMessage(event) {
  messageBox.innerHTML = 'Message received from worker:
' + event.data;
}
```

```
// Create a new worker.
var myNewWorker = new Worker('web-worker.js');

// Register error and message event handlers on the
worker.
myNewWorker.addEventListener('error', handleWorkerError);
myNewWorker.addEventListener('message',
handleWorkerMessage);

// Start the worker.
myNewWorker.postMessage('begin');
     </script>
   </body>
</html>
```

As before, create two event handlers, one for an error event and one for a message event. The error event handler simply logs the error to the console, while the message event handler appends the message text to the DOM. Then create a new Web Worker, register your event handlers on it, and finally, post a message to it to start it.

Listing 3-20 shows the code for the worker itself.

Listing 3-20. A Trivial Web Worker Script

```
/**
 * Handles a message event from the main context.
 * @param {WorkerMessageEvent} event The message event.
 */
function handleMessageEvent(event) {
  // Do something with the message.
  console.log('Worker received message:', event.data);

  // Send the message back to the main context.
  self.postMessage('Your message was received.');
}

// Register the message event handler.
self.addEventListener('message', handleMessageEvent);
```

This Web Worker creates a message event handler that logs the message to the console. It then sends a confirmation message back to the parent thread, and registers the event handler on the execution context.

When you run this example, it will pass messages back and forth, but won't demonstrate the true power of Web Workers, which is that they execute at the same time as the main thread.

Common Use Cases

Web Workers allow you to restructure your applications in such a way that you have a single main thread that handles the UI, and any other intensive or asynchronous action is handled by Web Worker threads. Good examples include:

- Asynchronous Activities: Because Web Workers have access the `XMLHttpRequest` constructor as well as WebSockets and `theimportScripts` method, they can be used to load and parse data, or (even better) to send large amounts of data back to the server.
- Computation-Intensive Activities: Anything that requires a great deal of computation is an ideal candidate for a Web Worker. Cryptography is a great example, as are physics engines for games.
- Image Processing: If you have a large amount of data to process from a canvas element, you can make use of Web Workers to handle the number crunching.
- Divide and Conquer: I've already mentioned using Web Workers to handle processing of large amounts of data. If you can divide the data in question up into smaller pieces, you can give each piece its own Web Worker to process, thus making things go even faster.

Summary

In this chapter I've covered several of the HTML5 JavaScript APIs. Using these new APIs, your applications can:

- Communicate more efficiently and securely with the server. Rather than relying on just `XmlHttpRequest`, your applications can now subscribe to Server-sent Event streams, or even set up two-way communication with a server using WebSockets. You can also use Cross Document Messaging for more secure communications

between script origins.

- Store information more efficiently on the client. Using the new Web Storage features, your applications can easily store and retrieve information, including serialized objects and data structures.
- Efficiently implement drag-and-drop interactions using the new Drag and Drop API. Dragging and dropping items is a very common user interaction metaphor, and now it's easier to accomplish with the new API.
- Create and manage threads. Using Web Workers, your applications can now be multithreaded.

Using these new APIs, your applications can be more efficient and easier to use as well as simpler to create and maintain.

In the next chapter you will dive into one of the most exciting features of HTML5: the `canvas` element.

CHAPTER 4

Canvas

When HTML5 was first announced, the feature that drew the most excitement was probably the new `canvas` element—an area on the page upon which you can draw bitmap graphics using the various commands present in the drawing context API. This meant that for the first time there was an official way to create dynamic graphics with JavaScript.

The `canvas` element was originally created by Apple in 2004 as a proprietary addition to WebKit. It was later adopted by other browser manufacturers, and then by the W3C as a part of HTML5. Today, `canvas` enjoys wide support in modern browsers.

SUPPORT LEVEL
Excellent

All modern browsers have supported `canvas` elements and all of the features covered in this chapter for at least the last three versions.

WHATWG Living Standard: `http://www.whatwg.org/specs/web-apps/current-work/multipage/the-canvas-element.html`

W3C Draft: `http://www.w3.org/html/wg/drafts/2dcontext/html5_canvas_(`

The `Canvas` Drawing Mode

If you're familiar with graphics libraries, you probably have heard the terms *immediate*

mode and *retained mode* as descriptions for how things are rendered on the screen. In immediate-mode rendering, graphics are rendered as the API calls are initiated, and nothing about them is stored by the drawing context.

In retained-mode rendering, calls to the API do not cause immediate rendering on the screen. Rather, the results of the APIs are stored in an internal model maintained by the library, thus allowing the library to do various optimizations when it does draw everything.

The `canvas` tag renders in immediate mode: as soon as you make a call to the API, the results will be rendered on the screen, and the `canvas` will not store any information about whatever was just drawn. If you wish to redraw the same thing, you will have to issue the same command(s) over again.

The `Canvas` Drawing Context

`Canvas` elements are accessible through the DOM just like any other HTML element. However, each `canvas` element exposes one or more drawing contexts that can be used to draw on the `canvas` in various ways. At the moment the only context specified in the standard and supported by browsers is the 2-dimensional (or *2d*) context.

The 2d context exposes an impressive API for drawing lines, curves, shapes, text, and so forth on the `canvas` element. Each `canvas` has a coordinate system with the origin (0, 0) in the upper left corner. The 2d drawing context uses an imaginary pen metaphor for its basic drawing functions, so the commands to draw on the `canvas` are something like, “Move the pen to these coordinates, then draw this thing.” In addition, drawing things and filling them in or stroking them are separate concepts and are carried out by separate commands. When you first draw a path, it is not shown on the screen—you must apply a fill or stroke to make it visible. This is for efficiency, because this way you can draw a complex path consisting of many parts, and then stroke or fill the entire thing at once.

To get started, draw a simple line. The syntax for this is quite straightforward, as demonstrated in Listing 4-1.

Listing 4-1. The Basic Drawing Syntax for `canvas`

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
```

```
canvas {
  border: 1px solid #000;
  width: 200px;
  height: 200px;
}
      </style>
    </head>
    <body>
      <canvas id="myCanvas"></canvas>
      <script>
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');
myContext.moveTo(0, 0);
myContext.lineTo(200, 200);
myContext.strokeStyle = '#000';
myContext.stroke();
      </script>
    </body>
</html>
```

This example has a basic `canvas` element on the page. It uses CSS to give the `canvas` dimensions and a border so you can see it. The script gets a reference to the `canvas` element and then uses that reference to get the drawing context. It then uses the `moveTo` method to move the pen to the upper left corner of the `canvas`, and then instructs the context to draw a line (as a path) to the lower right corner at (200, 200). Last, it sets the stroke style to black and instructs the context to stroke the path.

The results shown in Figure 4-1 are somewhat unexpected.

Figure 4-1. *The results of Listing 4-1*

You expected the line to go from 0, 0 to 200, 200 . . . and actually it did. The default size for a `canvas` element is 200 pixels high by 400 pixels wide. You used CSS to specify the dimensions of the `canvas`, which just made the `canvas` adjust its aspect ratio rather than actually reduce its default width. This brings us to an important detail: in a `canvas`, the coordinate system does not necessarily correspond with screen pixels.

This is a common mistake with canvases, and it happens because we're all trained to use CSS to change the appearance of HTML elements. In the case of the `canvas` element, though, you need to specify its dimensions using its `width` and `height` properties. Listing 4-2 adds those to the markup.

Listing 4-2. Specifying the Width and Height for a Canvas Element

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
canvas {
  border: 1px solid #000;
}
        </style>
    </head>
    <body>
```

```
      <canvas id="myCanvas" width="200" height="200"></canvas>
      <script>
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');
myContext.moveTo(0, 0);
myContext.lineTo(200, 200);
myContext.strokeStyle = '#000';
myContext.stroke();
      </script>
    </body>
</html>
```

As you can see, this removed the width and height declarations from the CSS rule and instead directly applied the dimensions to the `canvas` element using the `width` and `height` properties. Then it drew and stroked the path, and the results were as expected, as shown in Figure 4-2.

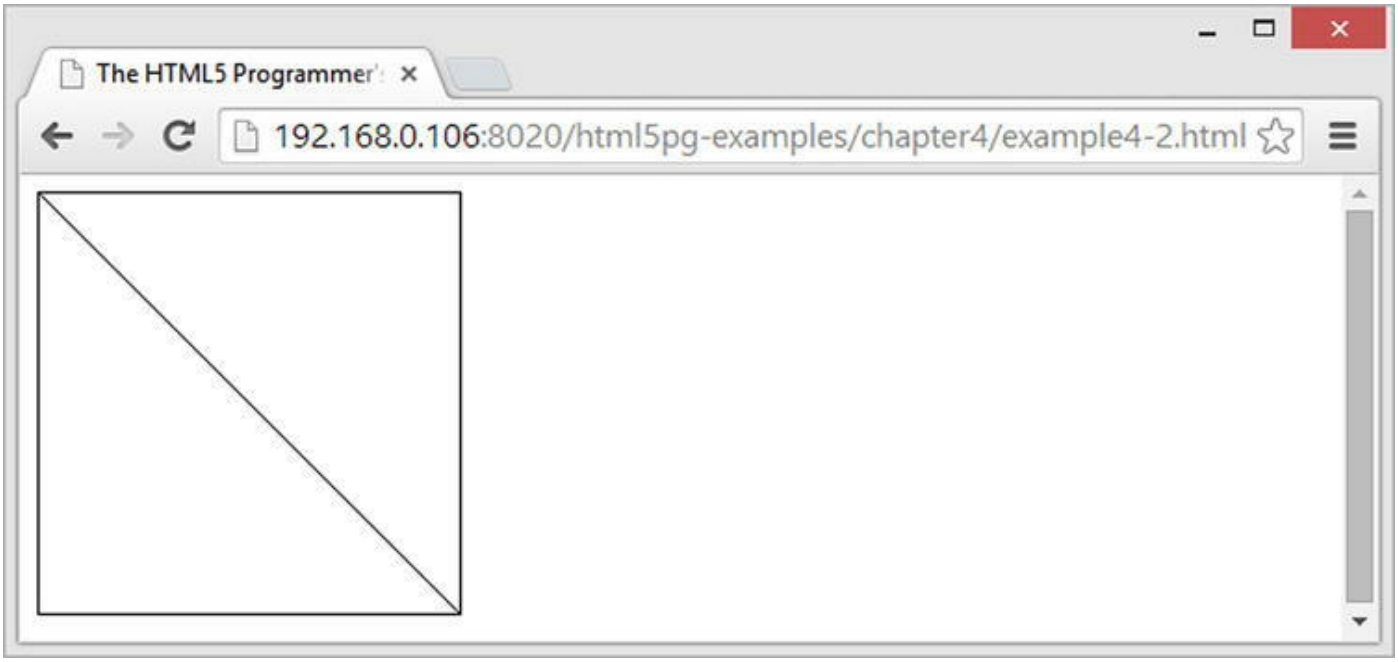


Figure 4-2. *Small victories*

As you can see, the `canvas` is now truly 200 pixels by 200 pixels, and your line draws exactly as you expected.

The `canvas` tag is not self-closing, so the closing tag is mandatory. You can include alternate content inside of the `canvas` tag, which will render if the browser does not support the `canvas` element. You can easily extend this simple example to include some alternate content for older browsers, as shown in Listing 4-3.

Listing 4-3. Alternate Content for Browsers That Don't Support Canvas

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
canvas {
  border: 1px solid #000;
}
        </style>
    </head>
    <body>
      <canvas id="myCanvas" width="200" height="200">Did
You Know: Every time
        you use a browser that doesn't support HTML5,
somewhere a kitten
        cries. Be nice to kittens, upgrade your browser!
      </canvas>
      <script>
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');
myContext.moveTo(0, 0);
myContext.lineTo(200, 200);
myContext.strokeStyle = '#000';
myContext.stroke();
      </script>
    </body>
</html>
```

As you can see, we should all think of the kittens.

Now that you have a basic idea of the canvas tag and the drawing context, you'll dive into the available drawing commands.

Basic Drawing Commands

`Canvas` provides a set of drawing commands that can be used to build complex graphics. Most of the drawing commands are for building paths. In fact, `canvas` only includes commands for one shape primitive: the rectangle. You will have to build any other shapes using a combination of simpler curves.

Given a drawing `Context`, the basic curves are:

- `Context.lineTo(x, y)`: Draws a line from the current pen position to the specified coordinates.
- `Context.arc(x, y, radius, startAngle, endAngle, anticlockwise)`: Draws an arc along a circle centered at `(x, y)` with the specified radius. The `startAngle` and `endAngle` parameters are the start and end angles in radians, and the optional `anticlockwise` parameter is a boolean indicating whether the curve should be drawn anticlockwise (the default is `false`, so arcs by default are drawn clockwise).
- `Context.quadraticCurveTo(cp1x, cp1y, x, y)`: Draws a quadratic curve starting at the current pen location and ending at the coordinates `(x, y)`, with the control point at `(cp1x, cp1y)`.
- `Context.bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y)`: Draw a bezier curve starting at the current pen location and ending at the coordinates `(x, y)`, with control point 1 specified by `(cp1x, cp1y)` and control point 2 specified by `(cp2x, cp2y)`.
- `Context.rect(x, y, width, height)`: Draw a rectangle starting at `(x, y)` with the width and height specified.

Declaring paths is done using two simple commands:

- `Context.beginPath()`: Starts a new path definition. All curves that follow until the path is closed will be included in the path.
- `Context.closePath()`: Ends the path definition, and closes the path by drawing a straight line from the current pen position to the starting point of the path.

By themselves, paths are invisible. You have to tell the `canvas` to either stroke them or fill them:

- `Context.strokeStyle`: This property defines the style that will be stroked on the current path when the stroke method is called. This property can take any valid CSS color string (e.g.,

'red','#000', or 'rgb(30, 50, 100)'), a gradient object, or a pattern object.

- `Context.stroke()`: Strokes the current path with the style specified in `Context.strokeStyle`.
- `Context.fillStyle`: This property defines the style that will be filled into the current path when the `fill` method is called. This property can take a CSS color string, a gradient object, or a pattern object.
- `Context.fill()`: Fills the current path with the style specified in `Context.fillStyle`.
- `Context.lineWidth`: This property defines the thickness of the stroke applied to paths. Defaults to 1 unit.
- `Context.lineCap`: This property defines how lines are capped. Valid values are:
    - `butt`: The line ends are squared off and end precisely at the specified endpoint. This is the default value.
    - `round`: The line ends are rounded and end slightly over the specified endpoint.
    - `square`: The line ends are squared by adding a box to the end of the line whose width is equal to the width of the line and whose height is half of the width of the line.
- `Context.lineJoin`: This property defines how connecting lines are joined together. Valid values are:
    - `bevel`: The joint is beveled.
    - `miter`: The joint is mitered.
    - `round`: The joint is rounded.

Listing 4-4 gives an illustration of the `lineCap` property.

Listing 4-4. Line Caps

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
```

```
      <style>
canvas {
  border: 1px solid #000;
}
      </style>
    </head>
    <body>
      <canvas id="myCanvas" width="200" height="200">Did
You Know: Every time
      you use a browser that doesn't support HTML5,
somewhere a kitten
      cries. Be nice to kittens, upgrade your browser!
     </canvas>
     <script>
// Get the context we will be using for drawing.
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');
myContext.lineWidth = 20;

// Set up an array of valid ending types.
var arrEndings = ['round', 'square', 'butt'];
var i = 0, arrEndingsLength = arrEndings.length;

for (i = 0; i < arrEndingsLength; i++){
  myContext.lineCap = arrEndings[i];
  myContext.beginPath();
  myContext.moveTo(50 + (i * 50), 35);
  myContext.lineTo(50 + (i * 50), 170);
  myContext.stroke();
}
     </script>
    </body>
</html>
```

This example uses the `canvas` to draw thick lines to better illustrate line caps. As always, begin by getting the drawing context of your target `canvas`, and setting the `lineWidth` for the drawing. Then make use of an array of line ending values and loop through the array to draw a line for each one, as shown in Figure 4-3.

Figure 4-3. *Canvas line caps*

You can see that the rounded and squared caps take the line a bit over the actual end of the line. Sometimes this can cause strange effects if your strokes need to be particularly tight. If that's the case, just reduce the length of the path a bit to account for the extra stroke.

Listing 4-5 illustrates the various values of the `lineJoin` property.

Listing 4-5. Line Joins

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
canvas {
  border: 1px solid #000;
}
        </style>
    </head>
    <body>
      <canvas id="myCanvas" width="200" height="200">Did
You Know: Every time
        you use a browser that doesn't support HTML5,
somewhere a kitten
        cries. Be nice to kittens, upgrade your browser!
```

```
      </canvas>
      <script>
// Get the context we will be using for drawing.
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');
myContext.lineWidth = 20;

// Set up an array of valid ending types.
var arrJoins = ['round', 'miter', 'bevel'];
var i = 0, arrJoinsLength = arrJoins.length;

for (i = 0; i < arrJoinsLength; i++){
  myContext.lineJoin = arrJoins[i];
  myContext.beginPath();
  myContext.moveTo(55, 60 + (i * 60));
  myContext.lineTo(95, 20 + (i * 60));
  myContext.lineTo(135, 60 + (i * 60));
  myContext.stroke();
}
      </script>
    </body>
</html>
```

Similar to the previous example, Listing 4-5 uses an array of valid join values to provide the structure for this demonstration. It loops through the array and draws an example of each one, as shown in Figure 4-4.

Figure 4-4. *Canvas line joins*

You can see that the `round` join provides a slightly rounded cap on the obtuse side of the joint, while the `miter` join slightly squares the obtuse side.

Listing 4-6 shows using the stroke properties on arcs.

Listing 4-6. Random Circle Generator

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
canvas {
  border: 1px solid #000;
}
        </style>
    </head>
    <body>
      <canvas id="myCanvas" width="200" height="200">Did
You Know: Every time
        you use a browser that doesn't support HTML5,
somewhere a kitten
        cries. Be nice to kittens, upgrade your browser!
      </canvas>
      <script>
// Get the context we will be using for drawing.
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

// Create a loop that will draw a random circle on the
canvas.
var cycles = 10,
    i = 0;
for (i = 0; i < cycles; i++) {
  var randX = getRandomIntegerBetween(50, 150);
  var randY = getRandomIntegerBetween(50, 150);
  var randRadius = getRandomIntegerBetween(10, 100);
  myContext.beginPath();
  myContext.arc(randX, randY, randRadius, 0, 6.3);
  randStroke();
}
```

```
/**
 * Returns a random integer between the specified minimum
and maximum values.
 * @param {number} min The lower boundary for the random
number.
 * @param {number} max The upper boundary for the random
number.
 * @return {number}
 */
function getRandomIntegerBetween(min, max) {
  return Math.floor(Math.random() * (max - min + 1))
+ min;
}

/**
 * Returns a random color formatted as an rgb string.
 * @return {string}
 */
function getRandRGB() {
  var randRed = getRandomIntegerBetween(0, 255);
  var randGreen = getRandomIntegerBetween(0, 255);
  var randBlue = getRandomIntegerBetween(0, 255);
  return 'rgb(' + randRed + ', ' + randGreen + ', ' +
randBlue + ')';
}

/**
 * Performs a randomized stroke on the current path.
 */
function randStroke() {
  myContext.lineWidth = getRandomIntegerBetween(1, 10);
  myContext.strokeStyle = getRandRGB();
  myContext.stroke();
}
    </script>
  </body>
</html>
```

In this example you are creating ten random circles on the `canvas`, each at a random location, with a random radius, line width, and stroke color. The `getRandomIntegerBetween` function makes it easy to get the numbers you need.

You also have a `randStroke` function for stroking the current path with a random width and color. The results are shown in Figure 4-5.

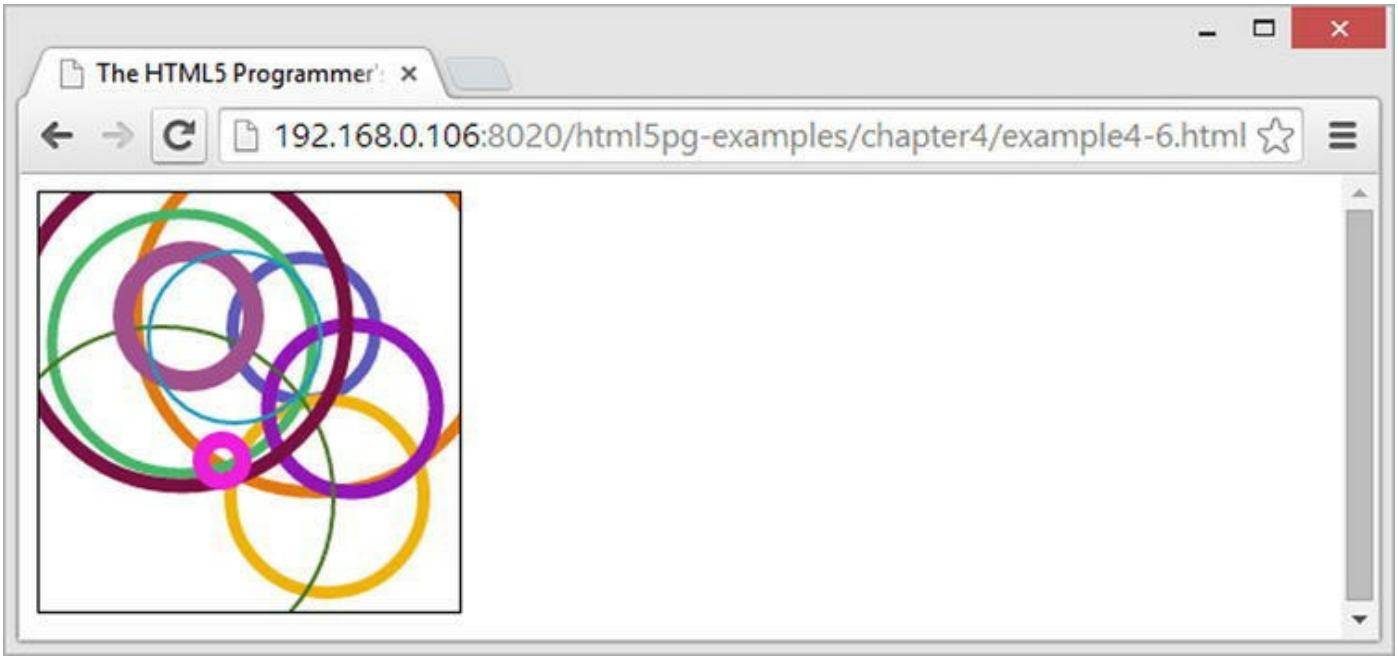


Figure 4-5. *The results of Listing 4-6*

I mentioned before that `canvas` can also draw rectangles. The commands are simple:

- `Context.fillRect(x, y, width, height)`: Draw a rectangle at the specified coordinates and with the specified width and height filled with the current fill style.
- `Context.strokeRect(x, y, width, height)`: Draw a rectangle at the specified coordinates and with the specified width and height stroked with the current stroke style.
- `Context.clearRect(x, y, width, height)`: Clears the specified rectangular area of any other drawing.

Listing 4-7 illustrates drawing rectangles.

Listing 4-7. Drawing Rectangles in a Canvas

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
```

```
      <style>
canvas {
  border: 1px solid #000;
}
      </style>
    </head>
    <body>
      <canvas id="myCanvas" width="200" height="200">Did
You Know: Every time
      you use a browser that doesn't support HTML5,
somewhere a kitten
      cries. Be nice to kittens, upgrade your browser!
     </canvas>
     <script>
// Get the context we will be using for drawing.
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

// Set a stroke style and stroke a rectangle.
myContext.strokeStyle = 'green';
myContext.strokeRect(30, 30, 50, 100);

// Set a fill style and fill a rectangle.
myContext.fillStyle = 'rgba(200, 100, 75, 0.5)';
myContext.fillRect(20, 20, 50, 50);

// Clear a rectangle.
myContext.clearRect(25, 25, 25, 25);
     </script>
    </body>
</html>
```

You're not doing anything fancy with this example, just stroking, filling, and clearing rectangles. The results look as you would expect (Figure 4-6).

Figure 4-6. *Rectangles—yay!*

Gradients and Patterns

You've seen how `canvas` can set different stroke and fill styles, and I mentioned that those styles can be any valid CSS color string (e.g. `green` or `rgba(100, 100, 100, 0.3)`). In addition, `canvas` can define `gradient` and `pattern` objects that can be used to fill and stroke paths.

Gradients

`Canvas` can create both linear and radial gradients:

- `Context.createLinearGradient(x, y, x1, y1)`: Creates a linear gradient starting at coordinates `(x, y)` and ending at coordinates `(x1, y1)`. Returns a `Gradient` object that can be used as a stroke or fill style.
- `Context.createRadialGradient(x, y, r, x1, y1, r1)`: Creates a radial gradient consisting of two circles, the first one centered at `(x, y)` with radius `r`, and the other centered at `(x1, y1)` with radius `r1`. Returns a `Gradient` object that can be used as a stroke or fill style.

- `Gradient.addColorStop(position, color)`: Adds a color stop to the `Gradient`. The position parameter must be between 0 and 1; it defines the relative position within the gradient of the color stop. You can add as many color stops as you want to a particular `Gradient`.

Listing 4-8 shows a simple three-stop gradient being used to stroke rectangles.

Listing 4-8. A Three-Stop Gradient

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
canvas {
  border: 1px solid #000;
}
        </style>
    </head>
    <body>
      <canvas id="myCanvas" width="200" height="200">Did
You Know: Every time
        you use a browser that doesn't support HTML5,
somewhere a kitten
        cries. Be nice to kittens, upgrade your browser!
      </canvas>
      <script>
// Get the context we will be using for drawing.
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

// Create a gradient object and add color stops.
var myGradient = myContext.createLinearGradient(0, 0, 200,
200);
myGradient.addColorStop(0, '#000');
myGradient.addColorStop(0.6, 'green');
myGradient.addColorStop(1, 'blue');

// Set the stroke styles and stroke some rectangles.
myContext.strokeStyle = myGradient;
myContext.lineWidth = 20;
```

```
myContext.strokeRect(10, 10, 110, 110);
myContext.strokeRect(80, 80, 110, 110);
     </script>
   </body>
</html>
```

This example creates a linear `gradient` object and adds three color stops to it, then uses it as the stroke style for two rectangles. The results are shown in Figure 4-7.

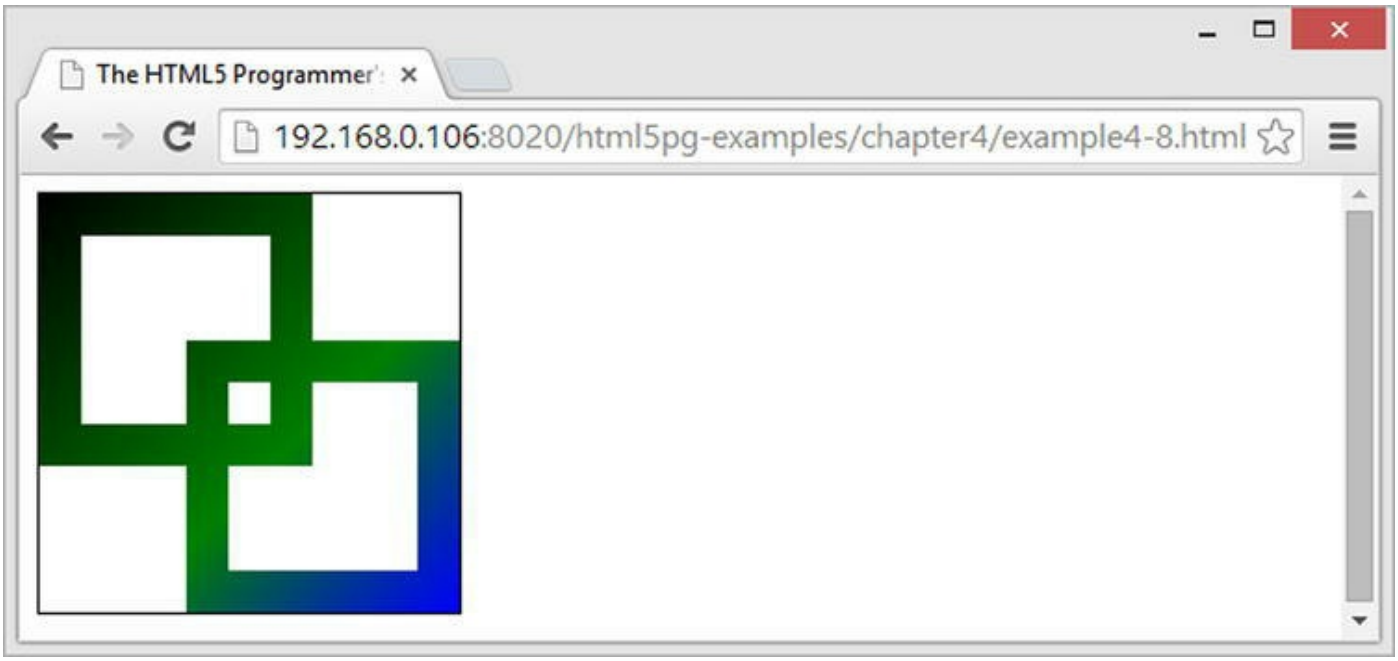


Figure 4-7. *A linear gradient*

Patterns

`Canvas` also supports the concept of a pattern as a fill or stroke style:

- `Context.createPattern(Image, repeat)`: Creates a `Pattern` object that can be used as a fill or stroke style. The `Image` parameter must be any valid `Image` (see "Images" section, next, for details). The `repeat` parameter specifies how the pattern image is repeated.ust be one of the following:
    - `repeat`: Tiles the image both horizontally and vertically.
    - `repeat-x`: Repeats the image only horizontally.
    - `repeat-y`: Repeats the image only vertically.

- `no-repeat`: Does not repeat the image at all.

Listing 4-9 illustrates using a simple image as a pattern.

Listing 4-9. Creating a Pattern

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
canvas {
  border: 1px solid #000;
}
        </style>
    </head>
    <body>
      <canvas id="myCanvas" width="200" height="200">Did
You Know: Every time
        you use a browser that doesn't support HTML5,
somewhere a kitten
        cries. Be nice to kittens, upgrade your browser!
      </canvas>
      <script>
// Get the context we will be using for drawing.
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

// Create a new image element and fill it with a kitten.
var myImage = new Image();
myImage.src = 'http://www.placekitten.com/g/50/50';

// We can't do anything until the image has successfully
loaded.
myImage.onload = function() {
  // Create a pattern with the image and use it as the
fill style.
  var myPattern = myContext.createPattern(myImage,
'repeat');
  myContext.fillStyle = myPattern;
  myContext.fillRect(5, 5, 150, 150);
};
```

```
    </script>
  </body>
</html>
```

This example creates a new `image` element and sets its URL to a placeholder image service. You have to wait for the image to finish loading before continuing, so you attach an `onload` event handler to it, in which you create the `pattern` and use it as the fill style for a rectangle.

The results look as cute as you would expect, as shown in Figure 4-8.

Figure 4-8. *A kitten as a pattern*

Images

The `canvas` element can also load and manipulate images. Once an image is loaded into a `canvas`, you can also draw on it with the drawing commands.

The `canvas` element can use these sources for images:

- an `img` element,
- a video element, and
- another `canvas` element.

`Canvas` has one method for drawing images, but it can take many different parameters and thus has multiple capabilities:

- `Context.drawImage(CanvasImageSource, x, y)`: Draw the image from `CanvasImageSource` at the coordinates `(x, y)`.
- `Context.drawImage(CanvasImageSource, x, y, width, height)`: Draw the image at coordinates `(x, y)`, scaling the image to the specified `width` and `height`.
- `Context.drawImage(CanvasImageSource, sliceX, sliceY, sliceWidth, sliceHeight, x, y, width, height)`: Slice the area from the image specified by the rectangle starting at `(sliceX, sliceY)` with `sliceWidth` and `sliceHeight`, and then draw that slice on the `canvas` at (`x`, `y`), scaling the slice to the specified `width` and `height`.

Listing 4-10 demonstrates the basic functionality of `drawImage`.

Listing 4-10. Drawing an Image on a Canvas

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
canvas {
  border: 1px solid #000;
}
        </style>
    </head>
    <body>
      <canvas id="myCanvas" width="200" height="200">Did
You Know: Every time
        you use a browser that doesn't support HTML5,
somewhere a kitten
        cries. Be nice to kittens, upgrade your browser!
      </canvas>
      <script>
// Get the context we will be using for drawing.
```

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

// Create a new image element and fill it with a kitten.
var myImage = new Image();
myImage.src = 'http://www.placekitten.com/g/150/150';

// We can't do anything until the image has successfully
loaded.
myImage.onload = function() {
  myContext.drawImage(myImage, 25, 25);
};
      </script>
    </body>
</html>
```

In this example all you're doing is creating a new `img` element of a placeholder image. Once the image is loaded, draw it on your `canvas`, as shown in Figure 4-9.

Figure 4-9. *An image drawn in a canvas*

Listing 4-11 demonstrates scaling an image on a canvas.

Listing 4-11. Image Scaling with Canvas

```
<!DOCTYPE html>
<html>
```

```
    <head>
       <title>The HTML5 Programmer's Reference</title>
       <style>
canvas {
  border: 1px solid #000;
}
       </style>
    </head>
    <body>
      <canvas id="myCanvas" width="200" height="200">Did
You Know: Every time
       you use a browser that doesn't support HTML5,
somewhere a kitten
       cries. Be nice to kittens, upgrade your browser!
      </canvas>
      <script>
// Get the context we will be using for drawing.
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

// Create a new image element and fill it with a kitten.
var myImage = new Image();
myImage.src = 'http://www.placekitten.com/g/50/50';

// We can't do anything until the image has successfully
loaded.
myImage.onload = function() {
  myContext.drawImage(myImage, 25, 25, 50, 150);
};
      </script>
    </body>
</html>
```

This example gives you a 100px by 100px placeholder, but when you draw it on the `canvas`, you scale it to be 50px × 150px, as shown in Figure 4-10.

Figure 4-10. *Scaling an image in canvas*

Finally, Listing 4-12 shows slicing a larger image and scaling the slice on the `canvas`.

Listing 4-12. Slicing and Scaling an Image on `canvas`

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
canvas {
  border: 1px solid #000;
}
        </style>
    </head>
    <body>
      <canvas id="myCanvas" width="200" height="200">Did
You Know: Every time
        you use a browser that doesn't support HTML5,
somewhere a kitten
        cries. Be nice to kittens, upgrade your browser!
      </canvas>
      <script>
// Get the context we will be using for drawing.
```

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

// Create a new image element and fill it with a kitten.
var myImage = new Image();
myImage.src = 'http://www.placekitten.com/g/300/300';

// We can't do anything until the image has successfully
loaded.
myImage.onload = function() {
  myContext.drawImage(myImage, 25, 25, 150, 150, 0, 0,
150, 50);
};
      </script>
    </body>
</html>
```

Here you are loading a 300px × 300px placeholder image, but slicing only a 75px × 75px portion of it starting at `(25, 25)`. Then you're taking that slice and rendering it in the `canvas`, scaling it to be 150px × 50px. The result is rather distorted, as Figure 4-11 shows.

Figure 4-11. *Poor kitty*

Saving `Canvas` Contents

Once you have a drawing on a `canvas`, you might want to save it somehow. This would involve grabbing the image data and transmitting it to a server from which it can be reconstituted and displayed. The `canvas` API does provide a method for saving a rendered bitmap:

- `Context.toDataUrl(opt_type, opt_quality)`: Translates the rendered bitmap to a data URI. Data URIs are a way of embedding data directly into web pages and are defined in RFC 2397, which you can read at `http://tools.ietf.org/html/rfc2397`. Valid types include `image/png` (the default), `image/jpeg`, and (for Chrome and Chromium-based browsers, `image/webp`). If the type is `image/jpeg` or `image/webp`, an optional second parameter of between 0 and 1 can be provided to indicate the quality. This method returns the rendered bitmap encoded as a data URI, which you can then transmit back to the server, or even use elsewhere in the same page.

Note that if you have loaded an image into the `canvas` that is from a different origin than the hosting page, or if you have loaded an image from your hard drive into the `canvas`, this method will throw a security error. This is done to prevent information leakage via careless or malicious scripts.

Text

In addition to drawing and images, the `canvas` element can render text. The methods and properties for text rendering are as follows:

- `Context.fillText(textString, x, y, opt_maxWidth)`: Fills the `textString` on the `canvas` starting at `(x, y)` with the current fill style. If the optional `maxWidth` parameter is specified, and the rendered text would exceed that width, the browser will attempt to render the text in such a way as to fit it within the specified width (e.g., use a condensed font face if available, use a smaller font size, etc.).
- `Context.measureText(textString)`: Measures the width that would result if the specified `textString` were to be rendered using the current style. Returns a `TextMetrics` object,

which has a width property that contains the value.

- `Context.strokeText(textString, x, y, opt_maxWidth)`: Strokes the `textString` on the `canvas` starting at `(x, y)` with the current stroke style. If the optional `maxWidth` parameter is specified, and the rendered text would exceed that width, the browser will attempt to render the text in such a way as to fit it within the specified width (e.g., use a condensed font face if available, use a smaller font size, etc.).
- `Context.font`: Sets the font that the text will be rendered in. Any valid CSS font string is permitted.
- `Context.textAlign`: Aligns the text as specified. Valid values are:
    - `left`: Left-aligns the text.
    - `right`: Right-aligns the text.
    - `center`: Centers the text.
    - `start`: Aligns the text at the starting side for the current locale (i.e., left for left-to-right languages and right for right-to-left languages). This is the default value.
    - `end`: Aligns the text at the ending side for the current locale.
- `Context.textBaseline`: Sets the baseline for the text as specified. Valid values are:
    - `alphabetic`: Uses the normal alphabetic baseline for the text. This is the default value.
    - `bottom`: The baseline is the bottom of the em square.
    - `hanging`: Uses the hanging baseline for the text.
    - `ideographic`: Uses the bottom of the body of characters (assuming they protrude beneath the alphabetic baseline).
    - `middle`: The text baseline is the middle of the em square.
    - `top`: The text baseline is the top of the em square.

Listing 4-13 demonstrates how easy it is to draw text on a `canvas`.

Listing 4-13. Rendering Text on Canvas

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
canvas {
  border: 1px solid #000;
}
        </style>
    </head>
    <body>
      <canvas id="myCanvas" width="200" height="200">Did
You Know: Every time
        you use a browser that doesn't support HTML5,
somewhere a kitten
        cries. Be nice to kittens, upgrade your browser!
      </canvas>
      <script>
// Get the context we will be using for drawing.
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

// Draw some text!
myContext.font = '35px sans-serif';
myContext.strokeStyle = '#000';
myContext.strokeText('Hello World', 0, 40);
myContext.textAlign = 'center';
myContext.fillStyle = 'rgba(200, 50, 25, 0.8)';
myContext.fillText('HTML5', 100, 100);
      </script>
    </body>
</html>
```

This example both strokes and fills some text on the `canvas`. The font size is large enough to reveal the actual stroke around the edges of the letters, as shown in Figure 4-12.

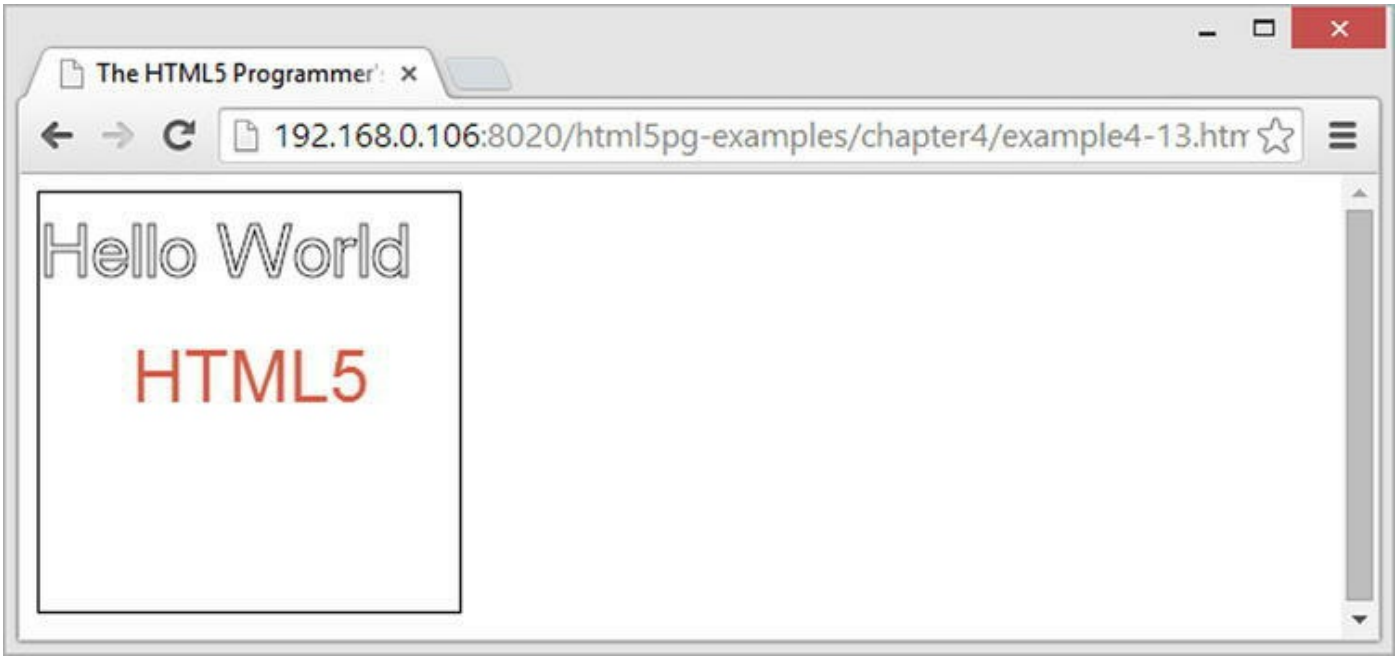


Figure 4-12. *Text rendered on a canvas*

Shadows

The `canvas` element can also cast shadows based on the elements drawn upon it. This is most often used with text, but it also works with shapes and paths. If you're already familiar with CSS drop shadows, then the parameters for `canvas` shadows will be very familiar:

- `Context.shadowBlur`: The size of the blurring effect. The default value is 0.
- `Context.shadowColor`: The color of the shadow. Can be any valid CSS color string. The default is `'rgba(0, 0, 0, 0)'`.
- `Context.shadowOffsetX`: The x-offset of the shadow. The default value is 0.
- `Context.shadowOffsetY`: The y-offset of the shadow. The default value is 0.

Listing 4-14 demonstrates casting drop shadows on some text.

Listing 4-14. Drop Shadows

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
canvas {
  border: 1px solid #000;
}
        </style>
    </head>
    <body>
      <canvas id="myCanvas" width="200" height="200">Did
You Know: Every time
        you use a browser that doesn't support HTML5,
somewhere a kitten
        cries. Be nice to kittens, upgrade your browser!
      </canvas>
      <script>
// Get the context we will be using for drawing.
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

// Add some shadow!
myContext.shadowOffsetX = 2;
myContext.shadowOffsetY = 2;
myContext.shadowBlur = 2;
myContext.shadowColor = "rgba(0, 0, 0, 0.8)";

// Draw some text!
myContext.font = '35px sans-serif';
myContext.strokeStyle = '#000';
myContext.strokeText('Hello World', 0, 40);
myContext.textAlign = 'center';
myContext.fillStyle = 'rgba(200, 50, 25, 0.8)';
myContext.shadowOffsetX = 4;
myContext.shadowOffsetY = 4;
myContext.fillText('HTML5', 100, 100);
      </script>
    </body>
</html>
```

This example simply adds drop shadows to the code in Listing 4-13. It adds two

different shadow offsets, one quite close and then one farther away, as shown in Figure 4-13.

Figure 4-13. *Shadows rendered on canvas*

Saving Drawing State

The `canvas` API provides a way to store some information about the current state of the drawing context. The information is stored in a stack, and you can push and pull states from the stack as needed. The drawing context properties that can be stored are as follows:

- The current value for `globalAlpha`
- The current `strokeStyle` and `fillStyle`
- The current line settings in `lineCap`, `lineJoin`, `lineWidth`, and `miterLimit`
- The current shadow settings in `shadowBlur`, `shadowColor`, `shadowOffsetX`, and `shadowOffsetY`
- The current compositing operation set in `globalCompositeOperation`
- The current clipping path

- Any transformations that have been applied to the drawing context

Together these values all make up the drawing state. The methods for saving and restoring state are simple:

- `Context.save()`: Takes a snapshot of the current drawing state and save the values in the stack.
- `Context.restore()`: Removes the most recently stored drawing state from the stack and restores it to the context.

The drawing state is saved in a first-in, first-out stack. The save and restore methods are the only two methods for accessing the stack and the states stored within.

Listing 4-15 provides a somewhat contrived demonstration of saving and restoring the drawing state.

Listing 4-15. Saving and Restoring Drawing States

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
canvas {
  border: 1px solid #000;
}
        </style>
    </head>
    <body>
      <canvas id="myCanvas" width="200" height="210">Did
You Know: Every time
        you use a browser that doesn't support HTML5,
somewhere a kitten
        cries. Be nice to kittens, upgrade your browser!
      </canvas>
      <script>
// Get the context we will be using for drawing.
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

// Create an array of colors to load into the stack.
var allTheColors = ['#ff0000', '#ff8800', '#ffff00',
```

```
'#00ff00', '#0000ff',
   '#4b0082', '#8f00ff'];

// Load the colors and stroke style into the stack.
for (var i = 0; i < allTheColors.length; i++) {
  myContext.strokeStyle = allTheColors[i];
  myContext.lineWidth = 30;
  myContext.save();
}

// Restore colors from the stack and draw.
for (var i = 0; i < 8; i++) {
  myContext.restore();
  myContext.beginPath();
  myContext.moveTo(0, ((30 * i) + 15));
  myContext.lineTo(200, ((30 * i) + 15));
  myContext.stroke();
}
     </script>
   </body>
</html>
```

This example programmatically creates a set of drawing states with different colors and a specific line width. Then it restores each state one at a time and draws a line.

You'll notice the y-coordinate for each line is based on the loop index. Each line is stroked 30 units wide: 15 units above the line and 15 units below the line. If you just drew the first line from `(0, 0)` to `(200, 0)` and then stroked it, you would not see the top 15 units of the stroke. Shifting each line down by 15 units assures that you will see the full stroke width of the first line and each subsequent line.

Compositing

In all of your `canvas` examples so far, when you have drawn multiple items on the `canvas` they have just layered one on top of the other. The `canvas` API provides the ability to composite items as they are drawn, which gives you the ability to do some fairly sophisticated manipulations.

Whenever you draw a new element on the `canvas`, the compositor looks at what is already present on the `canvas`. This current content is referred to as the *destination*. The new content is referred to as the *source*. Then the compositor draws the source in reference to the destination according to the currently active compositor.

Compositors are specified using the `globalCompositeOperation` property of the current context. The available compositors are as follows:

- `source-over`: Draws source content over destination content. This is the default compositor.
- `source-atop`: Source content is only drawn where it overlaps the destination content.
- `source-in`: Source content is only drawn where both source and destination content overlap. Everything else is made transparent.
- `source-out`: Source content is only drawn where it does not overlap destination content. Everything else is made transparent.
- `destination-over`: Source content is drawn underneath destination content.
- `destination-atop`: Source content is only kept where it overlaps the destination content. The destination content is drawn underneath the source. Everything else is made transparent.
- `destination-in`: Source content is only kept where it overlaps with the destination content. Everything else is made transparent.
- `destination-out`: Source content is only kept where it does not overlap with the destination content. Everything else is made transparent.
- `copy`: Only draws the destination content. Everything else is made transparent.
- `lighter`: Where destination content and source content overlap, the color is determined by adding the values of the two contents.
- `xor`: The destination content is rendered normally except where it overlaps with source content, in which case both are rendered transparent.

To specify a compositor, simply set `Context.globalCompositeOperation` to the desired value:

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');
myContext.globalCompositeOperation = 'lighter';
```

Listing 4-16 provides a way to view the different compositors in action.

Listing 4-16. Canvas Compositor Demonstration

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
canvas {
  border: 1px solid #000;
}
        </style>
    </head>
    <body>
      <canvas id="myCanvas" width="200" height="200">Did
You Know: Every time
        you use a browser that doesn't support HTML5,
somewhere a kitten
        cries. Be nice to kittens, upgrade your browser!
      </canvas>
      <br>
      <select id="compositor">
        <option value="source-over" selected>source-
over</option>
        <option value="destination-atop">destination-
atop</option>
        <option value="destination-in">destination-
in</option>
        <option value="destination-out">destination-
out</option>
        <option value="destination-over">destination-
over</option>
        <option value="source-atop">source-atop</option>
        <option value="source-in">source-in</option>
        <option value="source-out">source-out</option>
        <option value="copy">copy</option>
        <option value="lighter">lighter</option>
        <option value="xor">xor</option>
      </select>
      <button id="toggle-triangle">Toggle Triangle</button>
      <script>
// Get the context we will be using for drawing.
var myCanvas = document.getElementById('myCanvas');
```

```
var myContext = myCanvas.getContext('2d');

// Get references to the form elements.
var mySelector = document.getElementById('compositor');
var toggleTriangle = document.getElementById('toggle-
triangle');

/**
 * Draws the example shapes with the specified compositor.
 */
function drawExample() {
  // First set the compositing to source-over so we can
guarantee drawing the
  // first shape.
  myContext.globalCompositeOperation = 'source-over';
  myContext.clearRect(0, 0, 200, 200);
  myContext.beginPath();

  // Draw the circle first.
  myContext.arc(60, 100, 40, 0, 7);
  myContext.fillStyle = '#ff0000';
  myContext.fill();

  // Change the compositing to the chosen value.
  myContext.globalCompositeOperation = mySelector.value;

  // Draw a rectangle on top of the circle.
  myContext.beginPath();
  myContext.fillStyle = '#0000ff';
  myContext.rect(60, 60, 80, 80);
  myContext.fill();
}

/**
 * Whether or not to show the triangle.
 * @type {boolean}
 */
var showTriangle = false;

/**
 * Shows or hides the triangle.
 */
function showHideTriangle() {
```

```
  if (showTriangle) {
    myContext.fillStyle = '#00ff00';
    myContext.beginPath();
    myContext.moveTo(40, 80);
    myContext.lineTo(170, 100);
    myContext.lineTo(40, 120);
    myContext.lineTo(40, 80);
    myContext.fill();
  } else {
    drawExample();
  }
}

// Draw the example for the first time.
drawExample();

// Add a change event handler to the selector to redraw
the example with the
// chosen compositor.
mySelector.addEventListener('change', function() {
  showTriangle = false;
  drawExample();
}, false);

// Add a click event handler to the toggle button to show
or hide the triangle.
toggleTriangle.addEventListener('click', function() {
  showTriangle = showTriangle ? false : true;
  showHideTriangle();
}, false);
    </script>
  </body>
</html>
```

This example has created a simple select field with all of the available compositors to choose from. When you choose a compositor, shapes will redraw. The first shape (the red circle) will always draw with `source-over`. The second shape (the blue square) will draw with the newly chosen compositor. You can toggle the green triangle on and off to see how it will composite with the result of the first composition.

The compositors apply to anything that can be drawn on the `canvas`, even images, as demonstrated in Listing 4-17.

Listing 4-17. Compositing a Photograph

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
canvas {
  border: 1px solid #000;
}
        </style>
    </head>
    <body>
      <canvas id="myCanvas" width="200" height="200">Did
You Know: Every time
        you use a browser that doesn't support HTML5,
somewhere a kitten
        cries. Be nice to kittens, upgrade your browser!
      </canvas>
      <script>
// Get the context we will be using for drawing.
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

// Create a new image element and fill it with a kitten.
var myImage = new Image();
myImage.src = 'http://www.placekitten.com/g/150/150';

// We can't do anything until the image has successfully
loaded.
myImage.onload = function() {
  // Create a simple gray linear gradient and set it to
the fill style.
  var myGradient = myContext.createLinearGradient(25, 25,
25, 175);
  myGradient.addColorStop(0.1, '#000');
  myGradient.addColorStop(1, 'rgba(200, 200, 200, 1)');
  myContext.fillStyle = myGradient;

  // Draw a square that almost fills the region where the
image will be rendered
  // and fill it with the gradient.
```

```
  myContext.beginPath();
  myContext.rect(30, 30, 140, 140);
  myContext.fill();

  // Set the compositor to lighter.
  myContext.globalCompositeOperation = 'lighter';

  // Draw the kitten.
  myContext.drawImage(myImage, 25, 25);
};
      </script>
    </body>
</html>
```

This example creates a simple linear gradient and uses it as the fill style for a square, then composites the image of a kitten on top of it using the lighter compositor. An example of the results is shown in Figure 4-14.

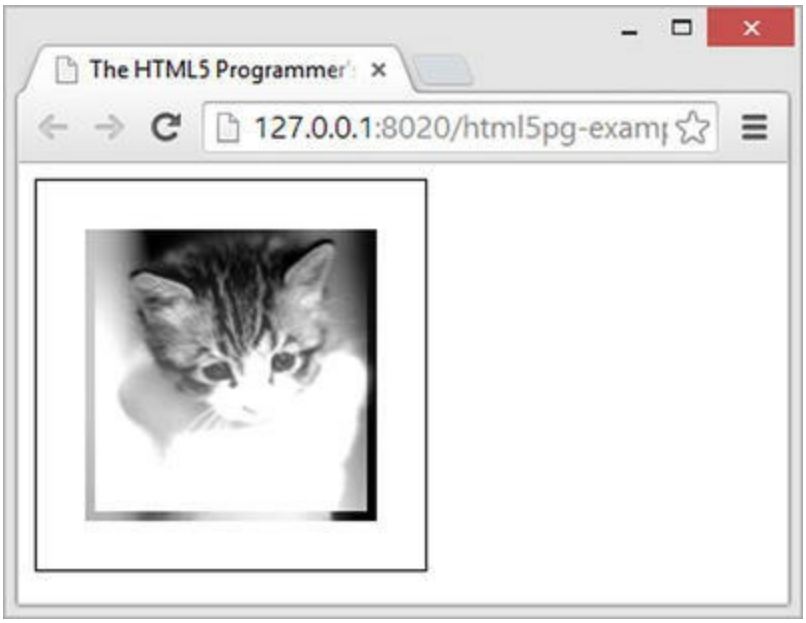


Figure 4-14. *The results of compositing a gradient with an image*

Using compositors with gradients, patterns, and images, you can create some very complex effects with your `canvas` drawings.

Clipping

You can limit the drawing area of the `canvas` to any closed path that you have defined.

This is referred to as *clipping*. You create a clipping area by first drawing a path on the `canvas`, and then calling the `Context.clip()` method, which will limit drawing to that area. You can still stroke and fill the path, or you can create new paths or other drawings. Visibility will be limited to the clipping area.

There are three ways to reset the clipping area:

- You can define a path that encompasses the entire `canvas`, and then clip to that.
- You can restore to a previous drawing state with a different clipping area.
- You can reset the entire `canvas` by resizing it.

Listing 4-18 demonstrates creating a clipping area to limit drawing.

Listing 4-18. Creating a Clipping Area

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
canvas {
  border: 1px solid #000;
}
        </style>
    </head>
    <body>
      <canvas id="myCanvas" width="200" height="200">Did
You Know: Every time
        you use a browser that doesn't support HTML5,
somewhere a kitten
        cries. Be nice to kittens, upgrade your browser!
      </canvas>
      <script>
// Get the context we will be using for drawing.
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

// Create a square clipping area.
myContext.beginPath();
```

```
myContext.rect(50, 50, 50, 50);
myContext.clip();

// Draw a large circle in the canvas and fill it. Only the
portion within
// the clipping area will be visible.
myContext.beginPath();
myContext.arc(75, 75, 100, 0, 7);
myContext.fillStyle = 'red';
myContext.fill();
     </script>
   </body>
</html>
```

This simple example first creates a square path using the `rect` method, and then sets it as the clipping area. Then it draws a large circle and fills it with red, but the only area that is visible is within the clipping area, as shown in Figure 4-15.

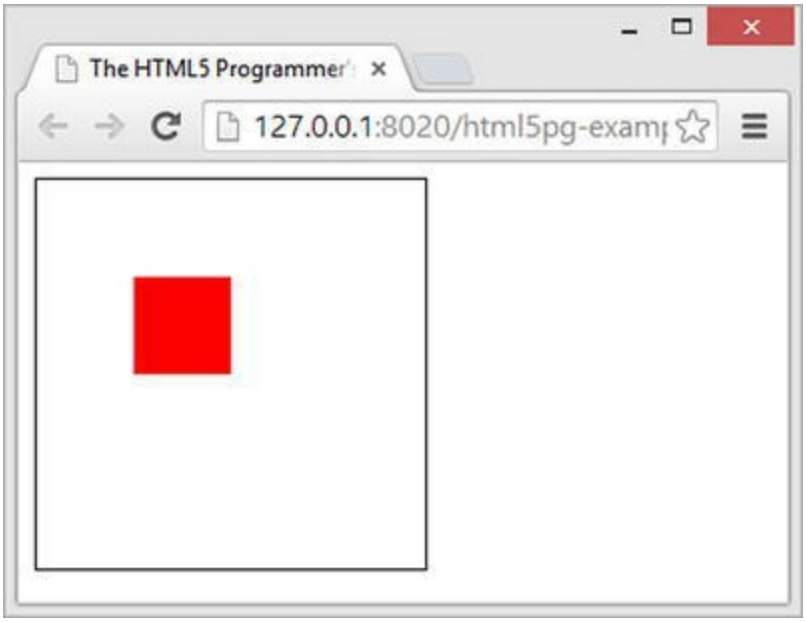


Figure 4-15. *The effects of clipping*

Transformations

The `canvas` API includes a set of methods for changing the way drawings are rendered upon the `canvas`: rotating them, scaling them, or even arbitrary changes like reflection or shearing. These changes are referred to as *transformations*. When a transformation is set, further drawings will be modified in the specified way. The

canvas API has a set of shorthand methods for a few common transformations:

- `Context.translate(translateX, translateY)`: Moves the origin of the `canvas` from its current position to the new x position `translateX` units from the current origin and the new y position `translateY` units from the current origin.
- `Context.rotate(angle)`: Rotates the `canvas` around the origin by the specified angle in radians.
- `Context.scale (scaleX, scaleY)`: Scales the `canvas` units by `scaleX` horizontally and `scaleY` vertically.

In addition, you can specify an arbitrary transformation matrix using the transform method:

- `Context.transform(scaleX, skewX, skewY, scaleY, translateX, translateY)`: Transform the `canvas` by applying a transformation matrix specified as:

$$\begin{bmatrix} scaleX & skewY & translateX \\ scaleX & skewY & translateX \\ 0 & 0 & 1 \end{bmatrix}.$$

The `rotate`, `translate`, and `scale` shorthand methods all map to transformation matrices and thus calls to the transform method. For example, `Context.translate(translateX, translateY)` maps to `Context.transform(1, 0, 0, 1, translateX, translateY)` and `Context.scale(scaleX, scaleY)` maps to `Context.transform(scaleX, 0, 0, scaleY, 0, 0)`.

Note If you're a linear algebra buff, all `canvas` transforms are Affine transforms.

The important thing to remember about `canvas` transformations is that they affect the entire `canvas`—once a transformation has been implemented, it affects everything that is drawn from that point on. `Canvas` transformations also "stack" in that when you apply two different transforms, the second will base its results on the first. This can lead to some unexpected results if you don't carefully manage the active transformations and reset them as needed. You can reset the transformation in one of three ways:

- Specify a special transform called the "unit transform matrix,"

which has no effect on drawing. You can specify this matrix using the transform method: `Context.transform(1, 0, 0, 1, 0, 0)`.

- Restore a previously saved drawing state, which will set the transform to the one for that state.
- Reset the entire `canvas` by resizing it.

Take a look at some simple examples before exploring some of the more complex transformations. Listing 4-19 demonstrates a simple `translate` transformation:

Listing 4-19. A Simple Translate Transformation

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
canvas {
  border: 1px solid #000;
}
        </style>
    </head>
    <body>
      <canvas id="myCanvas" width="200" height="200">Did
You Know: Every time
        you use a browser that doesn't support HTML5,
somewhere a kitten
        cries. Be nice to kittens, upgrade your browser!
      </canvas>
      <script>
// Get the context we will be using for drawing.
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

/**
 * Draws a 100x100 square at (0, 0) in the specified
color. Indicates the origin
 * corner with a small black square.
 * @param {string} color A valid CSS color string.
 */
```

```
function drawSquare(color) {
  myContext.fillStyle = color;
  myContext.beginPath();
  myContext.rect(0, 0, 100, 100);
  myContext.fill();
  myContext.fillStyle = '#000';
  myContext.beginPath();
  myContext.rect(0, 0, 5, 5);
  myContext.fill();
}

// Draw a square, fill it with red.
drawSquare('rgba(255, 0, 0, 0.5)');

// Translate the canvas.
myContext.translate(20, 40);

// Draw the same square again, fill it with blue.
drawSquare('rgba(0, 0, 255, 0.5)');

// Translate the canvas again.
myContext.translate(50, -20);

// Draw the same square again, fill it with green.
drawSquare('rgba(0, 255, 0, 0.5)');

    </script>
  </body>
</html>
```

This example (which will form the basis of the next few examples) creates a simple method for drawing a square at the origin of the `canvas`. The function fills the square with the specified color (or you could pass in any valid `fillStyle`). To help keep track of the origin, the function also creates a small back notch in the corner of the square at the origin.

First it draws a square at the origin and colors it red. Then it translates the `canvas`, and draw a blue square. Finally, it translates the `canvas` again and draws a green square. The results are shown in Figure 4-16.

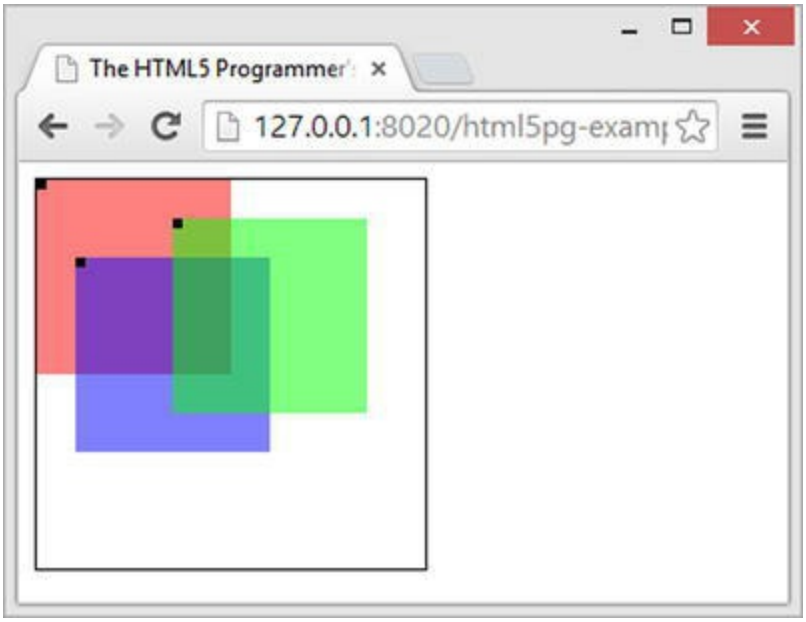


Figure 4-16. *The results of Listing 4-18*

As you can see, the translation causes the origin of the `canvas` to move as specified.

Next, Listing 4-20 builds on this example by applying a rotation as well as a transformation:

Listing 4-20. Stacking a Rotation on a Translation

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
canvas {
  border: 1px solid #000;
}
        </style>
    </head>
    <body>
      <canvas id="myCanvas" width="200" height="200">Did
You Know: Every time
        you use a browser that doesn't support HTML5,
somewhere a kitten
        cries. Be nice to kittens, upgrade your browser!
      </canvas>
      <script>
```

```
// Get the context we will be using for drawing.
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

/**
 * Draws a 100x100 square at (0, 0) in the specified
color. Indicates the origin
 * corner with a small black square.
 * @param {string} color A valid CSS color string.
 */
function drawSquare(color) {
  myContext.fillStyle = color;
  myContext.beginPath();
  myContext.rect(0, 0, 100, 100);
  myContext.fill();
  myContext.fillStyle = '#000';
  myContext.beginPath();
  myContext.rect(0, 0, 5, 5);
  myContext.fill();
}

// Draw a square, fill it with red.
drawSquare('rgba(255, 0, 0, 0.5)');

// Translate the canvas.
myContext.translate(20, 40);

// Rotate the canvas 45 degrees (about 0.785 radians).
myContext.rotate(0.785);

// Draw the same square again, fill it with blue.
drawSquare('rgba(0, 0, 255, 0.5)');

// Translate the canvas again.
myContext.translate(50, -20);

// Rotate the canvas 45 degrees (about 0.785 radians).
myContext.rotate(0.785);

// Draw the same square again, fill it with green.
drawSquare('rgba(0, 255, 0, 0.5)');
     </script>
   </body>
```

```
</html>
```

It uses the same translations as before, but adds a rotation as well before drawing the new squares. The results are shown in Figure 4-17.

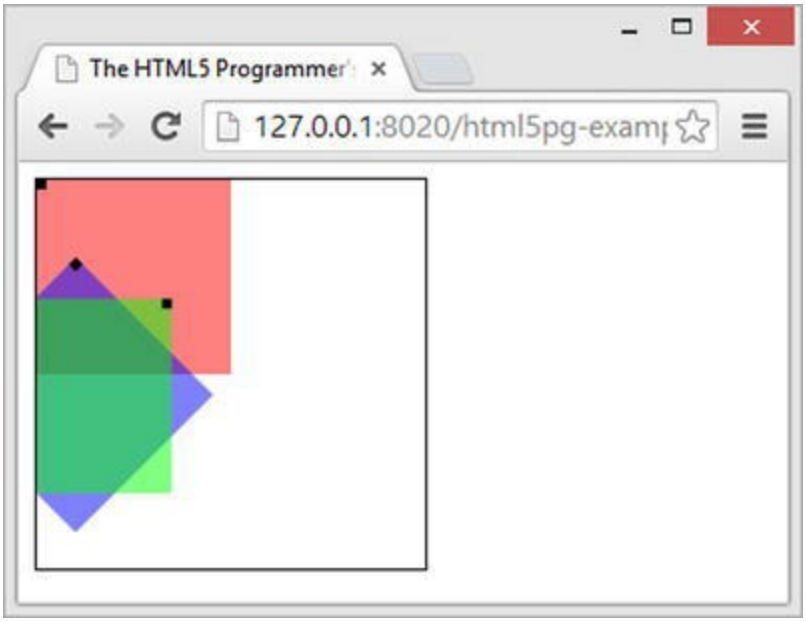


Figure 4-17. *Rotations and translations*

Here you can see the same translations, along with the rotations. You can see each square is rotated around its origin corner.

Finally, you can look at some scale transformations in Listing 4-21.

Listing 4-21. Scale and Translate Transformations

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
canvas {
  border: 1px solid #000;
}
        </style>
    </head>
    <body>
      <canvas id="myCanvas" width="200" height="200">Did
You Know: Every time
        you use a browser that doesn't support HTML5,
```

```
somewhere a kitten
       cries. Be nice to kittens, upgrade your browser!
     </canvas>
     <script>
// Get the context we will be using for drawing.
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

/**
 * Draws a 100x100 square at (0, 0) in the specified
color. Indicates the origin
 * corner with a small black square.
 * @param {string} color A valid CSS color string.
 */
function drawSquare(color) {
  myContext.fillStyle = color;
  myContext.beginPath();
  myContext.rect(0, 0, 100, 100);
  myContext.fill();
  myContext.fillStyle = '#000';
  myContext.beginPath();
  myContext.rect(0, 0, 5, 5);
  myContext.fill();
}

// Draw a square, fill it with red.
drawSquare('rgba(255, 0, 0, 0.5)');

// Translate the canvas.
myContext.translate(20, 40);

// Scale the canvas.
myContext.scale(1, 1.5);

// Draw the same square again, fill it with blue.
drawSquare('rgba(0, 0, 255, 0.5)');

// Translate the canvas again.
myContext.translate(50, -20);

// Scale the canvas again.
myContext.scale(1.5, 1);
```

```
// Draw the same square again, fill it with green.
drawSquare('rgba(0, 255, 0, 0.5)');
    </script>
  </body>
</html>
```

Again this example builds on Listing 4-19 and uses the same function and translations. This time it adds in a scale translation before drawing the second and third squares, as shown in Figure 4-18.

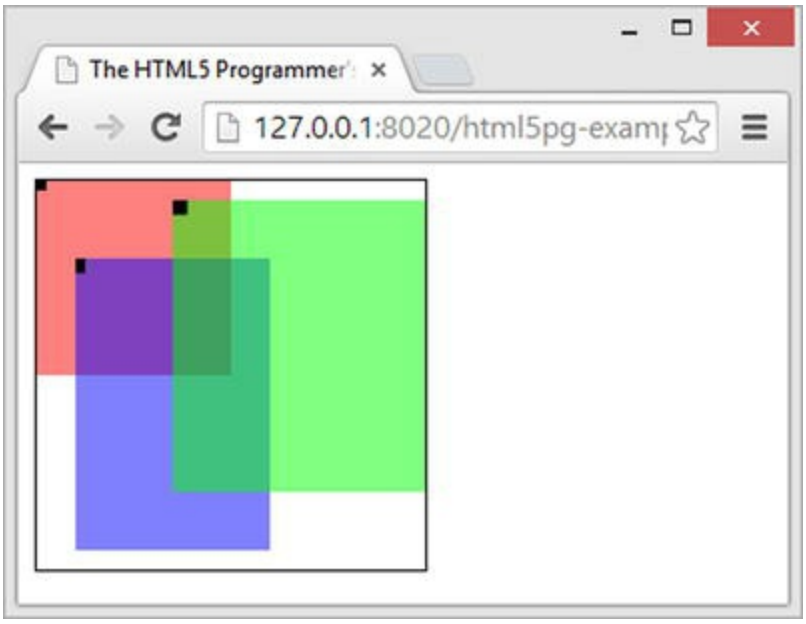


Figure 4-18. *Scaling and translating*

If you look closely, you can see that the origin marker for the blue square is slightly elongated as per the scale transformation you applied to it. And if you compare the origin marker for the green square with that of the red square, you'll see that the former is twice the size of the latter.

For a more practical example, consider creating dynamic reflections of elements. It's quite easy with transforms, as demonstrated in Listing 4-22.

Listing 4-22. A Simple Text Reflection

```
<!DOCTYPE html>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
    <style>
canvas {
```

```
  border: 1px solid #000;
}
      </style>
    </head>
    <body>
      <canvas id="myCanvas" width="200" height="200">Did
You Know: Every time
       you use a browser that doesn't support HTML5,
somewhere a kitten
       cries. Be nice to kittens, upgrade your browser!
      </canvas>
      <script>
// Get the context we will be using for drawing.
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

// Draw some text!
myContext.font = '35px sans-serif';
myContext.fillStyle = '#000';
myContext.fillText('Hello World', 10, 100);

// Set a reflection transform.
myContext.setTransform(1, 0, 0, -1, 0, 0);

// Set a slight scale transform.
myContext.scale(1, 1.2);

// Draw the text again with the transforms in place and
a light gray fill style.
myContext.fillStyle = 'rgba(100, 100, 100, 0.4)';
myContext.fillText('Hello World', 10, -85);
      </script>
    </body>
</html>
```

This example draws some text, then applies a reflection transform and a scale transform to the `canvas`, and then redraws the same text in a light gray. The result is shown in Figure 4-19.

Figure 4-19. *Text reflection*

You could even use a gradient as the fill style for the reflected text, resulting in a shadow that fades from top to bottom

Animation

The `canvas` API doesn't offer any native support for animation. It has no methods for incrementally animating its contents, and as you have seen it provides no way to reference the contents once they have been rendered. However, the drawing tools that `canvas` does provide are so low-level and efficient that you can create animations with `canvas` by literally drawing each animation frame separately.

As you will see in "Animation Timing" in Chapter 5, most JavaScript-based animation is done in timed loops, and animating with `canvas` is no different. In fact, to simplify the animation examples, you will use the DrawCycle constructor you built in Listing 5-5. That will allow you to create a draw cycle manager that uses `requestAnimationFrame` for maximizing the efficiency of your animations. For details on `requestAnimationFrame`, see "Animation Timing" in Chapter 5.

To animate with `canvas` you must draw each frame of your animation separately, clearing the `canvas` (and saving/restoring animation state if needed) between frames. Listing 4-23 illustrates this cycle.

Listing 4-23. Animating with Canvas

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
canvas {
  border: 1px solid #000;
}
        </style>
    </head>
    <body>
      <canvas id="myCanvas" width="500" height="500">Did
You Know: Every time
        you use a browser that doesn't support HTML5,
somewhere a kitten
        cries. Be nice to kittens, upgrade your browser!
      </canvas>
      <script src="drawcycle.js"></script>
      <script>
// Get the context we will be using for drawing.
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

// Set the stroke style.
myContext.strokeStyle = '#000';

// Create a new draw cycle object that we can use for our
animation.
var myDrawCycle = new DrawCycle();

/**
 * Draws a circle of specified radius at the specified
coordinates.
 * @param {number} x The x-coordinate of the center of the
circle.
 * @param {number} y The y-coordinate of the center of the
circle.
 * @param {number} rad The radius of the circle.
 */
function drawCircle(x, y, rad) {
  myContext.beginPath();
  myContext.moveTo(x + rad, y);
```

```
  myContext.arc(x, y, rad, 0, 7);
  myContext.stroke();
}

// Counter for the x-coordinate.
var x = 0;

/**
 * Animates a circle from one corner of the canvas to
another. Used as an
 * animation function for the draw cycle object.
 */
function animateCircle() {
  if (x < 500) {
    myContext.clearRect(0, 0, 500, 500);
    drawCircle(x, x, 10);
    x++;
  } else {
    myDrawCycle.stopAnimation();
  }
}

// Add the animation function to the draw cycle object.
myDrawCycle.addAnimation(animateCircle);

// Begin the animation.
myDrawCycle.startAnimation();
     </script>
    </body>
</html>
```

As mentioned, you'll load your draw cycle constructor before doing any animation. For details on how the draw cycle constructor works, see Chapter 5. This example creates a new instance of the draw cycle, and uses it to manage your animation timing for you.

Start by creating a function that draws a circle at a specified location. Then create your actual animation function that draws the circle in a new location with each cycle. You then register that animation function with the draw cycle, and start the animation. This example simply animates a circle from one corner of the `canvas` to the other.

Because you have to draw each frame separately on the `canvas`, and because the timing of animation frames is so fast, you will quickly run up against efficiency limits. To do complex animations you'll typically need a framework to help you manage

efficiency, provide basic animation functions like physics functions for movement, bouncing, and friction, and to just make it easier to create and manage individual animations.

Interaction

Since `canvas` is an element in the DOM, users can interact with it just like any other DOM element. A `canvas` element will dispatch all of the usual DOM events like mouse events and touch events; you can attach event handlers just as with any other element. However, `canvas` does not dispatch any new events, nor does it provide a way to access anything drawn within.

Using mouse events, it's very easy to create an application that enables users to draw on a `canvas`, as shown in Listing 4-24.

Listing 4-24. Drawing on a Canvas with the Mouse

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
canvas {
  border: 1px solid #000;
  cursor: crosshair;
}
        </style>
    </head>
    <body>
      <canvas id="myCanvas" width="500" height="500">Did
You Know: Every time
        you use a browser that doesn't support HTML5,
somewhere a kitten
        cries. Be nice to kittens, upgrade your browser!
      </canvas>
      <script>
// Get the context we will be using for drawing.
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');
myContext.strokeStyle = '#000';
```

```
// Whether or not the mouse button is being pressed.
var isMouseDown = false;

// Add a mousedown event listener that will set the
isMouseDown flag to true,
// and move the pen to the new starting location.
myCanvas.addEventListener('mousedown', function(event) {
  myContext.moveTo(event.clientX, event.clientY);
  isMouseDown = true;
}, false);

// Add a mouseup event handler that will set the
isMouseDown flag to false.
myCanvas.addEventListener('mouseup', function(event) {
  isMouseDown = false;
}, false);

// Add a mousemove event handler that will draw a line to
the current mouse
// coordinates.
myCanvas.addEventListener('mousemove', function(event) {
  if (isMouseDown) {
    window.requestAnimationFrame(function() {
      myContext.lineTo(event.clientX, event.clientY);
      myContext.stroke();
    });
  }
}, false);
    </script>
  </body>
</html>
```

To draw on the `canvas`, you want to start drawing when the user presses the mouse button, and stop drawing when they release it. So you need mousedown and mouseup event handlers that set a flag indicating the state of the mouse button. The mousedown event handler also moves the pen to the new location, so that you don't accidentally draw a line from the last stopping point to the new starting point. Then you need a mousemove event handler that draws a line to the current mouse pointer coordinates, assuming that the user is holding the mouse button down. To keep things efficient, use the `requestAnimationFrame` method; see "Animation Timing" in Chapter 5 for details on how this method works. Finally, you use a CSS to change the

cursor to a crosshair for the `canvas` element.

As you use the example, you'll notice that it doesn't draw right at the middle of the cursor. Instead, it draws near the lower right corner of the cursor. The mousemove event handler receives its coordinates from the event object that is passed to it by the DOM, and those coordinates are off a bit because the size of the cursor itself is nonzero. To account for this, all you have to do is offset the coordinates by a few pixels—half the width and height of the cursor, to be precise. The new event handler looks like this:

```
// Add a mousemove event handler that will draw a line to
the current mouse
// coordinates, with a slight offset.
myCanvas.addEventListener('mousemove', function(event) {
  if (isMouseDown) {
    window.requestAnimationFrame(function() {
      myContext.lineTo(event.clientX - 7, event.clientY
- 7);
      myContext.stroke();
    });
  }
}, false);
```

Now the example will draw directly under the crosshairs.

Summary

This chapter dove deep into the HTML5 `canvas` element. It covered all of the important features, including:

- drawing shapes and lines
- drawing text
- using `canvas` elements with images
- clipping and masking
- transformations
- basic animation with `canvas` elements
- handling user interactions with `canvas` elements

The HTML5 `canvas` element provides a fairly low-level but flexible API for

drawing directly on web pages. It also enjoys wide support in both desktop and mobile browsers, making it a great candidate for mobile applications.

In Chapter 5 you'll take a look at some JavaScript APIs that are related to HTML5 but not a direct part of the specification.

CHAPTER 5

Related Standards

The HTML5 standard covers a great deal of ground, but it isn't the only new web technology that is being developed by the W3C. There are a family of technologies that are also enhancements to the web platform but don't fall into the category of HTML5. In this chapter, I will cover some of the more exciting new technologies, with a special focus on technologies designed for mobile devices.

Geolocation

SUPPORT LEVEL
Excellent

All modern browsers support these features and have for the last three versions.

W3C Recommendation: http://www.w3.org/TR/geolocation-API/

Geolocation is the ability to determine the physical location of the device hosting the browser, typically in terms of latitude and longitude. Geolocation is very important for mobile devices, where it is used in conjunction with mapping applications, reminders, emergency transponders, and even games (like Ingress; see https://www.ingress.com/).

Devices can determine your location using a combination of technologies:

- **GPS satellites:** Almost all modern smartphones and other mobile devices have transceivers capable of communicating with Global Positioning System satellites.

- **Cellular towers:** Using triangulation algorithms, it's possible to determine the location (broadly) of a cellular device based on its communications with cellular towers.
- **Wi-Fi mapping:** Wi-Fi access points tend to be quite stationary and limited in range, so it is possible to create a "map" of Wi-Fi access points by simply driving around with a Wi-Fi–enabled device. Using such a map, one can determine the approximate location of a given device based on what Wi-Fi access points it has in range.
- **Bluetooth mapping:** Similar to Wi-Fi mapping; best for very close-range geolocation.
- **IP address mapping:** For non-mobile devices, it is possible to determine their location based on their external IP address. Several companies offer IP address mapping services.

All of these methods are imprecise and have their own limitations, but when employed together they can provide an accurate location of the device. However, there is no guarantee that they will return the actual location of the device, or do so with a useful level of accuracy.

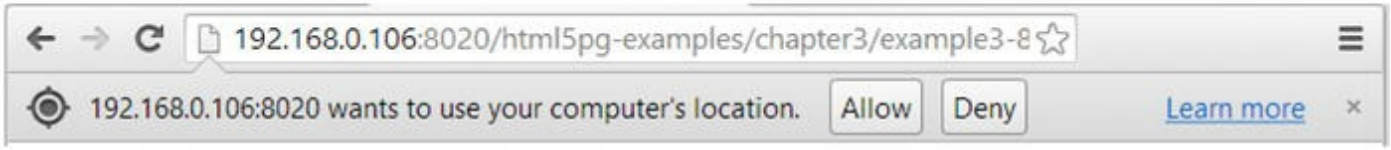
When they work well together, though, it's possible to locate devices quite accurately. This is why most mobile devices will warn you when your Wi-Fi is off that geolocation accuracy will be affected. For example, when you turn off the Wi-Fi radio on an iPhone, iOS will warn you that your location accuracy will be reduced, as shown in Figure 5-1.

Figure 5-1. *iOS location accuracy*

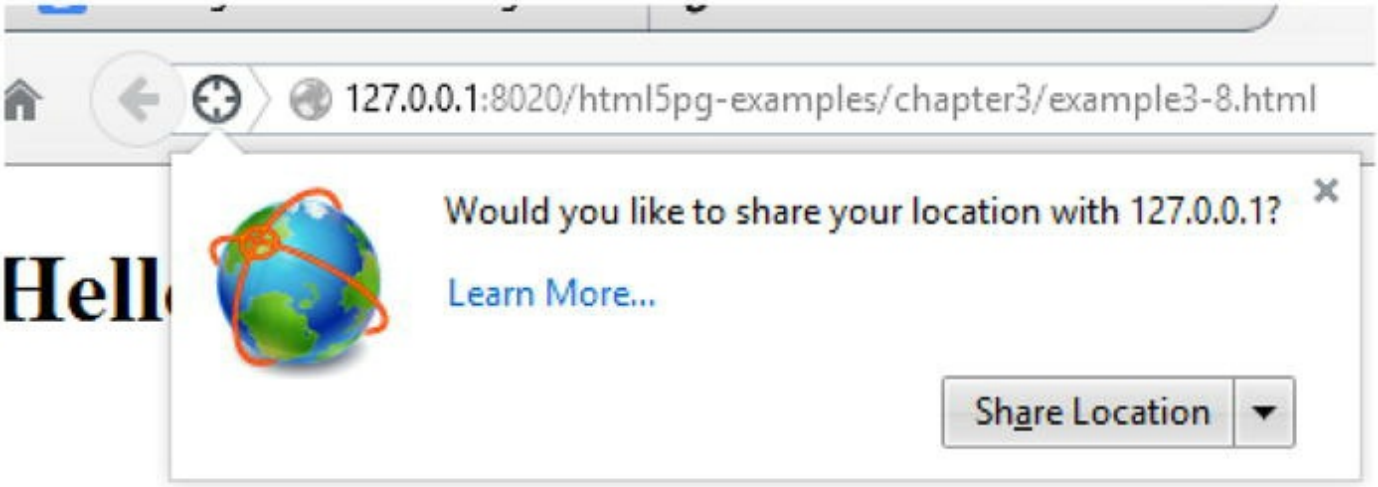
Privacy Considerations

Clearly geolocation has serious privacy implications. Locating and tracking devices—and the people carrying them—is a powerful feature. As a result all browsers have implemented a warning system to inform users that their location is about to be tracked. When your application first accesses the Geolocation API, the browser will inform the user and give them the option to prevent location. These warnings are designed to be conspicuous, but vary from browser to browser (Figure 5-2)

Chrome (Windows)



Firefox (Windows)



Internet Explorer (Windows)

Safari Mobile (iOS)

Figure 5-2. *Geolocation warnings from various browsers*

In all browsers, your script will pause and wait for the user to respond to the dialog. If the user opts to allow geolocation, the script will continue. If the user decides to block geolocation, the API will throw an error.

As you are building a geolocation-capable application, it's important that you consider the privacy and security needs of your users:

- You should only request location data as needed. This is important for both privacy/security and mobile device battery life, as geolocation queries activate multiple radios in a mobile device and can thus be very draining on the battery.
- You should only request just enough geolocation information as needed to fulfill your specific purpose.
- You should only use the information for a specific purpose, and once the purpose is fulfilled you should clear the geolocation data from memory.
- You should be careful how your application shares and transmits geolocation data. Any transmission of geolocation data across any network should be secure to prevent unauthorized access.
- If your application involves sending geolocation data to a server for further processing, you should be even more careful how your server software handles and stores the data, bearing in mind physical security and legal ramifications.

These may seem like obvious guidelines, and in fact they are the basic guidelines for handling any sensitive information. But it's easy to lose sight of these simple ideas while you're busy coding, so be sure to include them in your work from the start.

You should also be transparent with your users that your application collects and processes geolocation data. You should tell them:

- what data you collect;
- why you collect it;
- whether or not you share or transmit the data, and what security measures you take to secure that communication; and
- whether or not you store the data, and what security measures you take to secure that storage. If you do store the information, you should tell them how you secure it and how a user may remove their information from your storage.

If at all possible, you should also provide a way for users to opt out of the geolocation features of your application. Sometimes that's not practical, of course, but providing a way for users to control this feature will do a great deal for establishing trust.

Geolocation API

The Geolocation API specifies a new `navigator.geolocation` object. This object has three new methods that access the geolocation capabilities of the browser and the hosting device. Since it can take an unknown amount of time to resolve the location of the device (the script will pause the first time and wait for the user to respond to the permission dialog before continuing, and then the various location methods have to be queried, each of which can take an unknown amount of time), the methods are asynchronous, and provide a way to register success and error callback functions.

Tip You can use Promises (which are well-supported in mobile browsers) to help simplify the code for asynchronous actions. See the section on Promises in Appendix A.

- `navigator.geolocation.getCurrentPosition(successCa errorCallback, PositionOptions)`: Calls either the `successCallback` when the location is successfully returned or the `errorCallback` if an error occurs. When `successCallback` is called, it will receive a `Position` object as a parameter, and when `errorCallback` is called it will receive a `PositionError` object as a parameter.
- `navigator.geolocation.watchPosition(successCallbac errorCallback, PositionOptions)`: Immediately returns a `PositionWatch` identifier, and then calls the `successCallback` function every time the device's position changes. Calls `errorCallback` if an attempt to resolve the location fails. When `successCallback` is called, it will receive a `Position` object as a parameter, and when `errorCallback` is called it will receive a `PositionError` object as a parameter.
- `navigator.geolocation.clearWatch(PositionWatch)`: Stops a `watchPosition` call specified by the

`PositionWatch` value.

In addition, the API defines three new object templates: the `PositionOptions` object, the `Position` object, and the `PositionError` object. The `PositionOptions` object provides an interface for the `getCurrentPosition` and `watchPosition` methods to fine-tune the query and results, as follows.

```
PositionOptions = {
  // Specifies whether the query should return the most
accurate location possible
  boolean enableHighAccuracy,
  // The number of milliseconds to wait for the device to
return a location
  number timeout,
  // The number of milliseconds a cached value can be
used.
  number maximumAge
}
```

The `Position` object defines the response that will be returned by the `getCurrentPosition` and `watchPosition` methods upon successfully resolving the location of the host device, as follows.

```
Position = {
  object coords : {
    // The latitude in decimal degrees.
    number latitude,
    // The longitude in decimal degrees.
    number longitude,
    // The altitude in meters above nominal sea level.
    number altitude,
    // The accuracy of the latitude and longitude values,
in meters.
    number accuracy,
    // The accuracy of the altitude value, in meters.
    number altitudeAccuracy,
    // The current heading of the device in degrees
clockwise from true north.
    number heading,
    // The current ground speed, in meters per second.
    number speed,
  },
```

```
  // The time when the location query was successfully
created.
  date timestamp
}
```

Note that depending on the browser's implementation of the Geolocation standard and the capabilities of the host device, the values for `altitude`, `accuracy`, `altitudeAccuracy`, `heading`, and `speed` may return as `null`.

The `PositionError` object defines the response that will be returned if the user refuses to allow geolocation, or if somehow the device could not resolve its location, as shown here.

```
PositionError = {
  // The numeric code of the error (see table below).
  number code,
  // A human-readable error message.

  string message
}
```

Valid codes for `PositionError.code` are integers, as listed in Table 5-1.

Table 5-1. *Valid `PositionError` Codes*

Code	Constant	Description
0	`UNKNOWN_ERROR`	The device could not resolve its location due to an unknown error.
1	`PERMISSION_DENIED`	The application does not have permission to use the geolocation services, usually due to the user refusing permission.
2	`POSITION_UNAVAILABLE`	The device could not resolve its location because the services are unavailable. (Typically returned when the various required radios are deactivated, as when a mobile device is in "airplane mode.")
3	`TIMEOUT`	The device could not resolve its location within the timeout limit specified by `PositionOptions.timeout`.

The simplest example of using this API is to do a simple location query and show all of the values that are returned, as demonstrated in Listing 5-1.

Listing 5-1. A Basic Query of the Geolocation API

```
<!DOCTYPE html>
```

```
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
      <h1>Geolocation Example</h1>
      <div id="locationValues">
      </div>
      <div id="error">
      </div>
      <script>
/**
 * The success callback function for getCurrentPosition.
 * @param {Position} position The position object returned
by the geolocation
 *     services.
 */
function successCallback(position) {
  console.log('success')
  // Get a reference to the div we're going to be
manipulating.
  var locationValues
= document.getElementById('locationValues');

  // Create a new unordered list that we can append new
items to as we enumerate
  // the coords object.
  var myUl = document.createElement('ul');

  // Enumerate the properties on the position.coords
object, and create a list
  // item for each one. Append the list item to our
unordered list.
  for (var geoValue in position.coords) {
    var newItem = document.createElement('li');
    newItem.innerHTML = geoValue + ' : '
+ position.coords[geoValue];
    myUl.appendChild(newItem);
  }

  // Add the timestamp.
  newItem = document.createElement('li');
```

```
  newItem.innerHTML = 'timestamp : ' + position.timestamp;
  myUl.appendChild(newItem);

  // Enumeration complete. Append myUl to the DOM.
  locationValues.appendChild(myUl);
}

/**
 * The error callback function for getCurrentPosition.
 * @param {PositionError} error The position error object
returned by the
 *     geolocation services.
 */
function errorCallback(error) {
  var myError = document.getElementById('error');
  var myParagraph = document.createElement('p');
  myParagraph.innerHTML = 'Error code ' + error.code
+ '\n' + error.message;
  myError.appendChild(myParagraph);
}

// Call the geolocation services.
navigator.geolocation.getCurrentPosition(successCallback,
errorCallback);
    </script>
  </body>
</html>
```

First, this example creates a success callback function that enumerates the properties of the `Position` object. As it does so it adds them to an unordered list that is appended to the DOM so you can see it. The error callback behaves the same way, except instead of producing a list it simply updates the contents of a paragraph.

The first time you run this example your browser should prompt you for permission to access the geolocation APIs. The first time through, deny permission, so you can see what an error condition looks like. Figure 5-3 shows what the resulting page looks like in Chrome.

Figure 5-3. *Error condition for Listing 5-1 in Chrome*

You can see that the error handler was called with an error code of 1. The actual text for the error message varies from browser to browser (Internet Explorer 11, for example, uses the error message "This site does not have permission to use the Geolocation API.") but the error code is the same.

The Geolocation specification does not define the permission model that must be presented to the user, which is why every browser does it differently. The specification simply says,

> *User agents must not send location information to Web sites without the express permission of the user. User agents must acquire permission through a user interface, unless they have prearranged trust relationships with users, as described below. The user interface must include the host component of the document's URI. Those permissions that are acquired through the user interface and that are preserved beyond the current browsing session (i.e. beyond the time when the browsing context is navigated to another URL) must be revocable and user agents must respect revoked permissions.*
>
> *Some user agents will have prearranged trust relationships that do not require such user interfaces. For example, while a Web browser will present a user interface when a Web site performs a geolocation request, a VOIP telephone may not present any user interface when using location information to perform an E911 function.*

As a result, how a user can grant or refuse geolocation permission, how long that decision is remembered, and how a user can change their mind later, are all up to the browser manufacturer to decide and implement.

In Internet Explorer, for example, the user is presented with a pop-up that allows them some interesting options, as shown in Figure 5-4.

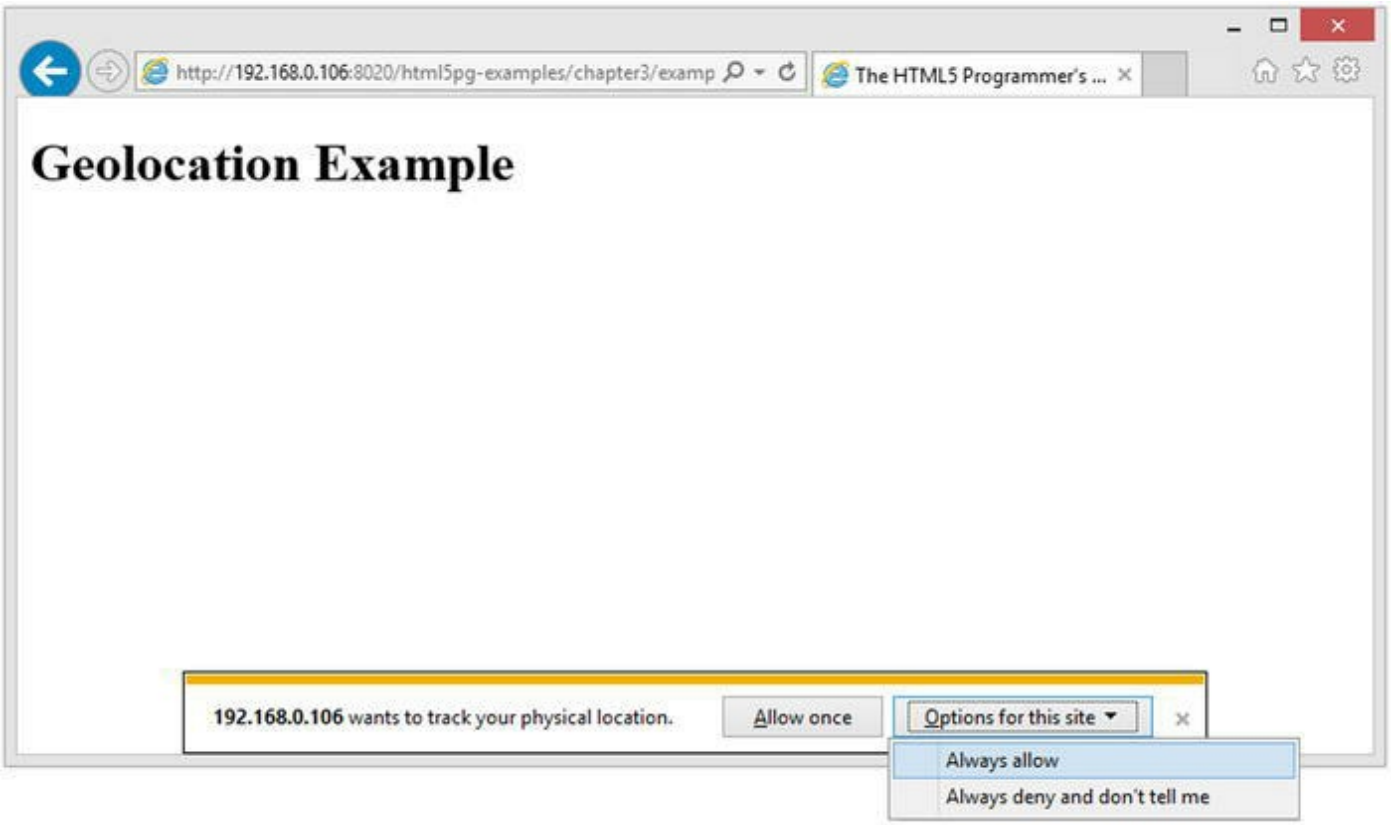


Figure 5-4. *Geolocation permission options in Internet Explorer 11*

If the user chooses "Allow once" or "Always allow", the script will continue and the browser will attempt to resolve the client's location. The option "Allow once" should probably read "Allow for this browsing session", because the permission remains in effect until the user closes and restarts the browser. At that point, revisiting the page will reprompt the user. The option "Always allow" functions as you would expect: once the user picks it, they will never again be prompted for permission. The option "Always deny and don't tell me" denies permission at that point and every subsequent time the user visits that page. They are never reprompted for permission, and the only way they can undo this decision is to open the Internet Options dialog for Windows, choose the Privacy tab, and click the "Clear sites" button in the Location section—which clears all permanent permissions granted or denied to all sites.

Firefox presents a completely different interaction to the user, as shown in Figure 5-5.

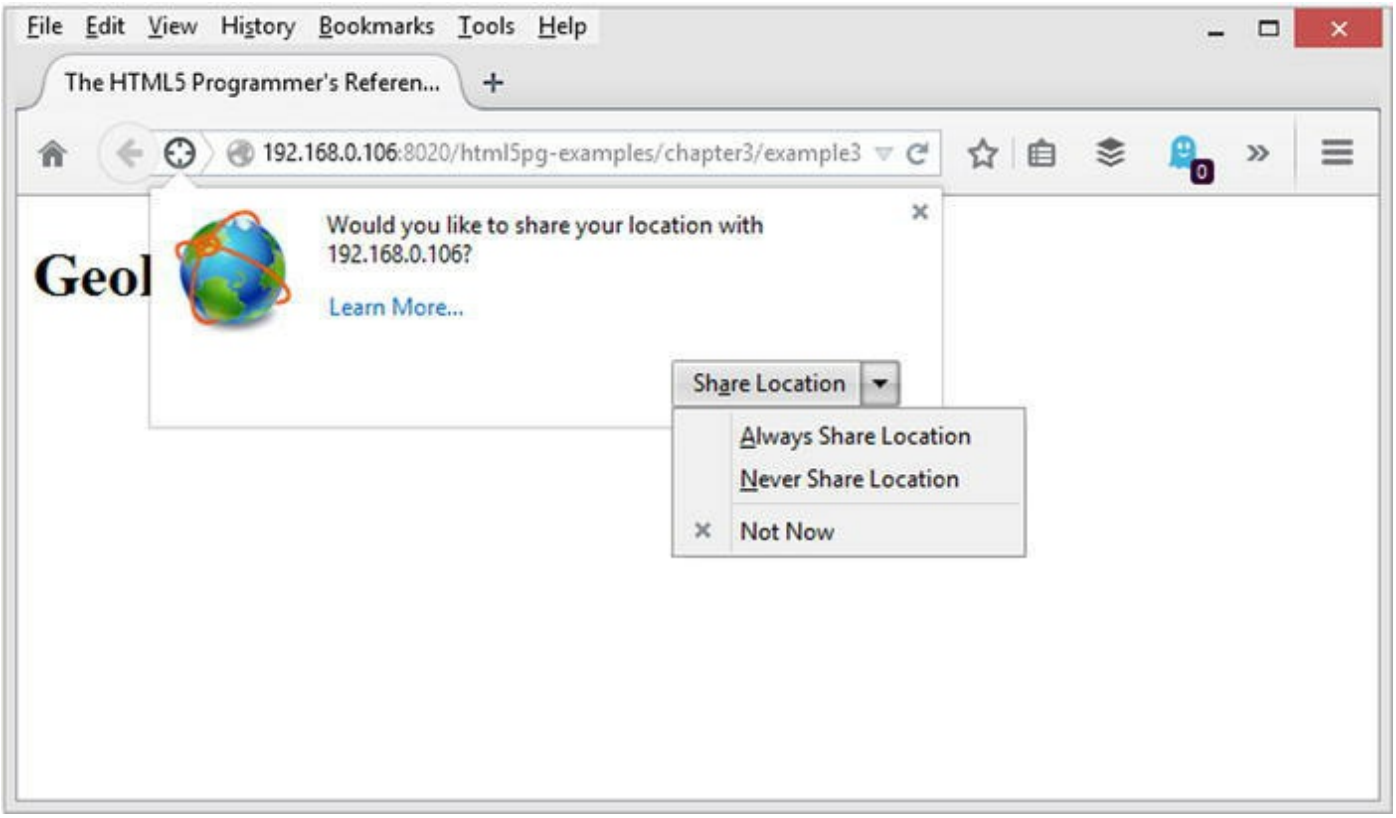


Figure 5-5. *Geolocation permission options in Firefox 29*

If the user chooses "Share Location" the script will continue and the browser will attempt to resolve the client's location. Unlike with Internet Explorer, however, this permission is not for the current browser session but only for the current visit to the web site. Reloading the page will immediately prompt the user for permission again. The user does not have to restart the browser. The "Always Share Location" option grants permanent permission to share location, and "Never Share Location" acts as a permanent denial of permission for the page. Choosing "Not Now" or clicking on the × icon in the upper right corner of the pop-up, or clicking anywhere outside of the pop-up, will close the pop-up without either granting or denying permission and will leave your application hanging. The pop-up can be reopened by clicking the "target" icon next to the URL, but that's not necessarily immediately obvious. This behavior is by design; see the relevant Bugzilla bug, `https://bugzilla.mozilla.org/show_bug.cgi?id=675533`, for an explanation.

Only in Safari Mobile on iOS is the permission pop-up an actual modal pop-up that requires the user to respond and cannot be dismissed unless they make a choice. In all other cases, the user can ignore (and in the case of Firefox completely dismiss) the pop-up and leave your script waiting to execute a callback. To make matters worse, time spent in this undefined state does not count toward any timeout you may have specified with `PositionOption.timeout`—that timer only begins running after the user has

granted permission and the browser has begun trying to resolve the location.

To get around this, you need to implement a global timeout timer that starts running as soon as the script accesses the Geolocation API. If the user does grant (or deny) permission, our regular callbacks should happen and this global timer should be canceled. If the user does not grant (or deny) permission, the global timer should execute a callback that does something—for example, redirect the browser to an error page that explains to the user what they need to do to continue. Or if your application doesn't require GPS, the global timer callback should cancel the success and error callbacks and your application can continue.

It's easy to add such a global timer to Listing 5-1, as shown in Listing 5-2.

Listing 5-2. Registering a Global Timeout

```
<!DOCTYPE html>
<html>
    <head>
       <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
      <h1>Geolocation Example</h1>
      <div id="locationValues">
      </div>
      <div id="error">
      </div>
      <script>
// Create the variable that will hold the timer reference.
var globalTimeout = null;

/**
 * The success callback function for getCurrentPosition.
 * @param {Position} position The position object returned
by the geolocation
 *     services.
 */
function successCallback(position) {
  // Check the state of the global timeout. If it is null,
the application has
  // timed out and we should not continue. If it isn't
null, the timeout timer
  // is still running, so we should cancel it and
continue.
```

```
  if (globalTimeout == null) {
    return;
  } else {
    clearTimeout(globalTimeout);
  }

  // Get a reference to the div we're going to be
manipulating.
  var locationValues
= document.getElementById('locationValues');

  // Create a new unordered list that we can append new
items to as we enumerate
  // the coords object.
  var myUl = document.createElement('ul');

  // Enumerate the properties on the position.coords
object, and create a list
  // item for each one. Append the list item to our
unordered list.
  for (var geoValue in position.coords) {
    var newItem = document.createElement('li');
    newItem.innerHTML = geoValue + ' : '
+ position.coords[geoValue];
    myUl.appendChild(newItem);
  }

  // Add the timestamp.
  newItem = document.createElement('li');
  newItem.innerHTML = 'timestamp : ' + position.timestamp;
  myUl.appendChild(newItem);

  // Enumeration complete. Append myUl to the DOM.
  locationValues.appendChild(myUl);
}

/**
 * The error callback function for getCurrentPosition.
 * @param {PositionError} error The position error object
returned by the
 *     geolocation services.
 */
function errorCallback(error) {
```

```
  // Check the state of the global timeout. If it is null,
the application has
  // timed out and we should not continue. If it isn't
null, the timeout timer
  // is still running, so we should cancel it and
continue.
  if (globalTimeout == null) {
    return;
  } else {
    clearTimeout(globalTimeout);
  }
  var myError = document.getElementById('error');
  var myParagraph = document.createElement('p');
  myParagraph.innerHTML = 'Error code ' + error.code
+ '\n' + error.message;
  myError.appendChild(myParagraph);
}

/**
 * The callback to execute if the whole process times out,
specifically in the
 * situation where a user ignores the permissions pop-ups
long enough.
 */
function globalTimeoutCallback() {
  alert('Error: GPS permission not given, exiting
application.');
  globalTimeout = null;
}

// Call the geolocation services.
navigator.geolocation.getCurrentPosition(successCallback,
errorCallback);

// Start the timer for the global timeout call.
globalTimeout
= setTimeout(globalTimeoutCallback.bind(this), 5000);
    </script>
  </body>
</html>
```

The first thing this example does is define a `globalTimeout` variable, which will hold the identifier for the timer it will start when it initiates the geolocation

request. Next, notice that in both the `successCallback` and `errorCallback` functions, it checks the state of the `globalTimeout` variable. If the variable is `null`, the global timeout has expired, and the code should not continue to execute those functions. If it isn't `null`, the timer is still active, so the code should cancel it and continue.

Next it provides a `globalTimeoutCallback` function that simply alerts a message to the user. In an actual application you would want to do something more useful here—redirect the user to another page, for example. The code also sets the `globalTimeout` variable to `null` so that if either of the callbacks should get executed somehow, they will not continue past the initial global timeout check.

Finally, it sets the timer running immediately after it calls the geolocation API. The timer is set to five seconds. When you load this page, you'll see one of the following:

- If you have permanently denied geolocation permission to the page, the `errorCallback` will execute and the global timer will be canceled. No permission pop-up will be displayed.
- If you have permanently allowed geolocation permission to the page, the `successCallback` will execute and the global timer will be canceled. No permission pop-up will be displayed.
- If you haven't permanently granted or denied permission, the permission pop-up will display. You can choose to grant or deny permission before the global timeout timer expires, in which case the appropriate callback will execute and the global timer will be canceled. Or you can do nothing and wait for the global timer to expire. When that happens, the alert message will appear.

In any case, you cannot programmatically force a permission choice for the user. They have to make their permission choice through the browser-supplied dialog.

From a user interaction standpoint, this is a somewhat unfortunate state of affairs because it means your application will cause the browser to display a notification over which you have no control. Some users might find this alarming and choose to deny permission, or even shut down the browser entirely and never return to your application. If you have been transparent with your users about how your application collects and stores geolocation information, they will be prepared for this interaction and will be more willing to grant permission, because they know what your application will be doing with the data.

Animation Timing

SUPPORT LEVEL
Good

All modern browsers support these features and have for the last two versions.

W3C Candidate Recommendation: `http://www.w3.org/TR/animation-timing/`

The Animation Timing standard is designed to help you build JavaScript-based visual animations. If you have ever tried to build an animation by hand using JavaScript, you're probably familiar with the simple pattern of a draw cycle:

- Create a draw function that is responsible for incrementally "drawing" the animated items: positioning elements, changing element properties, drawing on a `canvas` element, and so forth. Each time this function is called, it produces an entire animation "frame," just as if you were drawing animation frames by hand that would then be shown in a film.
- Call the draw function every few milliseconds.

A JavaScript draw cycle is typically implemented using a timer, which calls a drawing function every few milliseconds. An example can be seen in Listing 5-3.

Listing 5-3. A JavaScript Implementation of a Timer-Based Draw Cycle

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
#target-element {
  width: 100px;
  height: 100px;
  background-color: #ccc;
  position: absolute;
```

```
  top: 100px;
  left: 0px;
}
      </style>
    </head>
    <body>
      <h1>Simple Animation Example</h1>
      <div id="target-element"></div>
      <script>
// Get a reference to the element we want to move.
var targetEl = document.getElementById('target-element');

// Create a variable to keep track of its position.
var currentPosition = 0;

/**
 * Draws the animation by updating the position on the
target element and incrementing
 * the position variable by 1.
 */
function draw() {
  if (currentPosition > 500) {
    // Stop the animation, otherwise it would run
indefinitely.
    clearInterval(animInterval);
  } else {
    // Update the element's position.
    targetEl.style.left = currentPosition++ + 'px';
  }
}

// Initiate the animation timer.
var animInterval = setInterval(draw, 17);
     </script>
    </body>
</html>
```

This example uses a JavaScript timer to update the position of a div on the page. The interval between updates is 17 milliseconds. That's not an arbitrary number. Most monitors refresh at 60Hz, and so most browsers try and limit their screen repaints to no more than 60Hz. Sixty cycles per second is about 17 milliseconds between cycles. Any faster than that and you lose "frames."

Depending on the browser you use to run this example, and the system you are using, this animation can appear to be quite smooth or somewhat jerky. That's because this is a brute-force method of animation, and it doesn't take into account how the browser redraws the page. It just commands the screen to be updated, and the browser has to do the best it can. Also, there's no guarantee that the time between animation updates will be 17 milliseconds. The `setInterval` method just adds the updates to the browser's UI queue, which can easily become bogged down if the browser is busy doing something else (like resizing the window, or possibly fetching and rendering other content in the background), thus delaying the screen render.

Overall this method doesn't scale well. As animations increase in number and complexity, and the pages they are in also increase in complexity and interactive capability, these timer-based animation queues become more and more inefficient.

The Animation Timing specification addresses the problems with JavaScript-based timers by providing a new timer: `requestAnimationFrame`. Syntactically this method is used similarly to the existing JavaScript timer methods `setInterval` and `setTimeout`. Behind the scenes, though, the new method is tied to the browser's screen management algorithms. As a result, `requestAnimationFrame` has some important benefits:

- Animations queued with `requestAnimationFrame` are optimized by the browser into a single reflow/repaint cycle.
- Animations queued with `requestAnimationFrame` play well with animations from other sources, like CSS transitions.
- The browser will stop animations in browser tabs that are not visible. This is important on mobile devices, where intensive animations can rapidly consume battery power.

The specification creates two new methods in the global context:

- `requestAnimationFrame(callback)`: Request that the function `callback` be executed as part of the next animation cycle. The callback will receive as a parameter a timestamp. Like `setTimeout` and `setInterval`, `requestAnimationFrame` returns an identifier that can be used to stop the cycle.
- `cancelAnimationFrame(identifier)`: cancel the animation frame request identified by the identifier.

Updating Listing 5-3 to use `requestAnimationFrame` is easy, as shown in

Listing 5-4.

Listing 5-4. Listing 5-3 Rewritten Using `requestAnimationFrame`

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
#target-element {
  width: 100px;
  height: 100px;
  background-color: #ccc;
  position: absolute;
  top: 100px;
  left: 0px;
}
        </style>
    </head>
    <body>
      <h1>Simple requestAnimationFrame Example</h1>
      <div id="target-element"></div>
      <script>
var targetEl = document.getElementById('target-element');
var currentPosition = 0;

/**
 * Updates the position on the target element, the
increments the position
 * counter by 1.
 */
function animateElement() {
  // Stop the animation, otherwise it would run
indefinitely.
  if (currentPosition <= 500) {
    requestAnimationFrame(animateElement);
  }
  // Update the element's position.
  targetEl.style.left = currentPosition++ + 'px';
}

// Initiate the animation timer.
```

```
animateElement();
      </script>
    </body>
</html>
```

This example updates the `animateElement` function to use `requestAnimationFrame`. Each time that method is called, it updates the position of the element and increments the position counter. It also schedules itself for calling again via `requestAnimationFrame`. Once the element reaches the position of 500px, the animation stops.

Building a draw cycle manager using Animation Timing is also quite easy. A draw cycle manager will allow you to register animation functions (like the `animateElement` function in Listing 5-4), and start, stop, and pause the draw cycle. Listing 5-5 shows a simple draw cycle manager.

Listing 5-5. A Draw Cycle Manager

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
.animatable {
  width: 100px;
  height: 100px;
  background-color: #ccc;
  position: absolute;
  top: 110px;
  left: 0px;
}
#elementTwo {
  top: 220px;
}
        </style>
    </head>
    <body>
      <h1>Simple Animation Framework Example</h1>
      <div class="animatable" id="elementOne"></div>
      <div class="animatable" id="elementTwo"></div>
      <button id="startAnimation">Start Animation</button>
      <button id="togglePause">Toggle Pause</button>
```

```
    <button id="stopAnimation">Stop Animation</button>
    <button id="registerOne">Register Animation
One</button>
    <button id="unregisterOne">Unregister Animation
One</button>
    <button id="registerTwo">Register Animation
Two</button>
    <button id="unregisterTwo">Unregister Animation
Two</button>
    <script>

// Get references to the elements we will be animating,
and create position
// tracking variables for them.
var elementOne = document.getElementById('elementOne');
var elOnePosition = 0;
var elementTwo = document.getElementById('elementTwo');
var elTwoPosition = 0;

/**
 * Animates Element One by incrementally updating its left
position. Animation
 * stops at 500px.
 */
function animateElementOne() {
  if (elOnePosition <= 500) {
    elementOne.style.left = elOnePosition++ + 'px';
  } else {
    // Done animating, so remove this animation from the
draw cycle manager.
    myCycle.removeAnimation(animateElementOne);
    // Reset the counter so we can animate again. The
function can be
    // re-registered and will work as before.
    elOnePosition = 0;
  }
}

/**
 * Animates Element Two by incrementally updating its left
position. Animation
 * stops at 500px.
```

```
 */
function animateElementTwo() {
  if (elTwoPosition <= 500) {
    elementTwo.style.left = elTwoPosition++ + 'px';
  } else {
    // Done animating, so remove this animation from the
draw cycle manager.
    myCycle.removeAnimation(animateElementTwo);
    // Reset the counter so we can animate again. The
function can be
    // re-registered and will work as before.
    elTwoPosition = 0;
  }
}

/**
 * Creates a draw cycle object that will repetitively draw
animation functions.
 * @constructor
 * @returns {Object} A new draw cycle object.
 */
var DrawCycle = function() {
  var newCycle = {
    /**
     * The identifier for the current animation frame
loop.
     * @type {Number}
     */
    animationPointer: null,

    /**
     * @type {Boolean}
     */
    isPaused: false,

    /**
     * The array of animation callbacks.
     * @type {!Array.<Function>}
     */
    arrCallbacks: [],

    /**
     * Starts the animation cycle.
```

```
     */
    startAnimation: function() {
      // Like other JavaScript timers,
requestAnimationFrame sets the execution
      // context of its callbacks to the global execution
context (the window
      // object). We need the execution context to be
'this', the newCycle
      // object we're creating. By using the bind method
(which exists on
      // Function.prototype) we are able to override the
default execution
      // context with the one we need.
      this.animationPointer
= window.requestAnimationFrame(this.draw.bind(this));
    },

    /**
     * Stops the animation cycle.
     */
    stopAnimation: function() {
      window.cancelAnimationFrame(this.animationPointer);
    },

    /**
     * Pauses the invocation of the animation functions
each draw cycle. If set
     * to true, the animation functions will not be
invoked. If set to false,
     * the functions will be invoked.
     * @type {Boolean}
     */
    pauseAnimation: function(boolPause) {
      this.isPaused = boolPause;
    },

    /**
     * Adds an animation function to the draw cycle.
     * @param {Function}
     */
    addAnimation: function(callback) {
      if (this.arrCallbacks.indexOf(callback) == -1) {
```

```
        this.arrCallbacks.push(callback);
      }
    },

    /**
     * Removes an animation function from the draw cycle.
     * @param {Function}
     */
    removeAnimation: function(callback) {
      var targetIndex
= this.arrCallbacks.indexOf(callback);
      if (targetIndex > -1) {
        this.arrCallbacks.splice(targetIndex, 1);
      }
    },

    /**
     * Draws any registered animation functions (assuming
they are not paused)
     * and then kicks off another animation cycle.
     * You should not need to call this method directly.
     * @private
     */
    draw: function() {
      if (!this.isPaused) {
        var i = 0, arrCallbacksLength
= this.arrCallbacks.length;
        for (i = 0; i < arrCallbacksLength; i++) {
          this.arrCallbacks[i]();
        }
      }
      this.startAnimation();
    }
  };
  return newCycle;
};

// Create a new draw cycle object.
var myCycle = new DrawCycle();

// Register a callback for the Start Animation button that
starts the animation
// cycle.
```

```
var startAnimation
= document.getElementById('startAnimation');
startAnimation.addEventListener('click', function() {
  myCycle.startAnimation();
}, false);

// Register a callback for the Pause Animation button that
pauses/unpauses the
// animation cycle.
var pauseAnimation
= document.getElementById('togglePause');
pauseAnimation.addEventListener('click', function() {
  myCycle.pauseAnimation(!myCycle.isPaused);
}, false);

// Register a callback for the Stop Animation button that
stops the animation
// cycle.
var stopAnimation
= document.getElementById('stopAnimation');
stopAnimation.addEventListener('click', function() {
  myCycle.stopAnimation();
}, false);

// Register a callback for the Register Animation One
button that adds the
// animation function for element one to the draw cycle
object.
var registerOne = document.getElementById('registerOne');
registerOne.addEventListener('click', function() {
  myCycle.addAnimation(animateElementOne);
}, false);

// Register a callback for the Unregister Animation One
button that removes the
// animation function for element one from the draw cycle
object.
var unregisterOne
= document.getElementById('unregisterOne');
unregisterOne.addEventListener('click', function() {
  myCycle.removeAnimation(animateElementOne);
}, false);
```

```
// Register a callback for the Register Animation Two
button that adds the
// animation function for element two to the draw cycle
object.
var registerTwo = document.getElementById('registerTwo');
registerTwo.addEventListener('click', function() {
  myCycle.addAnimation(animateElementTwo);
}, false);

// Register a callback for the Unregister Animation Two
button that removes the
// animation function for element two from the draw cycle
object.
var unregisterTwo
= document.getElementById('unregisterTwo');
unregisterTwo.addEventListener('click', function() {
  myCycle.removeAnimation(animateElementTwo);
}, false);
    </script>
  </body>
</html>
```

This example creates a constructor function that gives you a new draw cycle object, which provides a simplified API for handling animations. The main API methods are:

- `addAnimation(animationFunction)`: Registers an animation function with the draw cycle. Every time the draw cycle runs, `animationFunction` will be invoked.
- `removeAnimation(animationFunction)`: Deregisters an animation function with the draw cycle.
- `startAnimation()`: Starts the animation drawing cycle. When called, this method will call `requestAnimationFrame` with the object's `draw` method as the callback, thus initiating a single loop. The method stores the identifier for the loop so that it can later be cancelled if desired.
- `stopAnimation()`: Stops the animation drawing cycle. When called, this method calls `cancelAnimationFrame` with the identifier stored by `startAnimation`.
- `pauseAnimation(boolPause)`: Pauses or unpauses calling the registered animation functions. The draw cycle still runs but

none of the animation functions are invoked.

Using this animation API is simple:

1. Create a new instance of a draw cycle using the constructor function.
2. Register one or more animation callbacks that you want to be called every draw cycle.
3. Start the animation cycle.

When you call `startAnimation` it requests an animation frame from the browser, with `draw` method as the callback. The browser invokes the `draw` method at the appropriate time. The `draw` method invokes all of the registered animation functions (assuming animation is not paused), thus completing one cycle. It then calls `startAnimation` to kick off a new cycle.

You can dynamically add new animation functions as desired; they will automatically be invoked in the next draw cycle. You can also remove animation functions as needed. Each animation method also removes itself from the draw cycle when it completes, and resets its counter. You can reregister the animation functions at that point and the animations will happen again. Note that the draw cycle will continue to run even if there are no animation functions registered, so when you remove the last animation function you also should be sure to call the `stopAnimation` method.

Selectors

SUPPORT LEVEL
Excellent

All modern browsers support these features and have for the last four versions.

W3C Candidate Recommendation: `http://www.w3.org/TR/selectors/`

The new Selectors standard provides new ways for accessing elements in the DOM. Previously the main ways for accessing elements in the DOM was either to use the `getElementById` method, to use traversal, or some combination of the two. With the new Selectors standard, you can access elements directly based on their CSS selectors.

The Selectors standard took cues from popular JavaScript frameworks like jQuery, which have made heavy use of selectors. If you're at all familiar with jQuery, Prototype, Dojo, or any other JavaScript library that uses selectors, you'll find the new Selectors API to be very familiar.

The Selectors standard defines two new methods on the Element abstract class:

- `querySelector(cssSelectorList)`: Returns a direct reference to the first element that matches all of the CSS selectors in the specified comma-delimited `cssSelectorList`. If there is no match, return `null`.
- `querySelectorAll(cssSelectorList)`: Returns a `NodeList` object containing all the matches to the CSS selectors specified in the comma-delimited `cssSelectorList`. If no elements match, return a `NodeList` with no members.

Note `NodeList` objects look a lot like arrays, in that they have member elements that can be accessed via their numeric index, and a `length` property that reflects the number of members. However, `NodeList` objects inherit directly from the `Object` prototype, rather than the `Array` prototype, so they do not have any of the array methods you might expect (e.g., `Array.forEach`).

Using the new Selectors API you can easily get direct references to DOM elements without extensive traversal, and without adding IDs to your markup that are only ever used for JavaScript selectors. This can help you keep both your markup and JavaScript code clean. In addition, you'll often find yourself using the same selectors both in your JavaScript and in your CSS, because often the elements you need to style are the same elements your scripts need to access.

I've been using the Selectors API throughout examples in the book. Here are some other examples that help illustrate how powerful the API can be:

- Attribute Selectors: `[attribute=value]` allows you to target DOM elements based on their assigned attributes. This is particularly useful in selecting elements that have data attributes assigned to them. You can also use pattern matching:
    - `[att^='val']` selects elements whose `att` attribute begins with the letters "val"
    - `[att$='lue']` selects elements whose `att` attribute

ends with the letters “lue”

- `[att*='val']` selects elements whose `att` attribute contains the letters “val”

- Element State Pseudo-classes allow you to target DOM elements based on their state pseudo-classes. Particularly useful are `:enabled` (selects form fields that are enabled), `:disabled` (selects disabled form fields), and `:checked` (selects checkboxes and radio buttons that are checked).
- Negation Pseudo-class: `not(selector)` targets DOM elements that do not match the specified selector.
- Structural Pseudo-classes allow you to target DOM elements based on their location in the DOM structure. Particularly useful are:
    - `:nth-child(n)` selects the element that is the nth child of its parent
    - `:nth-last-child(n)` selects the element that is the nth child of its parent, counting from the last child backward
    - `:nth-of-type(n)` selects the element that is the nth sibling of its type
    - `:nth-last-of-type(n)` selects the element that is the nth sibling of its type, counting from the last sibling backward
    - `:last-child` selects the last child element of a parent element
    - `:first-of-type` and `:last-of-type` select the sibling element that is the first or the last of its type
    - `:only-child` selects elements that are the only child of their parents

Since the `querySelector` and `querySelectorAll` methods are Element methods, you can use them on any element. This limits the search for matching selectors to the descendants of that element, as shown in Listing 5-6.

Listing 5-6. Limiting a Selector Query to a Containing Element

```
<!DOCTYPE html>
```

```
<html>
    <head>
       <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
      <p class="selectme">This has the selectme class, but
will not be clickable.</p>
      <div class="noselect">
        <p class="selectme">This has the selectme class,
but will not be clickable.</p>
      </div>
      <div class="selectable">
        <p class="selectme">This has the selectme class,
and will be clickable.</p>
      </div>
      <script>
// Get a reference to the containing element we want to
search.
var selectable = document.querySelector('.selectable');

// Get a reference to the paragraph.
var targetPar = selectable.querySelector('.selectme');

// Give the target paragraph an event handler for the
click event.
targetPar.addEventListener('click', function() {
  alert('I was clicked!');
});
      </script>
    </body>
</html>
```

This simple example limits the search for the desired selector to the specified `div` element. This is the equivalent of using the selector `".selectable .selectme"`. This technique is particularly useful for selecting the descendants of event targets.

Device Orientation

SUPPORT LEVEL

Poor

This API is only useful on devices with necessary hardware, which are typically mobile devices. Support for this API in mobile browsers is quite good, except for Internet Explorer Mobile, which does not implement the API at all. Internet Explorer 11 does support the API, as do Chrome and Firefox, but Safari does not.

W3C Working Draft: `http://www.w3.org/TR/orientation-event/`

Most mobile and handheld devices contain sensitive gyroscopes that enable the device to be aware of its orientation in space. The Device Orientation API provides a standard API for host devices to share this information with browser-based applications.

Caution This particular standard has undergone frequent changes in response to industry feedback. As a result, most browser manufacturers have held back on full implementation. I'm presenting the standard in its current incarnation at press time because I believe this is an important feature that deserves coverage despite its draft status.

The standard specifies a new set of events that fire on the window object, as well as new data properties on the resulting event object. By registering event listeners for these events, you can gain access to these data properties.

The `compassneedscalibration` Event

According to the standard, the `compassneedscalibration` event fires on the `window` object "when the user agent determines that a compass used to obtain orientation data is in need of calibration." However, there is no instruction for what the user agent should do to calibrate itself, or how to communicate this to either the developer or the end users. For this reason, this event is currently disabled in Firefox (see `https://bugzilla.mozilla.org/show_bug.cgi?id=738121`). Other mobile user agents may fire this event, though I have never seen it happen.

Like any other event, you simply register an event handler for it on the `window` object, as shown in this code snippet:

```
window.addEventListener('compassneedscalibration',
function(event) {
```

```
  alert('Your compass needs calibration. Wave your device
in a figure-8 motion.');
}, false);
```

The `deviceorientation` Event

According to the standard, the `deviceorientation` event fires on the `window` object "whenever a significant orientation change occurs," but leaves the definition of "significant change" up to the browser manufacturer. In practice, this event appears to fire regularly on the `window` object, even for devices that are completely at rest on a table.

The event object for the `deviceorientation` event is a `DeviceOrientationEvent` object, which has the following properties:

`DeviceOrientationEvent.alpha`: The alpha angle of rotation.

`DeviceOrientationEvent.beta`: The beta angle of rotation.

`DeviceOrientationEvent.gamma`: The gamma angle of rotation.

If you're familiar with Euler Angles, the alpha, beta, and gamma angles are Tait-Bryan angles of the type Z-X'-Y". To visualize these angles, imagine a device sitting flat on a table, as shown in Figure 5-6.

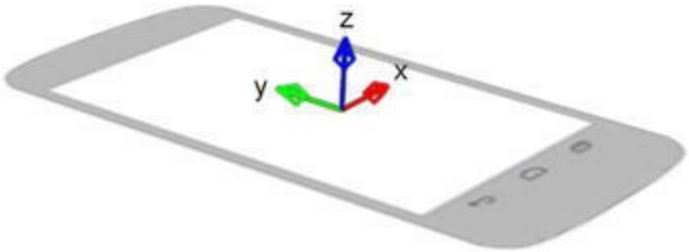


Figure 5-6. *A device sitting flat on a table*

Rotating about the z axis will translate both the x and y axes by the amount of the rotation, as shown in Figure 5-7.

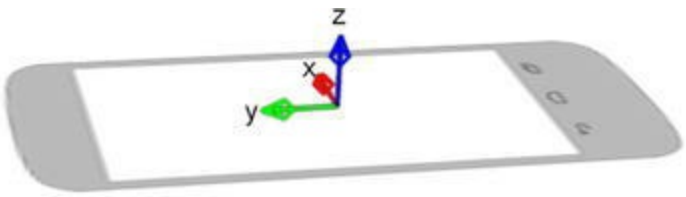


Figure 5-7. *Rotation about the z axis*

The resulting angle is referred to as the *alpha angle*.

Rotating about the x axis will translate both the z and y axes by the amount of the rotation, as shown in Figure 5-8.

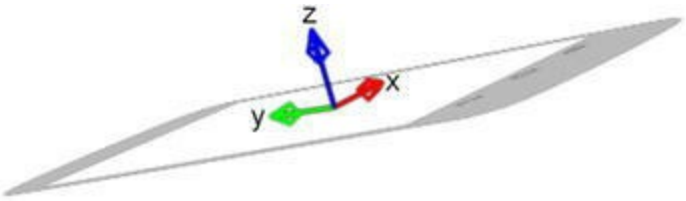


Figure 5-8. *Rotation about the x axis*

The resulting angle is referred to as the *beta angle*.

Finally, rotating about the y axis will translate both the x and z axes by the amount of the rotation, as shown in Figure 5-9.

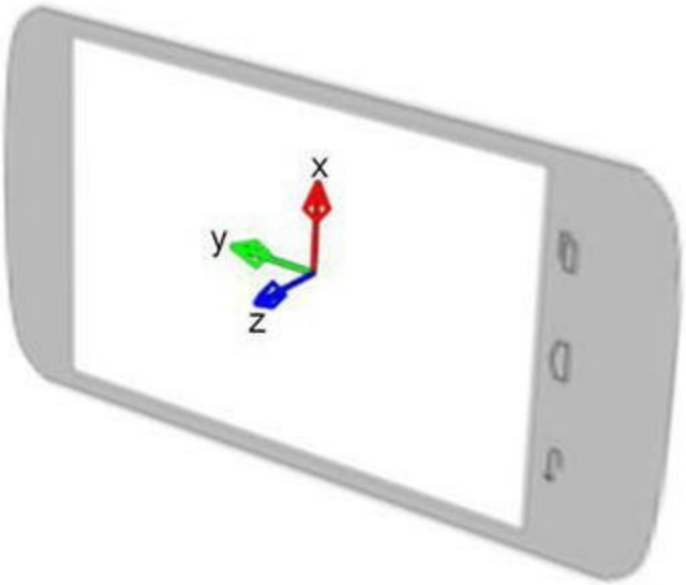


Figure 5-9. *Rotation about the y axis*

The resulting angle is referred to as the *gamma angle*.

The definitive example of how to use these angles is to move a DOM element on the screen according to the gamma and beta angles. Because the angles vary from positive

to negative, you can simply add the rounded value of the angle to the current value of the associated ordinate: for the x ordinate you use the gamma angle, and for the y ordinate you use the beta angle. The more the device is tilted, the larger the angle, the greater the increment on the coordinate, and the faster the element will move, as shown in Listing 5-7.

Listing 5-7. Moving a Ball on the Screen

```
<!DOCTYPE html>
<html>
    <head>
      <meta name="viewport" content="width=device-width,
user-scalable=no">
       <title>The HTML5 Programmer's Reference</title>
       <style>
#container {
  position: absolute;
  top: 220px;
  left: 50px;
  width: 204px;
  height: 204px;
  border: 1px solid red;
}
#ball {
  width: 10px;
  height: 10px;
  position: absolute;
  top: 0px;
  left: 0px;
  background-color: red;
  border-radius: 50%;
}
       </style>
    </head>
    <body>
      <h1>Device Orientation Demonstration</h1>
      <ul>
        <li>Alpha: <span id="alpha"></span></li>
        <li>Beta: <span id="beta"></span></li>
        <li>Gamma <span id="gamma"></span></li>
        <li>y-pos <span id="ypos"></span></li>
```

```
      <li>x-pos <span id="xpos"></span></li>
    </ul>
    <div id="container">
      <div id="ball"></div>
    </div>
    <script>
// Get references to the various DOM elements we will be
manipulating.
var alpha = document.getElementById('alpha');
var beta = document.getElementById('beta');
var gamma = document.getElementById('gamma');
var ypos = document.getElementById('ypos');
var xpos = document.getElementById('xpos');
var ball = document.getElementById('ball');

// Initialize x and y coordinates.
var yposit = 0;
var xposit = 0;

/**
 * Handles a deviceorientation event on the window object.
 * @param {DeviceOrientationEvent} event A standard device
orientation event.
 */
function handleDeviceOrientation(event) {
  // Update the DOM with the raw event data.
  alpha.innerHTML = event.alpha;
  beta.innerHTML = event.beta;
  gamma.innerHTML = event.gamma;
  // Use the raw data to get x and y coordinates for the
ball.
  xposit = getCoord(event.gamma, xposit);
  xpos.innerHTML = xposit;
  yposit = getCoord(event.beta, yposit);
  ypos.innerHTML = yposit;
  ball.style.top = yposit + 'px';
  ball.style.left = xposit + 'px';
}

/**
 * Increments a coordinate based on an angle from the
device orientation event.
```

```
 * @param {number} angle The orientation angle.
 * @param {number} coord The coordinate to increment.
 */
function getCoord(angle, coord) {
  // First, get a delta value from the angle.
  var delta = Math.round(angle);
  var tempVal = coord + delta;
  // Limit the incremented value to between 0 and 194.
  if (tempVal > 0) {
    coord = Math.min(194, tempVal);
  } else {
    coord = 0;
  }
  return coord;
}

// Register the event handler.
window.addEventListener('deviceorientation',
handleDeviceOrientation, false);
     </script>
    </body>
</html>
```

This example displays the raw event data on the screen, and uses that raw event data to determine the coordinates of the element on the screen. In this case, it limits the position of the element so that it stays inside of its containing element.

The `devicemotion` Event

The `devicemotion` event fires regularly on the `window` object, and produces an event of type `DeviceMotionEvent`. The `DeviceMotionEvent` has four properties: `acceleration` (the values of which represent the acceleration of the device along the x, y, and z axes, in meters per second squared), `accelerationIncludingGravity` (the values of acceleration with the effects of the Earth's gravity included, if any), `rotationRate` (the rate of rotation of the alpha, beta, and gamma angles in degrees per second), and `interval` (how often this information is refreshed from the hardware, in milliseconds). Overall the schema of the `DeviceMotionEvent` looks like this:

```
object DeviceMotionEvent = {
  object acceleration: {
```

```
    number x,
    number y,
    number z
  },
  object accelerationIncludingGravity: {
    number x,
    number y,
    number z
  },
  object rotationRate: {
    number alpha,
    number beta,
    number gamma
  }
  number interval
}
```

You can easily display each of these values, as shown in Listing 5-8.

Listing 5-8. Displaying the Values of a `devicemotion` Event

```
<!DOCTYPE html>
<html>
    <head>
      <meta name="viewport" content="width=device-width,
user-scalable=no">
        <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
      <h1>Device Motion Demonstration</h1>
      <ul>
        <li>acceleration:
          <ul>
            <li id="accX">x: <span class="current"></span>,
<br>
              max: <span class="max"></span></li>
            <li id="accY">y: <span class="current"></span>,
<br>
              max: <span class="max"></span></li>
            <li id="accZ">z: <span class="current"></span>,
<br>
              max: <span class="max"></span></li>
```

```
        </ul>
      </li>
      <li>accelerationIncludingGravity:
        <ul>
          <li id="aigX">x: <span class="current"></span>,
<br>
            max: <span class="max"></span></li>
          <li id="aigY">y: <span class="current"></span>,
<br>
            max: <span class="max"></span></li>
          <li id="aigZ">z: <span class="current"></span>,
<br>
            max: <span class="max"></span></li>
        </ul>
      </li>
      <li>rotationRate:
        <ul>
          <li id="rrAlpha">alpha: <span class="current">
</span>,<br>
            max: <span class="max"></span></li>
          <li id="rrBeta">beta: <span class="current">
</span>,<br>
            max: <span class="max"></span></li>
          <li id="rrGamma">gamma: <span class="current">
</span>,<br>
            max: <span class="max"></span></li>
        </ul>
      </li>
    </ul>
    <script>
// Create a data structure to store the references to the
various DOM elements
// we will be manipulating, as well as associated maximum
values. The structure
// also includes an interface method for processing
incoming data and mapping
// it to the correct DOM elements.
var motionValues = {
  acceleration : {
    x : {
      domCurr : document.querySelector('#accX .current'),
      domMax : document.querySelector('#accX .max'),
```

```
      maxVal : 0
    },
    y : {
      domCurr : document.querySelector('#accY .current'),
      domMax : document.querySelector('#accY .max'),
      maxVal : 0
    },
    z : {
      selector : '#accZ',
      domCurr : document.querySelector('#accZ .current'),
      domMax : document.querySelector('#accZ .max'),
      maxVal : 0
    }
  },
  accelerationIncludingGravity : {
    x : {
      domCurr : document.querySelector('#aigX .current'),
      domMax : document.querySelector('#aigX .max'),
      maxVal : 0
    },
    y : {
      domCurr : document.querySelector('#aigY .current'),
      domMax : document.querySelector('#aigY .max'),
      maxVal : 0
    },
    z : {
      selector : '#accZ',
      domCurr : document.querySelector('#aigZ .current'),
      domMax : document.querySelector('#aigZ .max'),
      maxVal : 0
    }
  },
  rotationRate : {
    alpha : {
      domCurr : document.querySelector('#rrAlpha
.current'),
      domMax : document.querySelector('#rrAlpha .max'),
      maxVal : 0
    },
    beta : {
      domCurr : document.querySelector('#rrBeta .current'),
      domMax : document.querySelector('#rrBeta .max'),
```

```
      maxVal : 0
    },
    gamma : {
      selector : '#accZ',
      domCurr : document.querySelector('#rrGamma
.current'),
      domMax : document.querySelector('#rrGamma .max'),
      maxVal : 0
    }
  },

  /**
   * Processes an acceleration value object of a specific
type. The values are
   * enumerated and mapped to their associated DOM
elements for display.
   * @param {string} valueType The type of the value
object, one of
   *     'acceleration', 'accelerationIncludingGravity',
or 'rotationRate'.
   * @param {object} valueObject The object containing the
acceleration data.
   */
  processValues : function(valueType, valueObject) {
    // First, get a reference to the subproperty of the
motionValues object we
    // will be manipulating.
    var mvRef = this[valueType];
    // Enumerate the valueObject and process each
property.
    for (property in valueObject) {
      // Convenience references to the current values we're
working with.
      var currMVRef = mvRef[property];
      var currVal = valueObject[property];
      // Update the DOM to display the current value.
      currMVRef.domCurr.innerHTML = currVal;
      // If the current value is larger than the last
stored maximum value,
      // update the stored max value to match and display
it in the DOM.
      if (currVal > currMVRef.maxVal) {
```

```
        currMVRef.maxVal = currVal;
        currMVRef.domMax.innerHTML = currVal;
      }
    }
  }
};

/**
 * Handles a devicemotion event on the window object.
 * @param {DeviceMotionEvent} event A standard device
motion event object.
 */
function handleDeviceMotion(event) {
  motionValues.processValues('acceleration',
event.acceleration);
  motionValues.processValues('accelerationIncludingGravity'
     event.accelerationIncludingGravity);
  motionValues.processValues('rotationRate',
event.rotationRate);
}

// Register the event handler.
window.addEventListener('devicemotion',
handleDeviceMotion, false);
    </script>
  </body>
</html>
```

Because there are many values to display, and much of the data is specifically structured thanks to the `DeviceMotionEvent` schema, Listing 5-8 begins this example by creating an object that has a similar schema. For each individual property it stores a DOM reference to the element that will display its current value, a DOM reference to the element that will display the maximum value achieved, and the maximum value itself. It also includes a simple interface method that maps the `DeviceMotionEvent` subproperties to their associated subproperties in the object, and updates the DOM to reflect the new information.

To use this example you need to move your device around. These values are for acceleration, which is the rate of change of velocity (while velocity is the rate of change of position). In order to see appreciable values you will need to move your device fairly quickly. It's sufficient to shake your device along the various axes of motion. Be careful to keep a firm grip on your device so you don't accidentally throw it. The maximum values of acceleration along the various axes will be recorded for you so you

can see them after you're done moving your device around. You can also spin the device to see rotation rates.

WebGL

SUPPORT LEVEL
Good

All modern desktop browsers support these features for at least the last two versions, with the exception of Internet Explorer, which has only supported them since version 11. Mobile support is poor, as Mobile Safari for iOS does not currently support WebGL, though Apple has committed to full support with iOS version 8.

Specifications: `http://www.khronos.org/webgl/`

The Web Graphics Library (WebGL) is an API for drawing complex 2d and 3d graphics in HTML `canvas` elements. The WebGL API is presented as a drawing context on a given `canvas` element, just like the standard drawing context that you explored in Chapter 4. Just like the standard `canvas` drawing context, the WebGL drawing context is accessible in JavaScript via an extensive API. Many WebGL tasks, such as image processing, are delegated to the host system's Graphics Processing Unit and are not handled by the system's main CPU, thus providing a significant speed boost.

Unlike most of the other standards covered in this book, the WebGL standard is not maintained by either the W3C or the WHATWG. The standard is maintained by the nonprofit technology consortium Khronos Group. The language itself is based on the OpenGL language, and grew out of experiments in 3d rendering done at Mozilla in 2009. The current stable release of WebGL is 1.0.2. Work started on WebGL 2 in 2013.

Initializing a WebGL drawing context is very similar to initializing a standard drawing context in a `canvas` element, as shown in Listing 5-9.

Listing 5-9. Initializing a WebGL Drawing Context

```
<!DOCTYPE html>
<html>
    <head>
```

```
        <title>The HTML5 Programmer's Reference</title>
        <style>
canvas {
  border: 1px solid #000;
}
        </style>
    </head>
    <body>
      <canvas id="myCanvas" width="200" height="200">Did
You Know: Every time
        you use a browser that doesn't support HTML5,
somewhere a kitten
        cries. Be nice to kittens, upgrade your browser!
      </canvas>
      <script>
var myCanvas = document.getElementById('myCanvas');
var myGlContext = myCanvas.getContext('webgl');
      </script>
    </body>
</html>
```

This example uses the `getContext` method just as you did in Chapter 4. The difference is that instead of providing a parameter of `'2d'` for a 2d drawing context, it provides the `'webgl'` parameter to specify a WebGL drawing context. You can easily expand this to be a function, which even provides a place for initializing the context as needed, as shown in Listing 5-10.

Listing 5-10. A WebGL Initialization Function

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
canvas {
  border: 1px solid #000;
}
        </style>
    </head>
    <body>
      <canvas id="myCanvas" width="200" height="200">Did
```

```
You Know: Every time
       you use a browser that doesn’t support HTML5,
somewhere a kitten
       cries. Be nice to kittens, upgrade your browser!
     </canvas>
     <script>
/**
 * Returns a WebGL drawing context on a specified canvas
element. If opt_setup
 * is provided and set to true, this method also performs
some basic
 * initialization on the context.
 * @param {!Element} targetCanvas The reference to the
desired canvas element.
 * @param {boolean} opt_setup Whether or not to perform
additional setup on the
 *     context.
 * @return {Object} The WebGL drawing context, or null if
WebGL is not supported
 *     or was otherwise unavailable.
 */
function initWebGLOnCanvas(targetCanvas, opt_setup) {
  // If opt_setup was not specified, set it to false. In
JavaScript, null and
  // undefined are == to each other and nothing else, so:
  if (opt_setup == null) {
    opt_setup = false;
  }

  // Try and get the context.
  var glContext = targetCanvas.getContext(’webgl’);
  if (glContext == null) {
    // Try falling back to an experimental version, works
on some older browsers.
    glContext = targetCanvas.getContext(’experimental-
webgl’);
    if (glContext == null) {
      // We were unable to get a WebGL context. Provide
a warning diagnostic
      // message on the console, in case anyone is looking.
      console.warn(’WebGL is not supported in this
browser.’);
```

```
    }
  }

  // If there is a context and setup was requested, do the
setup.
  if ((opt_setup === true) && (glContext != null)) {
    // Set the clear color to black (rgba).
    glContext.clearColor(0.0, 0.0, 0.0, 1.0);
    // Initialize the depth function so that objects that
are closer in
    // perspective hide things that are further away.
    glContext.depthFunc(glContext.EQUAL);
    // Enable depth testing.
    glContext.enable(glContext.DEPTH_TEST);
    // Clear both the color and the depth buffer.
    glContext.clear(glContext.COLOR_BUFFER_BIT|glContext.DE
  }
  return glContext;
}

var myCanvas = document.getElementById('myCanvas');
var myGLContext = initWebGLOnCanvas(myCanvas, true);
     </script>
    </body>
</html>
```

This example expands the initialization function to detect if there has been a problem getting the WebGL context, with a fallback to an older syntax that is present on older browsers. If the context cannot be fetched at all, a warning is output to the console. Running this example will produce a black square in the browser.

As of this writing, Firefox is currently blacklisting a significant number of Windows, MacOS, Linux, and Android graphics drivers in the WebGL initialization process. If you have one of these drivers, Firefox by default will not initialize a WebGL drawing context. If you run this example in Firefox and you see the warning message in the console, chances are your setup is blacklisted. For details and instructions on how to override the block, see https://wiki.mozilla.org/Blocklisting/Blocked_Graphics_Dri Another alternative is to use a browser that has a less brittle WebGL implementation (Chrome's implementation of WebGL is quite solid).

Note Up until recently, Safari Mobile on iOS did not support WebGL. Safari 8.1

introduced full WebGL support.

WebGL is an extensive language, and fully covering it and everything you can do with it is beyond the scope of this book. If you want to learn more, check out *Beginning WebGL for HTML5* by Brian Danchilla (Apress, 2012).

SVG

SUPPORT LEVEL
Excellent

All modern browsers support SVG and have for at least the last three versions.

W3C Recommendation: `http://www.w3.org/TR/SVG11/`

Scalable Vector Graphics (SVG) is a graphics format for creating raster graphics, vector graphics, and text. Graphics objects (both defined in SVG and imported from external files, such as regular image files) can be grouped and manipulated easily using SVG.

Most graphics formats (like the Portable Network Graphics [PNG] format) consist of binary data. SVG graphics are defined using XML markup, and so can be easily created using simple text editors, just like web pages. Since SVG graphics are defined in XML markup, the contents can easily be scanned and indexed. This gives SVG the potential to be significantly more accessible than other graphics formats.

As mentioned, SVG markup can produce raster graphics just as `canvas` elements can. It can also produce vector graphics, which are graphics defined by mathematical functions involving points, lines, and curves. The primary difference between raster graphics and vector graphics is that vector graphics scale better than raster graphics. SVG-defined vector graphics are therefore a great choice for mobile applications because they will remain crisp at any resolution and size.

As with WebGL, SVG is a large standard, and fully covering it is beyond the scope of this text.

Summary

In this chapter, I explored some of the JavaScript APIs that aren't a part of the HTML5 standard but are often used in conjunction with HTML5 features. Many of them have exciting mobile uses as well.

- The Geolocation API gives your JavaScript applications access to mobile devices' geolocation features. You can use this API to write exciting new mobile apps that are location aware. I also covered important privacy considerations when using geolocation.
- Animation Timing provides tools for making smooth animations by giving new timers that are directly related to the painting of the browser window.
- The Selectors API provides a way to easily access DOM elements using CSS selectors.
- The Device Orientation API gives your JavaScript applications access to the orientation features of mobile devices. You can use this API to create applications that respond to movements of the hosting mobile device.
- Finally, I briefly touched upon two new and exciting technologies, WebGL and SVG.

Using these APIs with HTML5 features will enable you to build exciting and dynamic applications on a wide range of devices.

In Chapter 6 you'll dive into practical development with HTML5, including building an entire HTML5 mobile game from scratch.

CHAPTER 6

Practical HTML5

Now that I have covered HTML5 and its related technologies, it's time for you to build something with them. This chapter will focus on working with HTML5. It will cover issues of browser compatibility, including feature detection, polyfills and shims, and libraries designed to work with HTML5. Last, you will work through a full HTML5 project, from start to finish.

I will begin by defining the requirements for the project, work out how best to implement it, and then break down the implementation method by method.

Browser Support

The biggest barrier to using HTML5 is browser support. If you're working on a project that has to support a lot of older browsers, you will rapidly run into lack of support in your major browsers. This is particularly a problem for desktop browsers; Internet Explorer didn't even support the basic semantic tags of HTML5 until version 9. Mobile browsers have quite good implementation of HTML5 features because they tend to come from newer code bases. However, even mobile browsers have support issues. For example, many phones running the Android operating system earlier than version 4 will have browsers that don't support several modern features.

How your application handles browser support is an important decision. You could decide to support only the latest version of every browser. This would guarantee that your application would have access to the widest range of HTML5 features, but might shut out users who are stuck on older browsers or operating systems.

Far more common is the requirement that your application support browser versions that go back a few revisions at least. This means your users will be trying to use your application in browsers that might not support the HTML5 features you need. In such a situation you will have to choose how your application should behave, but the initial

choice will be based on detecting whether or not the feature is supported. Using scripts to determine if a given feature is supported is called "feature detection," and you can easily test most HTML5 features.

A Crash Course in Feature Detection

Feature detection is an important tool for HTML developers that allows you to customize your application based on what the current browser is capable of doing. There are a variety of feature detection techniques based on how the feature in question is implemented—for example, as a property or method on an existing object or as a new element type.

Detecting JavaScript Properties and Methods

Many of the new HTML5 JavaScript APIs are implemented as new properties or methods on existing objects such as `window`, `document`, or `navigator`. If you try and access these features in browsers that don't support them, the JavaScript engine will produce an error and your script will come to a grinding halt, as demonstrated in Listing 6-1.

Listing 6-1. Invoking a Method That Doesn't Exist

```
<!DOCTYPE html>
<html>
    <head>
       <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
    <h1>Feature Detection</h1>
      <script>
// Attempt to invoke a fake method foo() on the window
object
window.foo()
      </script>
    </body>
</html>
```

When you run this example, it will produce an error, which you can see if you have the browser's JavaScript console open. If this were in the middle of a larger script, it

would bring the entire script to a halt, which is a pretty catastrophic result.

When you attempt to access a property that doesn't exist, the result is a little more subtle. Simply reading a nonexistent property will return the value of `undefined` but will not actually crash the script, as shown in Listing 6-2.

Listing 6-2. Accessing a Property That Doesn't Exist

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
    <h1>Feature Detection</h1>
      <script>
// Attempt to access a fake property bar on the window
object
alert(window.bar);
      </script>
    </body>
</html>
```

If you run this script it will work perfectly, and the alert pop-up will contain the text "undefined." The script will not throw an error and will continue to run. In JavaScript, however, `undefined` is a specific value and its own data type, so if you attempt to manipulate it further (as you would if you were accessing a real property), the results can be surprising, as demonstrated in Listing 6-3.

Listing 6-3. What Exactly Is `undefined`?

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
    <h1>Feature Detection</h1>
      <script>
// Compare undefined with basic data types
alert(window.bar + 5);
alert(window.bar + ' is its own data type');
```

```
alert(window.bar == true);
alert(window.bar == false);
if (window.bar) {
  alert('undefined is equal to true');
}
if (!window.bar) {
  alert('undefined is equal to false');
}
if (window.bar == null) {
  alert('undefined and null are equal');
}
if (!(window.bar === null)) {
  alert('undefined and null are not strictly equal');
}
    </script>
  </body>
</html>
```

When you run this script you'll see that `undefined` doesn't play well with numbers; even a simple addition operation produces the value `NaN` (for "Not a Number"). And although `undefined` doesn't exist it has a string value of "undefined." And while `undefined` is not equal to either `true` or `false`, it evaluates as `false` for purposes of flow control. Finally, you can test how `undefined` and `null` equate to each other using both the type-converting or "weak" equality operator (==) and the strict equality operator (===). The weak equality operator automatically resolves type differences between the operands, while the strict equality operator does not. In the case of `undefined` and `null`, the two values are equal to one another when using the weak operator, but since they have different fundamental data types they fail to pass the strict equality test.

Tip `undefined` vs. `null`: It's important to remember that while these behaviors may be counterintuitive, they are in fact well defined by the ECMAScript Standard and are actual features of the language. Just remember that `undefined` as a value is meant to indicate any property that has not been assigned a value, while `null` is meant to indicate an intentional absence of value.

To fully explain these behaviors, I'd have to dive into a discussion of JavaScript data types and how the language resolves data type differences for the weak equality operator ==, which is a bit beyond the scope of this chapter. Regardless, Listing 6-3 does demonstrate a way to detect the presence of a property on a JavaScript object with

predictable results. As shown in Listing 6-4, this method also works for detecting methods, and doesn't throw an error.

Listing 6-4. Detecting Properties and Methods on JavaScript Objects

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
    <h1>Feature Detection</h1>
      <script>
// To check for a method foo() on the window object, check
to see if it is
// defined
if (window.foo) {
  alert('Method foo() is available');
} else {
  alert('Method foo() is not available');
}

// To check for a property bar on the window object use
the same test.
if (window.bar) {
  alert('Property bar is available');
} else {
  alert('Property bar is not available');
}
      </script>
    </body>
</html>
```

When you run Listing 6-4 it will show that neither `window.foo()` nor `window.bar` are available, and the script will throw no errors. It's easy to use this method to detect real HTML5 features, as shown in Listing 6-5.

Listing 6-5. Detecting HTML5 JavaScript APIs

```
<!DOCTYPE html>
<html>
```

```
    <head>
       <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
    <h1>Feature Detection</h1>
      <script>
if (window.postMessage) {
  alert('The postMessage feature is available on this
browser!');
} else {
  alert('The postMessage feature is not available on this
browser');
}

if (window.localStorage) {
  alert('The localStorage feature is available on this
browser!');
} else {
  alert('The localStorage feature is not available on this
browser');
}
     </script>
    </body>
</html>
```

When you run this example you will find out whether or not the `postMessage` and `localStorage` features are available on your browser.

This same method works to detect the new HTML5 event interfaces, such as the new device motion and orientation events. Instead of checking for the presence of the event handler (e.g., `ondevicemotion`) directly, check to see if the event interface is present (e.g., `window.DeviceMotionEvent` as shown in Listing 6-6).

Listing 6-6. Detecting Support for Event Interfaces

```
<!DOCTYPE html>
<html>
    <head>
       <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
    <h1>Feature Detection</h1>
      <script>
```

```
if (window.DeviceMotionEvent) {
  alert('This browser supports the device motion API!');
} else {
  alert('This browser does not support the device motion
API.');
}
     </script>
    </body>
</html>
```

Detecting Support for New HTML5 Elements

There are two main ways to detect support for the new elements:

- Create an instance of the element and then test the result for expected properties and methods. If the browser does not know how to implement the element, then the expected properties will be `undefined`. This test is useful for elements like `canvas` and `video`, which implement their own unique properties and methods.
- Create an instance of the element and then test the interface it implements. If the browser does not know how to implement the element, it will implement the `HTMLUnknownElement` interface (see hereafter for details). This test is useful for elements that do not implement unique properties and methods, such as structural elements.

Listing 6-7 demonstrates the first method.

Listing 6-7. Detecting Support for the Canvas Element

```
<!DOCTYPE html>
<html>
    <head>
       <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
    <h1>Feature Detection</h1>
      <script>

// Test for canvas support.
```

```
var testCanvas = document.createElement('canvas');
if (testCanvas.getContext) {
  alert('This browser supports the canvas element!');
} else {
  alert('This browser does not support the canvas
element.');
}

// We are done with the test element, so delete it.
testCanvas = null;

    </script>
  </body>
</html>
```

This example creates a `canvas` element and then tests for the presence of the `getContext` method. If the browser knows how to implement the `canvas` element properly, the method will be present, otherwise it will be `undefined`.

Tip Creating elements for testing without attaching them to the DOM is a fairly safe thing to do. These elements exist in memory (and thus take up physical memory space) but are not part of the DOM and will not affect the rest of your document. Because they take up memory, it's always a good idea to remove them when they are no longer needed by setting their reference to null.

Detecting expected properties and methods only works for elements that implement properties or methods that are unique outside of basic element properties and methods. What about elements like `article` or `aside` that don't implement unique properties or methods? The answer lies in the interface hierarchy defined by the HTML standard.

The HTML standard defines a base interface called `HTMLElement` with a set of properties and methods common to all HTML elements: `title`, `lang`, `focus`, `blur`, and so on. The standard also defines a set of child interfaces that inherit from it, such as `HTMLDivElement`, `HTMLTitleElement`, and the like. Most supported elements inherit from these child interfaces and so share the base properties and methods of the `HTMLElement` interface. The standard also defines a child interface for unsupported elements called `HTMLUnknownElement`. You can create any arbitrary element using `document.createElement`; if the element is not supported it will inherit from the `HTMLUnknownElement` interface.

Determining which interface a particular element implements is a simple matter of checking the element's `toString` method. When you call that method on an element it

will output the name of the interface that it implements, as Listing 6-8 demonstrates.

Listing 6-8. Determining the Interface That an HTML Element Implements

```
<!DOCTYPE html>
<html>
    <head>
       <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
    <h1>Feature Detection</h1>
      <script>

// Create a div.
var myDiv = document.createElement('div');
alert(myDiv.toString());

// Create a fake element.
var myFake = document.createElement('itsafake');
alert(myFake.toString());

// Delete elements now that they are no longer needed.
myDiv = myFake = null;
     </script>
    </body>
</html>
```

This example creates two elements, a `div` and a fake element, and then calls each element's `toString` method. As you can see, the `itsafake` element implements the `HTMLUnknownElement` interface. This gives you an easy test for unsupported elements, as shown in in Listing 6-9.

Listing 6-9. Testing for Supported Elements

```
<!DOCTYPE html>
<html>
    <head>
       <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
    <h1>Feature Detection</h1>
      <script>
```

```
// Test for support for the article element.
var myArticle = document.createElement('article');
if (myArticle.toString().indexOf('HTMLUnknownElement') > -
1) {
  alert ('This browser does not support the article
element.');
} else {
  alert('This browser supports the article element.');
}

// Create a fake element and test for support.
var myFake = document.createElement('itsafake');
if (myFake.toString().indexOf('HTMLUnknownElement') > -1)
{
  alert('This browser does not support the itsafake
element');
} else {
  alert('This browser supports the itsafake element');
}

myArticle = myFake = null;
     </script>
   </body>
</html>
```

In this example you test for support for the `article` element as well as for the `itsafake` element by checking for the presence of the substring `'HTMLUnknownElement'` within the value returned by each element's `toString` method.

Detecting Support for New Element Properties

HTML5 also defines a whole new set of properties that can be applied to elements, such as `placeholder` or `draggable`. Detecting support for these properties is simple: just create an element and set the desired property, then test to see if the property has maintained its value. When you set the value, be sure to set it to the proper type; some properties (such as `autocomplete` and `placeholder`) will expect strings as values, and others (such as `autofocus` and `draggable`) will require boolean values. If you set the incorrect type in the test, it will produce a false negative. Listing 6-10 demonstrates using this technique to test for support for the

placeholder attribute on input elements.

Listing 6-10. Testing for Attribute Support

```
<!DOCTYPE html>
<html>
    <head>
       <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
    <h1>Feature Detection</h1>
      <script>

// Test for support for the new placeholder attribute on
input elements.
var testPlaceholderText = 'Test placeholder text';
var testInput = document.createElement('input');
testInput.setAttribute('placeholder',
testPlaceholderText);
if (testInput.placeholder === testPlaceholderText) {
  alert ('This browser supports the placeholder attribute
on input elements');
} else {
  alert ('This browser does not support the placeholder
attribute on input elements');
}

// To prove the method works, test for an attribute we
know doesn't exist.
testInput.setAttribute('fakeattr', testPlaceholderText);
if (testInput.fakeattr === testPlaceholderText) {
  alert ('This browser supports the fakeattr attribute on
input elements');
} else {
  alert ('This browser does not support the fakeattr
attribute on input elements');
}

// We are done with the test element, so delete it.
testInput = null;

     </script>
```

```
    </body>
</html>
```

This example also includes a demonstration that it will fail in the case of an unsupported attribute (in this case you just make up a fake attribute and test for it).

Clearly this technique requires the target object to implement a `setAttribute` method. As a result it cannot be used to detect features on the `window` or `navigator` elements, which do not have a `setAttribute` method.

Building a Feature Detection Script

Now that you know how to test for various HTML5 features, you can build a single script that tests for everything. Start by creating a constructor function that, when called, will run the feature detection tests and return an object that contains all of the results. Each result will be a named property on the object set to either `true` or `false` depending on the support for that feature. The object will also have three convenience methods:

- `getTests`: This method will return an alphabetized array of all of the features that were tested.
- `getTestResults`: This method will return an array consisting of all of the results for all of the features that were tested. A single result will consist of an object with a `feature` property set to the name of the feature and an `isSupported` property that will be set to `true` or `false` depending on whether or not the feature is supported.
- `getFailedTestResults`: This method will return an array consisting of all of the results for all of the features that failed their tests and are not supported in the current browser.

Listing 6-11 gives the full listing of the detection script.

Listing 6-11. HTML5 Feature Detection Script

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
```

```
table {
  font-family: verdana,arial,sans-serif;
  color: #333;
  border-width: 1px;
  border-color: #666;
  border-collapse: collapse;
}
th {
  border-width: 1px;
  padding: 8px;
  border-style: solid;
  border-color: #666;
  background-color: #dedede;
}
td {
  border-width: 1px;
  padding: 8px;
  border-style: solid;
  border-color: #666;
  background-color: #fff;
}

      </style>
    </head>
    <body>
    <h1>Feature Detection</h1>
    <table id="supported">
      <tr><th>Feature</th><th>Support</th></tr>
    </table>
      <script>

/**
 * Detects support for various HTML5 features.
 * @constructor
 * @returns {Object} An object with properties for each
feature, each
 *     set to true or false depending on support.
 */
function DetectHTML5Support() {
  var returnVal = {};

  // Test for HTML5 APIs on the window object.
```

```
  var apisToTest = ['EventSource', 'postMessage',
'sessionStorage',
      'localStorage', 'Worker', 'requestAnimationFrame',
      'cancelAnimationFrame', 'DeviceMotionEvent',
'DeviceOrientationEvent'];
  for (i = 0; i < apisToTest.length; i++) {
    var currApi = apisToTest[i];
    returnVal[currApi] = (window[currApi] != undefined);
  }

  // Test for HTML5 APIs on the navigator object.
  var apisToTest = ['geolocation'];
  for (i = 0; i < apisToTest.length; i++) {
    var currApi = apisToTest[i];
    returnVal[currApi] = (navigator[currApi] !=
undefined);
  }

  // Test for HTML5 APIs on the document object.
  var apisToTest = ['querySelector', 'querySelectorAll'];
  for (i = 0; i < apisToTest.length; i++) {
    var currApi = apisToTest[i];
    returnVal[currApi] = (document[currApi] != undefined);
  }

  // Test for suport for the new HTML5 elements.
  var unsupported = 'HTMLUnknownElement';
  var elementsToTest = ['article', 'aside', 'nav',
'footer', 'header',
      'section', 'figure', 'figcaption', 'main', 'bdi',
'data', 'mark',
      'ruby', 'rp', 'rt', 'time', 'wbr', 'dialog',
'details', 'summary',
      'datalist', 'meter', 'output', 'progress', 'audio',
'canvas', 'video'];
  for (i = 0; i < elementsToTest.length; i++) {
    var currItem = elementsToTest[i];
    var testEl = document.createElement(currItem);
    returnVal[currItem]
= (testEl.toString().indexOf(unsupported) == -1);
    testEl = null;
  }
```

```
  // Test for support for new input properties that are
booleans.
  var propsToTest = ['autofocus', 'draggable'];
  var inputEl = document.createElement('input');
  // For variety we'll use Array.forEach to run these
tests instead of an
  // explicit for loop.
  propsToTest.forEach(function(currProp) {
    var testValue = true;
    inputEl.setAttribute(currProp, testValue);
    returnVal[currProp] = (inputEl[currProp] ===
testValue);
  }, this);

  // Test for support for new input properties that are
strings.
  propsToTest = ['autocomplete', 'placeholder'];
  propsToTest.forEach(function(currProp) {
    var testValue = 'testval';
    inputEl.setAttribute(currProp, testValue);
    returnVal[currProp] = (inputEl[currProp] ===
testValue);
  }, this);
  inputEl = null;

  /**
   * Returns a sorted array of all features that were
tested for.
   * @returns {Array.<string>}
   */
  returnVal.getTests = function() {
    // Get all of the properties and methods we've added
to returnVal and
    // sort them.
    var allPropsAndMethods = Object.keys(this).sort();

    // This list will contain all the properties and
methods, but we only want
    // properties, so filter out the methods.
    var allTests = [];
    allPropsAndMethods.forEach(function(currItem) {
      if (typeof this[currItem] != 'function') {
```

```
        allTests.push(currItem);
      }
    }, this);

    return allTests;
  };

  /**
   * Returns an array consisting of all test results. Each
result is an object
   * with the feature property set to the name of the test
and the isSupported
   * property set to true or false, depending on the
support for that feature.
   * @returns {Array.<Object>}
   */
  returnVal.getTestResults = function() {
    var tests = this.getTests();
    var allResults = [];
    tests.forEach(function(currTest) {
      var currResult = {
        feature: currTest,
        isSupported: this[currTest]
      };
      allResults.push(currResult);
    }, this);
    return allResults;
  };

  /**
   * Returns an array of test results for all failed
tests. Each result is an
   * object as described in getResults.
   * @returns {Array.<Object>}
   */
  returnVal.getFailedTestResults = function() {
    var tests = this.getTests();
    var failures = [];
    tests.forEach(function(currTest) {
      if (!this[currTest]) {
        var currResult = {
          feature: currTest,
```

```
          isSupported: this[currTest]
        };
        failures.push(currResult);
      }
    }, this);
    return failures;
  };

  // Return the object with all the results.
  return returnVal;
}

// Test for supported features.
var supportedFeatures = new DetectHTML5Support();

// Fill the table with support information.
var supportTable = document.getElementById('supported');
var allResults = supportedFeatures.getFailedTestResults();
allResults.forEach(function (currTest) {
  var newRow = document.createElement('tr');
  var featureCell = document.createElement('td');
  var supportCell = document.createElement('td');
  featureCell.innerHTML = currTest.feature;
  supportCell.innerHTML = currTest.isSupported;
  newRow.appendChild(featureCell);
  newRow.appendChild(supportCell);
  supportTable.appendChild(newRow);
});

    </script>
  </body>
</html>
```

The script groups together similar tests to make it easier to add or remove tests as fits your needs. In each case you define a set of things to test as an array of simple strings that are the names of the feature to test: the name of the API, the name of the element, or the name of the property. Then each section loops through the arrays and applies the appropriate test and records the result. Note that throughout these tests you are making use of the fact that in JavaScript you can access properties either by dot notation (`Object.property`) or by bracket notation (`Object['property']`), as explained in Chapter 2.

The script demonstrates the detection process by calling the constructor to run the

tests and get a new results object, and then uses the `getFailedTestResults` method to fetch a list of unsupported features and builds a table to show them. (You could easily alter this to use the `getTestResults` method instead to see all the results.) If you run this in different browsers you'll see variations in what isn't supported, especially if you have access to older versions of browsers . . . or Internet Explorer, as shown in Figure 6-1.

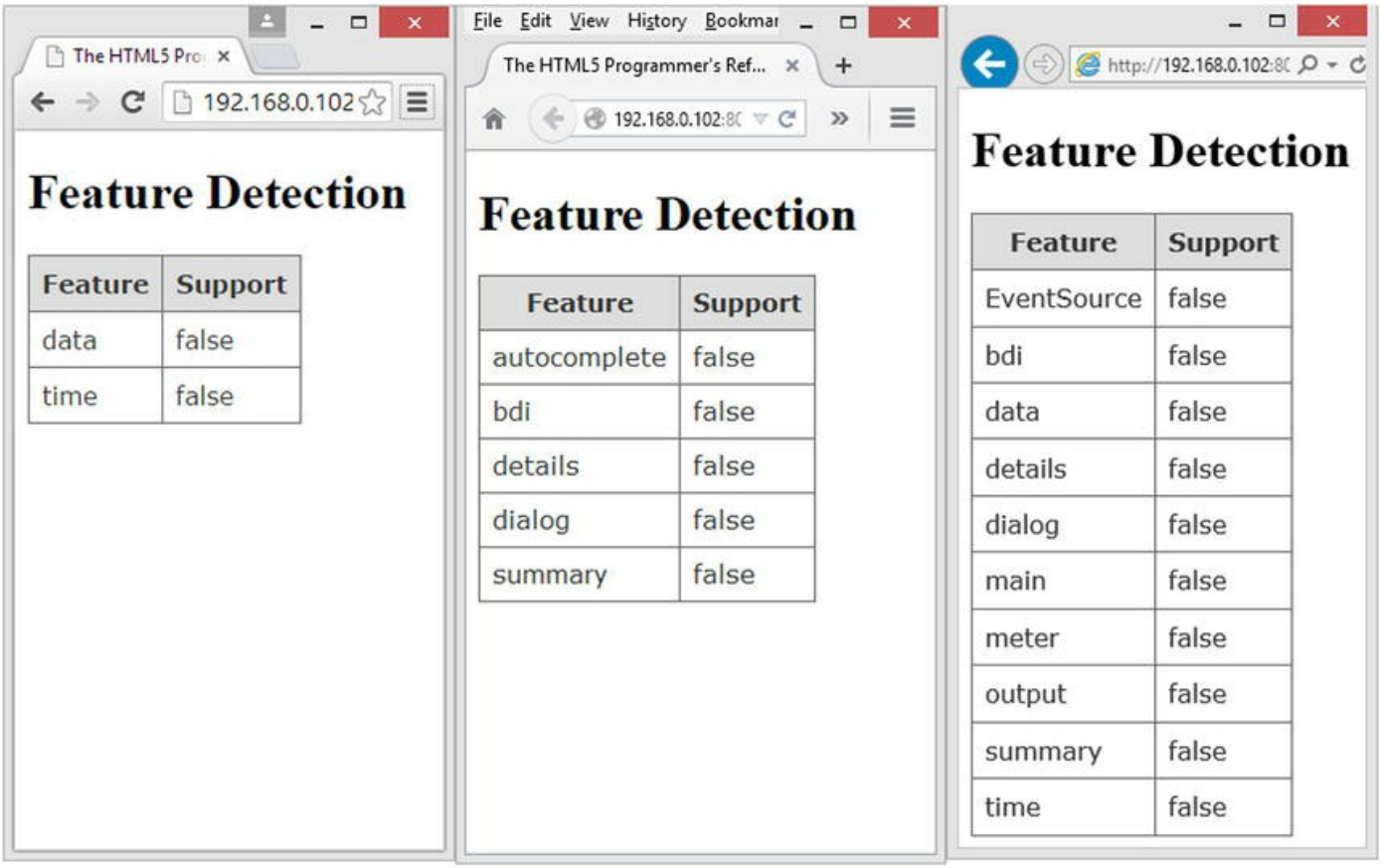


Figure 6-1. *The results of the feature detection script in Chrome, Firefox, and Internet Explorer*

As you can see, support for some features is still missing even in modern browsers. It's particularly sad that Firefox and Internet Explorer do not support the `dialog`, `summary`, or `details` elements; that Firefox doesn't support the `autocomplete` property; and that Internet Explorer doesn't support server-sent events.

Working with Broken or Missing HTML5 Implementations

This brings us neatly to the next question: now that you can detect what HTML5 features are supported, what do you do with that information? You want to use server-sent

events, but Internet Explorer doesn't support them. You want to use autocomplete, but Firefox doesn't know how to do that. Or a significant portion of your users are stuck on older systems so you need to support a broad range of legacy browsers.

The bad news is there is no "one size fits all" solution to the problem of broken or missing implementations. The good news is that many HTML5 features can be mimicked using JavaScript. A script that reproduces a missing feature in this way is called a *shim*.

Consider for example the Web Storage feature (see the "Web Storage" section in Chapter 3). Older browsers will not have the `localStorage` or `sessionStorage` methods available, but you can still store information on the client using HTTP Cookies. With a bit of work you can implement `localStorage` and `sessionStorage` in older browsers by using HTML Cookies as the storage mechanism. A solution like this would enable you to use Web Storage in just about any browser.

Unfortunately, not everything can be completely reproduced with a shim. Features that require access to underlying hardware, such as the Device Orientation API (see the "Device Orientation" section in Chapter 5), which requires access to the host device's accelerometer and gyroscope, can't be reproduced with JavaScript.

Returning to the question posed at the beginning of the section, it's clear that if you know what isn't supported, you can load shim scripts to reproduce those features as needed. To do this, you'll have to dynamically load JavaScript files on demand. The technique for this is fairly simple: just create a script element using `document.createElement` and then set the source attribute to the URL of the desired script. When the script element is appended to the DOM, the browser will load and execute the script. Listing 6-12 demonstrates using this technique along with the feature detection script.

Listing 6-12. Dynamically Loading Shims Based on Feature Support

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
      <script src="../js-lib/detect-support.js"></script>
      <script>
/**
 * Dynamically load a script.
 * @param {string} srcUrl The URL of the script file to
load.
```

```
 */
function loadScript(srcUrl) {
  var newScript = document.createElement(script);
  newScript.src = srcUrl;
  document.querySelector('head').appendChild(newScript);
}

// Test for supported features.
var supportedFeatures = new DetectHTML5Support();

if (!supportedFeatures.localStorage) {
  // The Web Storage is not supported, so load a shim.
  loadScript('../js-lib/webstorage-shim.js');
}
     </script>
    </body>
</html>
```

This example saves the feature detection script as a separate file and loads it on its own. Then it creates a simple function that will dynamically load scripts on demand. Finally, it detects support for the Web Storage feature, and if it is not present it loads a hypothetical shim script that reproduces the Web Storage methods.

Unfortunately, this simple technique doesn't take into account two important issues. First, the technique doesn't allow for an error while the script is loading. What if the script file isn't found? What if the application is a mobile application and the user drops off the network? Your script needs to account for these situations. Fortunately, when a script element encounters an error it publishes an `error` event for which you can register an event handler.

The second issue this technique doesn't account for is that the shim will take time to load. It may only be a few seconds, but you don't want to continue running your script until it has loaded and the methods are available. Otherwise you might access the feature before it has been shimmed, which could result in a serious error in your application. As with error conditions, when a script element loads it publishes an event that you can listen for. Unfortunately, the event type varies depending on the browser. For Chrome, Firefox, Opera, and Safari, the event is a `load` event, and you can register an event handler for it.

For Internet Explorer, however, the event is a `readystatechange` event. When the `readystatechange` event fires, the value of the script element's `readyState` property changes and the new value indicates what stage of loading the script is in:

- `uninitiated`: This is the default state; the script element is

doing nothing.

- `loading`: The script has begun downloading to the browser, but is not yet done.
- `loaded`: The script has completely downloaded to the browser.
- `interactive`: The script has completely downloaded but isn't ready to be used.
- `complete`: The script is ready to be used.

To complicate matters, Internet Explorer doesn't always dispatch `readystatechange` events for each stage of the loading process. You should be most interested in the `loaded` and `complete` states, and Internet Explorer might publish only one of these or both of them, so your `readystatechange` event handler will need to check for both of them, and if one of them occurs the handler will need to do its job and then unregister itself so that it won't be called again if the browser fires another `readystatechange` event.

Listing 6-13 shows a new `loadScript` method that provides a way to register callbacks for both success and error during the loading process.

Listing 6-13. Waiting for a Shim to Load Before Continuing

```
<!DOCTYPE html>
<html>
    <head>
       <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
      <script src="../js-lib/detect-support.js"></script>
      <script>
/**
 * Dynamically loads a script and invokes an optional
callback.
 * @param {string} srcUrl The URL of the script file to
load.
 * @param {function=} onLoadCallback An optional function
to call when the
 *     script is loaded.
 * @param {function=} onErrorCallback An optional function
to call if the script
 *     fails to load.
```

```
 */
function loadScript(srcUrl, onLoadCallback,
onErrorCallback) {

  // Create a script tag.
  var newScript = document.createElement('script');

  // Apply the load callback, if one was provided.
  if (onLoadCallback) {
    if (newScript.readyState) {
      // Internet explorer.
      newScript.onreadystatechange = function() {
        if (newScript.readyState == 'loaded' ||
            newScript.readyState == 'complete') {
          newScript.onreadystatechange = null;
          onLoadCallback.call();
        }
      };
    } else {
      // Every other browser in the universe.
      newScript.onload = onLoadCallback;
    }
  }

  // Apply the error callback, if one was provided.
  if (onErrorCallback) {
    newScript.onerror = onErrorCallback;
  }

  newScript.src = srcUrl;
  document.querySelector('head').appendChild(newScript);
}

// Test for supported features.
var supportedFeatures = new DetectHTML5Support();

if (!supportedFeatures.localStorage) {
  // The Web Storage is not supported, so load a shim.
  loadScript('../js-lib/webstorage-shim.js',
      initApplication,
      handleScriptLoadError);
} else {
  // Web Storage was supported, so continue with the
```

```
application.
  initApplication();
}

/**
 * Handles an error during a script load.
 */
function handleScriptLoadError() {
  console.log('Script failed to load.');
  // Etc.
}

/**
 * Hypothetical function for initializing the application.
 */
function initApplication() {
  console.log('Application continues...');
  // Etc.
}
     </script>
    </body>
</html>
```

The new `loadScript` function now takes both `onLoadCallback` and `onErrorCallback` parameters. Call the `onLoadCallback` function when the script is done loading, and the `onErrorCallback` function if the script loading process fails. Both of these parameters are optional, but most likely you will need them. The script checks for Web Storage support as before, and if present it simply continues. If not, it loads the shim and then continues when the load is complete.

This is great if you only need one HTML5 feature, but your project will probably need to verify support for multiple features. To make this easier, you can create a simple registry that contains the names of all of the features you need, and the paths to shims that can be loaded if they're not supported:

```
Object featureRegistryEntry {
  string 'featureName',
  string 'shim'
}

Array featureRegistry[featureRegistryEntry]
```

This registry then becomes a single place in your code to manage all of the features you need, making it easier to add or remove features as your application changes and

grows.

However, this means you will be checking multiple features, and could be loading multiple shims. Each shim could take a different amount of time to load, and you wouldn't want your script to continue until all of the shims are done loading. And to complicate matters, one of the scripts might fail to load for some reason. To keep track of what is loading and what has succeeded and failed, you will need to build a loading queue. This queue can be a simple data structure that has a simple success condition boolean (set to `true` by default, but as soon as a script fails to load you will set it to `false`) and an array that consists of entries for each script currently loading:

```
Object loadQueue {
  boolean 'noErrorsOccurred',
  Array.<boolean> 'queue'
}
```

Each time a script starts loading, it adds an entry to `loadQueue.queue`. The actual entry itself doesn't matter, because we only care when all scripts are done loading, not when a particular script is done or the order in which they complete. In this case an entry into the queue will be a simple `true` value. When a script is done loading, you will remove an entry from the queue. If the queue is empty at this point, you know all scripts are done.

When an individual script element load process finishes, it will invoke either the success callback or the failed callback. In the case of a failed load, set the `loadQueue.noErrorsOccurred` value to `false`. That way, when the queue is empty, you'll know which final callback to invoke.

Tip This simple load queue data structure could be reworked as a formal class with methods for adding and removing items as well as setting the error state, similar to what you did with the `DrawCycle` class in the "Animation Timing" section of Chapter 3.

Listing 6-14 demonstrates these techniques.

Listing 6-14. Loading Multiple Shims and Tracking Process Using a Queue

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
    </head>
```

```
    <body>
      <script src="../js-lib/detect-support.js"></script>
      <script>
// Create a registry of HTML features that we need and
shims to apply if they
// are not present. The registry will be an array of
objects; each object will
// consist of a feature name and a path to a shim to apply
if that feature is
// not supported.
var featureRegistry = [
  {
    'featureName' : 'localStorage',
    'shim' : '../js-lib/webstorage-shim.js'
  },
  {
    'featureName' : 'requestAnimationFrame',
    'shim' : '../js-lib/animationframe-shim.js'
  }
];

/**
 * Dynamically loads a script and invokes an optional
callback.
 * @param {string} srcUrl The URL of the script file to
load.
 * @param {function=} onLoadCallback An optional function
to call when the
 *     script is loaded.
 * @param {function=} onErrorCallback An optional function
to call if the script
 *     fails to load.
 */
function loadScript(srcUrl, onLoadCallback,
onErrorCallback) {

  // Create a script tag.
  var newScript = document.createElement('script');

  // Apply the load callback, if one was provided.
  if (onLoadCallback) {
    if (newScript.readyState) {
```

```
      // Internet explorer.
      newScript.onreadystatechange = function() {
        if (newScript.readyState == 'loaded' ||
            newScript.readyState == 'complete') {
          newScript.onreadystatechange = null;
          onLoadCallback.call();
        }
      };
    } else {
      // Every other browser in the universe.
      newScript.onload = onLoadCallback;
    }
  }

  // Apply the error callback, if one was provided.
  if (onErrorCallback) {
    newScript.onerror = onErrorCallback;
  }

  newScript.src = srcUrl;
  document.querySelector('head').appendChild(newScript);
}

/**
 * Verifies all features in the registry and applies shims
as needed.
 * @param {function=} onLoadCallback An optional callback
function. If no shims
 *     were loaded this callback will be invoked
immediately, otherwise it will
 *     be invoked after all shims have successfully
loaded.
 * @param {function=} onErrorCallback An optional callback
function which will
 *     be invoked if even one of the shim scripts fails to
load.
 */
function verifyAllFeatures(onLoadCallback,
onErrorCallback) {

  // Create loading queue. This queue consists of an error
condition boolean
  // and a simple array of entries.
```

```
  window.loadQueue = {
    'noErrorsOccurred' : true,
    'queue' : []
  };

  // Flag for all feature support.
  var allFeaturesSupported = true;

  featureRegistry.forEach(function(currFeature) {
    if (!supportedFeatures[currFeature.featureName]) {
      // A feature is not supported.
      allFeaturesSupported = false;

      // Add an entry to the loading queue.
      window.loadQueue.queue.push(true);

      /**
       * Callback function that is invoked when the shim
script is loaded.
       * Removes an entry from the loading queue and if the
queue is then empty
       * invokes one of the callbacks. If the queue is in
an error condition,
       * the error callback is invoked. Otherwise, the load
callback is invoked.
       */
      var handleThisLoad = function() {
        // Remove entry from the loading queue.
        window.loadQueue.pop();

        // If the queue is empty, all scripts are loaded
and we can invoke the
        // callback.
        if (window.loadQueue.queue.length === 0) {
          // Check for error condition.
          if (window.loadQueue.noErrorsOccurred) {
            // Everything loaded, so call the load callback,
if one was
            // provided.
            if (onLoadCallback) {
              onLoadCallback.call();
            }
          } else {
```

```
          // At least one of the scripts failed to load,
so call the error
          // callback, if one was provided.
          if (onErrorCallback) {
            onErrorCallback.call();
          }
        }
      }
    };

    /**
     * Callback function that is invoked when the shim
script fails to load.
     * Places the load queue in an error state and
removes an entry. If the
     * queue is then empty it invokes the final error
callback.
     */
    var handleThisError = function() {
      // Immediately put the load queue into an error
condition.
      window.loadQueue.noErrorsOccurred = false;

      // Remove entry from the loading queue.
      window.loadQueue.pop();

      // If the queue is empty, we need to invoke the
error callback, if one
      // was provided.
      if (window.loadQueue.queue.length === 0 &&
onErrorCallback) {
        onErrorCallback.call();
      }
    };

    // Call the loadScript function with our custom
handlers.
    loadScript(currFeature.shim, handleThisLoad,
handleThisError);
  }
 }, this);
 if (allFeaturesSupported && onLoadCallback) {
  onLoadCallback.call();
```

```
  }
}

// Test for supported features.
var supportedFeatures = new DetectHTML5Support();
verifyAllFeatures(initApplication, handleScriptLoadError);

/**
 * Handles an error during a script load.
 */
function handleScriptLoadError() {
  console.log('A script failed to load.');
  // Etc.
}

/**
 * Hypothetical function for initializing the application.
 */
function initApplication() {
  console.log('Application continues...');
  // Etc.
}
    </script>
  </body>
</html>
```

In this example, you start by creating the registry of tests to check and shims to load. You've also added a `verifyAllFeatures` function that performs the following operations:

1. It checks each feature for support.
2. For each unsupported feature it does the following:
    a. It adds an entry to the loading queue.
    b. It creates a load callback function for this script that removes an entry from the loading queue and then invokes the final callback if the queue is then empty. If the queue is in an error state, the final callback that is invoked is the error callback, otherwise it is the load callback.
    c. It creates an error callback function for this script that immediately sets the queue in an error condition. It then

removes an entry from the queue and, if the queue is then empty, it invokes the final error callback.

   d. Finally, it calls the `loadScript` function with the newly created callbacks.

3. If all features were supported, it calls the `initApplication` function to continue the application.

When run, this script will verify all of the features specified in the registry and invoke the appropriate callbacks depending on the results.

This technique has a major disadvantage: for each required shim, it will generate a separate script tag and thus HTTP request. If you have several shims loading, this alone can cause a noticeable delay in your application. If any of the shims are resource intensive, that will slow it further. To complicate this problem, you will generally only need shims for older browsers, which usually will only be running on older hardware with older operating systems, so they will already be resource constrained. This is particularly a problem for mobile applications, where system resources are quite limited.

If you need the shims, there's not much you can do about it and you should load them so your application can work. One way to help mitigate the problem is to make sure you provide feedback in your user interface that loading is happening, so users know that the application hasn't just frozen. Another way to mitigate the expense of loading shims is to spread it out. In all of these examples so far you've been testing all features at once. However, one of the benefits of using a dynamic technique like this is you can test for features as you need them. After all, if the user never goes into the part of the application that needs a particular feature, there's no need to load the shim for that feature. This is especially important if the shim is a large file or otherwise resource intensive.

You can easily add a new function that checks a single feature in the registry, as shown in Listing 6-15.

Listing 6-15. A Function for Checking a Single Feature

```
/**
 * Checks a single feature and applies a shim if needed.
 * @param {string} featureName The name of the feature to
check.
 * @param {function=} onLoadCallback An optional function
to call when the shim
 *     is loaded.
```

```
 * @param {function=} onErrorCallback An optional function
to call if the shim
 *     fails to load.
 * @return {boolean} True if the feature was supported
natively, or false if
 *     the feature was not supported and a shim was
applied.
 */
function verifyFeature(featureName, onLoadCallback,
onErrorCallback) {
  var returnVal = true;
  featureRegistry.forEach(function(currFeature) {
    if ((currFeature.featureName === featureName) &&
        !supportedFeatures[currFeature.featureName]) {
      loadScript(currFeature.shim, onLoadCallback,
onErrorCallback);
      returnVal = false;
    }
  });
  return returnVal;
}

// Test for supported features.
var supportedFeatures = new DetectHTML5Support();

// Verify the Animation Timing feature.
if (verifyAllFeatures('requestAnimationFrame',
initApplication,
    handleScriptLoadError)) {
  initApplication();
}

/**
 * Handles an error during a script load.
 */
function handleScriptLoadError() {
  console.log('A script failed to load.');
  // Etc.
}

/**
 * Hypothetical function for initializing the application.
 */
```

```
function initApplication() {
  console.log('Application continues...');
  // Etc.
}
```

This function runs through the `featureRegistry` until it finds the entry corresponding to the desired feature. It then checks for support and applies a shim if needed. For this function, if no shim was needed it returns `true` rather than invoking the callback itself. This gives more flexibility in how the callback is invoked, allowing you to call different functions depending on feature support.

Online Resources for Browser Support, Feature Detection, and Shims

Now that you can detect features and load shims as needed, you need to have shims to load. Building shims for HTML5 features ranges from the simple (structural element shims) to the moderate (implementing the Web Storage feature on top of HTTP Cookies) to the forbiddingly complex (implementing Web Sockets). The good news is that shims have already been written for most HTML5 features.

Can I Use

The Can I Use database, located at `www.caniuse.com`, is probably the most important resource for researching browser support and shims for not only HTML5 features, but also CSS3 and advanced JavaScript features as well. The site has up-to-date tables of browser support indicating the level of support of the feature in question, including how far back the browser supported it. Also included are global support percentages, various ways of visualizing support data, links to relevant specifications, articles, shims, and a brilliant custom-coded test suite.

Modernizr

Modernizr, located at `www.modernizr.com`, is a suite of feature detection scripts that detect support for HTML5 and CSS3 features. Modernizr also implements dynamic loading of shims using YepNope (`www.yepnopejs.com`); however, they recommend dynamically combining required shims on the server into a single file, thus saving HTTP responses (which are very expensive in terms of application efficiency). Modernizr also has a page devoted to shims for HTML5, CSS5, and JavaScript

functions on their wiki, located at https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-Browser-Polyfills.

HTML5 Rocks

The HTML5 Rocks website, located at www.html5rocks.com, is a great resource for articles on HTML5, CSS3, and JavaScript features. Articles include tutorials, best practices, shims, and more.

Webshim

The Webshim library at http://afarkas.github.io/webshim/demos is a polyfill library that enables several HTML5 features on older browsers.

Example Project: MobiDex, a Mobile Dexterity Puzzle

Dexterity puzzles are among the oldest puzzles in history. The simplest version of a dexterity puzzle consists of a small board held flat in the palm of the hand, with a ball bearing or marble on top. The goal of the puzzle is to guide the ball from one location to another without letting it fall through holes in the board. Other dexterity puzzles include mazes and games with multiple balls where the goal is to get the balls to rest in specific places, either to score points or to complete a picture.

Using HTML5 technologies, specifically the Device Motion and Animation Timing APIs, you can easily implement a version of a dexterity puzzle on a mobile device. Your dexterity puzzle, called MobiDex (for Mobile Dexterity), will consist of a square playing field. Within the playing field you will draw a "ball" that will be animated as the user tilts their device. You will also draw a set of targets in the playing field as well as a set of obstacles. The player will have a limited number of chances to collect all of the targets while avoiding the obstacles. Every time the player runs into an obstacle, it will cost them one of their chances. If they can collect all of the targets before running out of "lives" they win the game.

A game like this sounds pretty simple on the surface, but it has significant complexity once you think about it. One of the biggest difficulties in a project like this is

successfully capturing all of the requirements so you have some idea of the tasks ahead of you. Since this project is a game it makes sense to approach defining the project from a user's point of view. A great tool for defining a user-centered project from the user's point of view is a technique called "user stories." A user story is a simple statement that encapsulates a single feature as described from the user's point of view. User stories are similar to use cases, but are smaller and more compact, and typically define a single feature as opposed to a workflow. The typical pattern of a user story goes like this:

As a <type of user>, I want <functionality> so that <desired goal>.

The pattern isn't set in stone and can be modified to be useful for the given project. For example, in your project there's only one type of user (the player), so there's really no need to keep repeating it in each user story. And occasionally the functionality specified is its own goal, so there's no need for the desired goal clause.

Tip User stories are a technique often employed in Agile software development.

A set of user stories for your game would be as follows:

- As a player, I want there to be a well-defined playing field for the game, so I know what the boundaries are.
- I want the game to have a "ball" that responds to the tilting motion of my device, so that I can play the game.
- I expect the "ball" to never leave the confines of the playing field.
- I want the game to have a set of targets on the field for me to pick up so that I can win the game.
- I want the targets to disappear from the playing field when I touch them with the "ball" so that I know how many targets are left.
- I want the game to have a set of obstacles on the field so that the game is challenging.
- I want the targets and obstacles to not be on top of one another, so that I can successfully gather all of the targets without needing to hit an obstacle.
- I want to have a limited number of chances to collect all of the targets while avoiding the obstacles, so that the game is more challenging.

- I expect to lose one of my chances each time my "ball" collides with an obstacle.
- I want the game to make it clear when I have collided with an obstacle.
- I expect to be able to see how many chances I have left to win the current game.
- I expect the game to tell me if I have won or lost.
- I expect to be able to restart the game when it is over, so that I can play again.

These user stories define a set of features to build: targets, obstacles, a ball, an indicator for the number of chances remaining, the interactions the player expects, and so on.

You'll need the following features:

- A UI capable of displaying a playing field, a ball, obstacles, targets, and the like.
- A method for generating random sets of targets and obstacles. The only difference between a target and an obstacle is the color, so you should be able to write one set of code to generate both.
- A method for moving the ball around on the screen. This will have to use the Device Orientation API, similar to what we did in Listing 5-7 in Chapter 5. In your case, though, we don't want the `deviceorientation` event to drive the redrawing of the screen; we want to use `requestAnimationFrame` from the Animation Timing API.
- A method for determining collisions between the ball and both targets and obstacles. There's a couple of ways you could do this; the easiest is to keep track of the coordinates of each target and obstacle and compare them to the ball's coordinates as the game progresses.
- A unified way of indicating coordinates—you can create a simple class for this and then instantiate it as needed.
- A way of initializing and resetting the game so that it can be replayed.

The Playing Field UI

To start, build the playing field user interface. The first decision you'll have to make is the technology to use to implement game's UI. You could use Canvas, but you don't really need to. Your game has a fairly simple interface; all you need to do is display a playing field and some items on it. You can easily implement what you need using HTML and CSS. This also has the virtue of being quite fast, which is good because this game targets mobile devices.

Start by drawing a simple playing field on the screen. Within that field you'll lay out the ball, obstacles, targets, and the game over message. Above the field you'll display the number of remaining lives. You'll also add a title above the playing field. The base markup is shown in Listing 6-16.

Listing 6-16. Markup for MobiDex Playing Field

```
<h1>MobiDex</h1>
<div id="remaining-balls"></div>
<div id="container-field">
  <div id="game-field"></div>
  <div id="ball"></div>
  <div id="gameover" class="hidden"></div>
</div>
```

As you can see, there's a container for the lives, a ball, the game over message (which can be used both for winning or losing), and a field where the targets and obstacles will be drawn.

Keep the styling simple for the sake of efficiency. Start with a field that is 200 pixels wide by 200 pixels high. Draw a simple 1-pixel border around the field (so, according to the CSS box model, the dimensions for the field will actually need to be 202 pixels wide by 202 pixels high). The field will be absolutely positioned on the screen, which allows you to absolutely position elements inside of it relative to its coordinate origin.

The "ball" and the targets and obstacles will be div elements. They will all be 10 pixels wide by 10 pixels high for uniformity. The ball will be blue, the targets will be green, and the obstacles will be red. Make all of them look round by giving them a `border-radius` of 50%.

Each time the player runs into an obstacle it will cost them a ball. Their remaining number of balls will be shown in the div above the playing field.

One of the user stories specified the requirement that the game distinctly let the

player know when they have hit an obstacle. Of course you'll remove a ball from the display of remaining balls, but the player will have their eyes on the playing field and might not notice that. An easy way to let the player know when they've collided with an obstacle is to change the background color of the playing field while the collision is in progress. That's easy enough; all it requires is adding a CSS class to the container.

Finally, you'll provide some styles to the game over message, and have different background colors for it: one for winning and one for losing.

Listing 6-17 shows the resulting CSS for the entire game.

Listing 6-17. CSS for MobiDex Playing Field

```
body {
  font-family: arial, helvetica, sans-serif;
}

h1 {
  text-align: center;
}

#remaining-balls {
  height: 10px;
  left: 50px;
  position: absolute;
  top: 109px;
}

#container-field {
  border: 1px solid red;
  height: 202px;
  left: 50px;
  position: absolute;
  top: 120px;
  width: 202px;
}

#game-field {
  height: 202px;
  left: 0px;
  position: absolute;
  top: 0px;
  width: 202px;
}
```

```
#ball,
.life {
  background-color: blue;
  height: 10px;
  left: 0px;
  position: absolute;
  top: 0px;
  width: 10px;
}

#gameover {
  border-radius: 5px;
  font-size: 2em;
  font-weight: bold;
  margin-left: 10px;
  margin-right: 10px;
  margin-top: 80px;
  padding: 5px 0;
  position: relative;
  text-align: center;
}

.obstacle,
.target {
  height: 10px;
  position: absolute;
  width: 10px;
}

.obstacle {
  background-color: red;
}

.target {
  background-color: green;
}

#ball,
.life,
.obstacle,
.target {
  border-radius: 50%;
```

```
}

.hidden {
  display: none;
}

.winner {
  background-color: rgb(116, 216, 94);
}

.loser {
  background-color: rgb(255, 85, 85);
}

.collision {
  background-color: rgba(215, 44, 44, 0.4);
}
```

When you render the game field, including sample targets and obstacles, the result will look like Figure 6-2.

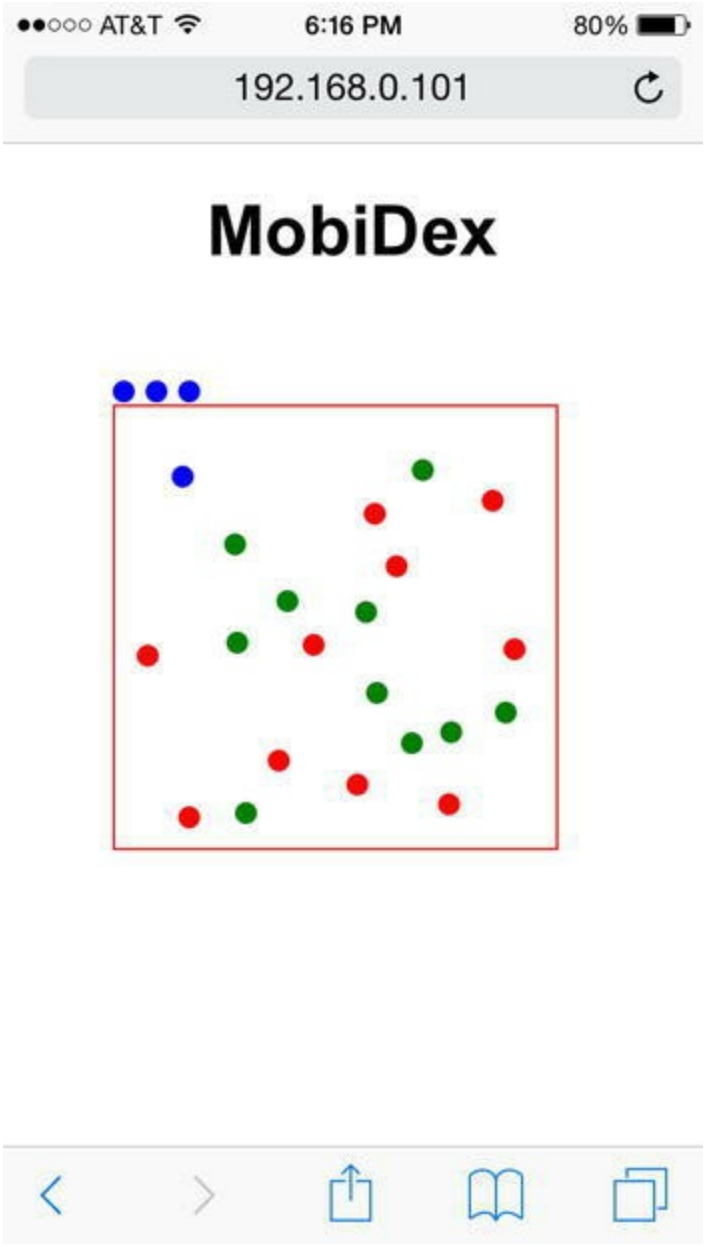


Figure 6-2. *A sample rendering of the MobiDex game*

Generating Obstacles and Targets

Probably the easiest place to start is to generate the obstacles and targets. From a high level, you need to randomly generate a set of coordinates for each obstacle and target. That's not hard to do, but if you just generate random numbers chances are pretty good that some of your obstacles and targets will be very close to one another, if not right on top of one another. That contravenes one of your user stories, which specifies that targets and obstacles should not be on top of one another. So you'll need a way of detecting when a newly generated obstacle or target is colliding with an existing one. This is the same functionality you'll need to determine if the ball is colliding with an

obstacle or target, so this code path can serve both purposes.

Since you're dealing with coordinates, create a simple `Coordinate` class that you can use throughout this process, as shown in Listing 6-18.

Listing 6-18. A Simple `Coordinate` Class

```
/**
 * Coordinate class.
 * @param {number} xOrd The x ordinate of the coordinate.
 * @param {number} yOrd The y ordinate of the coordinate.
 * @constructor
 */
function Coordinate(xOrd, yOrd) {
  this.x = xOrd;
  this.y = yOrd;
}
```

This simple class records the x and y values (corresponding to the CSS properties `left` and `top`, respectively). You could just save a reference to the element itself and get the left and top CSS properties when you need to, but those properties are actually strings with units on them (e.g., `"5px"`) so you would need to recast them as numbers in order to compare them with one another as required for collision detection. And since you're generating these numbers ourselves, you can store them and have them handy whenever you need them, no parsing required.

You'll need a way to generate random numbers between upper and lower boundaries. Initially you might think any value between 0 and 200 would be viable since the playing field is 200 × 200. However that doesn't take into account the width and height of the element; if you were to place a 10 × 10 div at `(200, 200)` it would be outside the playing field. To avoid accidentally playing a target or obstacle outside of the playing field, limit your random numbers to integers between 10 and 190. Listing 6-19 shows a utility function for this.

Listing 6-19. A Utility Function for Generating Random Integers Between Two Bounds

```
/**
 * Returns a random integer between the specified minimum
and maximum values.
 * @param {number} min The lower boundary for the random
number.
 * @param {number} max The upper boundary for the random
```

```
number.
 * @return {number}
 */
getRandomIntegerBetween_ = function(min, max) {
  return Math.floor(Math.random() * (max - min + 1))
+ min;
};
```

To generate a new `Coordinate`, do something like the following:

```
var target = new Coordinate(getRandomIntegerBetween(10,
190), getRandomIntegerBetween(10, 190));
```

You can store the `Coordinates` in arrays, one for targets and one for obstacles. That way you can compare new `Coordinates` to determine if they're too close to existing ones.

Comparing `Coordinates`

To detect collisions, you'll need a `Coordinate` to check and an array of `Coordinates` to check it against. If the `Coordinate` is too close to any of the Coordinates in the array, you can return a collision.

What determines "too close," though? There's a few ways you could determine this, but for the purposes of your simple game you can just check to see if a given ordinate (x or y) of the target `Coordinate` is within a defined range of the `Coordinate` you're checking against. The range is defined by a sensitivity value. So given a target `Coordinate` (x.t, y.t) and an original `Coordinate` (x.o, y.o) and a sensitivity s:

```
(x.o - s) < x.t < (x.o + s)
(y.o - s) < y.t < (y.o + s)
```

If both inequalities hold true, the target `Coordinate` is close enough to the original `Coordinate` that their associated elements are visually colliding. This is an approximation, of course, but for such small elements it should work.

Listing 6-20 shows a function that performs this check.

Listing 6-20. Checking for Collisions Between a Target Coordinate and an Array of Existing Coordinates

```
/**
 * Check to see if the specified coordinates collide with
```

```
an existing set of
 * coordinates.
 * @param {Coordinate} coordinate The coordinate to check.
 * @param {number} sensitivity The sensitivity for
a collision. If coordinates
 *     are within sensitivity distance of a target
coordinate, a collision
 *     will be registered.
 * @param {Array.<Coordinate>} arrTargetCoords An array of
target coordinates
 *     to check against.
 * @return {number} The index of the member of the target
coordinates array
 *     that is being hit, or -1 if no collision is
detected.
 */
checkCollision_ = function(coordinate, sensitivity,
arrTargetCoords) {
  // Loop through each target coordinate and compare the
provided values.
  for (var i = 0; i < arrTargetCoords.length; i++) {
    var currObstacle = arrTargetCoords[i];
    var xcoll = false;
    var ycoll = false;

    // If the provided x coordinate is within range of the
obstacle coordinate,
    // then there is an x collision.
    if (((currObstacle.x - sensitivity) < coordinate.x) &&
        (coordinate.x < (currObstacle.x + sensitivity))) {
      xcoll = true;
    }

    // If the provided y coordinate is within range of the
obstacle coordinate,
    // Then there is a y collision.
    if (((currObstacle.y - sensitivity) < coordinate.y) &&
        (coordinate.y < (currObstacle.y + sensitivity))) {
      ycoll = true;
    }

    // If there is both an x and a y collision, then
```

```
return true.
    if (xcoll && ycoll) {
      return i;
    }
  }
  return -1;
};
```

This function takes a `Coordinate`, a sensitivity value, and an array of `Coordinates` to check against. It then loops through each `Coordinate` in the array and determines if a collision is occurring.

Now that you have a way of checking collisions, you can build a function for generating an array of `Coordinates`. Your game will need two arrays of `Coordinates`, one for targets and one for obstacles. And you'll need to check both arrays when checking for collisions while generating new `Coordinates` for either array.

At this point you should start thinking about how you want to encapsulate these functions and the data you're going to be generating. It's easy to create a simple class constructor that contains the functions you've built so far, and adds in the arrays for targets and obstacles, as shown in Listing 6-21.

Listing 6-21. The Beginnings of the `MobiDex` Class

```
/**
 * Creates a new game. Assumes that the required DOM
elements are present.
 * @constructor
 */
function MobiDex() {

  /**
   * Array of obstacle coordinates.
   * @type {Array.<Coordinate>}
   * @private
   */
  this.arrObstacles_ = [];

  /**
   * Array of target coordinates.
   * @type {Array.<Coordinate>}
   * @private
```

```
   */
  this.arrTargets_ = [];

  /**
   * The number of obstacles to draw on the game field.
   * @type {number}
   * @private
   */
  this.numberOfObstacles_ = 10;

  /**
   * The number of targets to draw on the game field.
   * @type {number}
   * @private
   */
  this.numberOfTargets_ = 10;

  /**
   * Checks to see if the specified coordinates collide
with an existing set of
   * coordinates.
   * @param {Coordinate} coordinate The coordinate to
check.
   * @param {number} sensitivity The sensitivity for
a collision. If coordinates
   *     are within sensitivity distance of a target
coordinate, a collision
   *     will be registered.
   * @param {Array.<Coordinate>} arrTargetCoords An array
of target coordinates
   *     to check against.
   * @return {number} The index of the member of the
target coordinates array
   *     that is being hit, or -1 if no collision is
detected.
   * @private
   */
  this.checkCollision_ = function(coordinate, sensitivity,
arrTargetCoords) {
    // Loop through each target coordinate and compare the
provided values.
    for (var i = 0; i < arrTargetCoords.length; i++) {
```

```
      var currObstacle = arrTargetCoords[i];
      var xcoll = false;
      var ycoll = false;

      // If the provided x coordinate is within range of
the obstacle coordinate,
      // then there is an x collision.
      if (((currObstacle.x - sensitivity) < coordinate.x)
&&
          (coordinate.x < (currObstacle.x + sensitivity)))
{
        xcoll = true;
      }

      // If the provided y coordinate is within range of
the obstacle coordinate,
      // Then there is a y collision.
      if (((currObstacle.y - sensitivity) < coordinate.y)
&&
          (coordinate.y < (currObstacle.y + sensitivity)))
{
        ycoll = true;
      }

      // If there is both an x and a y collision, then
return true.
      if (xcoll && ycoll) {
        return i;
      }
    }
    return -1;
  };

  /**
   * Generates a set of random coordinates and adds them
to the provided array.
   * Tries to avoid duplicating too closely any
previously-generated
   *     coordinates.
   * @param {number} numberOfCoords The number of
coordinates to generate.
   * @param {Array} targetArray The array to fill with the
new coordinates.
```

```
   * @private
   */
  this.generateCoords_ = function(numberOfCoords,
targetArray) {
    for (var i = 0; i < numberOfCoords; i++) {
      var newCoord = new
Coordinate(this.getRandomIntegerBetween_(10, 190),
          this.getRandomIntegerBetween_(10, 190));
      while (this.checkCollision_(newCoord, 15,
           this.arrObstacles_.concat(this.arrTargets_)) > -
1) {
        newCoord.x = this.getRandomIntegerBetween_(10,
190);
        newCoord.y = this.getRandomIntegerBetween_(10,
190);
      }
      targetArray.push(newCoord);
    }
  };

  /**
   * Returns a random integer between the specified
minimum and maximum values.
   * @param {number} min The lower boundary for the random
number.
   * @param {number} max The upper boundary for the random
number.
   * @return {number}
   * @private
   */
  this.getRandomIntegerBetween_ = function(min, max) {
    return Math.floor(Math.random() * (max - min + 1))
+ min;
  };
};

/**
 * Coordinate class.
 * @param {number} xOrd The x ordinate of the coordinate.
 * @param {number} yOrd The y ordinate of the coordinate.
 * @param {Element} element A reference to the DOM element
for these
```

```
 *     coordinates.
 * @constructor
 */
function Coordinate(xOrd, yOrd, element) {
  this.x = xOrd;
  this.y = yOrd;
  this.element = element;
}
```

The new `MobiDex` class has your `getRandomIntegerBetween` and `checkCollision` methods, as well as arrays for the obstacles and targets. It also has constants for the number of obstacles and targets that should be generated.

The class also has a new method: `generateCoords`. This method takes two parameters: a number (for the number of `Coordinates` to generate) and a target array to fill with the `Coordinates` it generates. The method automatically checks both the existing target and obstacle arrays for collisions using the `checkCollisions` method. If a collision is detected, a new `Coordinate` is generated and checked for collision. The process continues until the new `Coordinate` is not colliding with any existing `Coordinates`.

This is enough functionality to actually generate the targets and obstacles and draw them in the interface. To do that you'll add a new method to the class, `drawGameField`, as shown in Listing 6-22.

Listing 6-22. The `drawGameField` Method and Associated Properties

```
/**
 * Reference to the 'gamefield' DOM element.
 * @type {Element}
 * @private
 */
this.gameField_ = document.getElementById('game-field');

/**
 * Initializes the obstacles and targets and draws the UI.
 * @private
 */
this.drawGameField_ = function() {
  // Clear the game field.
  this.gameField_.innerHTML = '';

  // Fill up the obstacle and target arrays with random
```

```
coordinates.
  this.generateCoords_(this.numberOfObstacles_,
this.arrObstacles_);
  this.generateCoords_(this.numberOfTargets_,
this.arrTargets_);

  // Create a div that can be used as a template for
cloning.
  var templateDiv = document.createElement('div');

  // Add the obstacles to the playing field.
  this.arrObstacles_.forEach(function(currCoord) {
    var newObstacle = templateDiv.cloneNode();
    newObstacle.classList.add('obstacle');
    newObstacle.style.left = currCoord.x + 'px';
    newObstacle.style.top = currCoord.y + 'px';
    this.gameField_.appendChild(newObstacle);
  }, this);

  // Add the targets to the playing field.
  this.arrTargets_.forEach(function(currCoord) {
    var newTarget = templateDiv.cloneNode();
    newTarget.classList.add('target');
    newTarget.style.left = currCoord.x + 'px';
    newTarget.style.top = currCoord.y + 'px';
    this.gameField_.appendChild(newTarget);
  }, this);
};
```

Here you have added two new items to the class. The first is a reference to the game field DOM element, since you'll be using that throughout the game. (Because this class is going to get fairly large, this example, and future examples, will only show what you're adding to the class, rather than repeating everything each time. At the end of the chapter I'll provide a full, organized listing for study; you can also download the examples.)

Tip Your DOM structure is quite simple. Fetching the element reference every time you needed it wouldn't necessarily be a performance problem, but the reference isn't something that will change, so you might as well fetch the reference and store it for later use. You'll use this technique throughout the class.

The second is the `drawGameField` method, which clears the game field, generates the arrays of target and obstacle `Coordinates`, and then draws them on the game field. You've used the technique of creating a template element and cloning it for each new `Coordinate`.

Tip If you are in a situation where you are creating the same kind of element over and over again, it's often faster to make a template element like this and clone it rather than create each element anew. See http://jsperf.com/clonenode-vs-createelement-performance/32 for various tests and results.

There's one other thing you need to do in the `drawGameField` method, and that's draw the initial number of balls left. To do that you'll create a generic function that you can call at any time to update that part of the UI, since you'll need it when collisions with obstacles occur (see Listing 6-23).

Listing 6-23. The `updateRemainingBalls` Method

```
/**
 * Reference to the 'remaining-balls' DOM element.
 * @type {Element}
 * @private
 */
this.remainingBalls_ = document.getElementById('remaining-
balls');

/**
 * The number of balls remaining.
 * @type {number}
 * @private
 */
this.balls_ = 3;

/**
 * Updates the number of remaining balls displayed.
 * @private
 */
this.updateRemainingBalls_ = function() {
  // Clear the current lives.
  this.remainingBalls_.innerHTML = '';
  // Create a template that we can clone and use multiple
```

```
times.
  var lifeTemplate = document.createElement('div');
  lifeTemplate.classList.add('life');
  // Add an element for each life.
  for (var i = 0; i < this.balls_; i++) {
    var currLife = lifeTemplate.cloneNode();
    currLife.style.left = (i * 15) + 'px';
    this.remainingBalls_.appendChild(currLife);
  }
};
```

Here you've added two new constants: a reference to the `remaining-balls` DOM element (again cached for future use) and the total number of balls each game starts with. You can add a call to this method at the end of the `drawGameField` method.

The `deviceorientation` Event Handler

You will need to use the `deviceorientation` event for the Device Motion API to actually detect changes in the orientation of the mobile device and use it to calculate where to move the ball. The `deviceorientation` event fires continuously, so for the sake of efficiency the event handler for it should be as lean as possible. For example, you don't want to do any DOM manipulation within the event handler (that should be done in the draw cycle, which I'll discuss in the next section).

The event handler should do two things:

- it should determine the new coordinates of the ball and store that information in the class, and
- it should check for collisions at the new coordinates and store that information in the class.

Then the draw cycle can actually update the position of the ball and react to any collisions that have occurred. This means the event handler is only doing some arithmetic and storing the results, which is pretty efficient.

I will borrow a little code from Example 5-7 in Chapter 5, for calculating a new `Coordinate` based on the Euler Angles published by the `deviceorientation` event. Listing 6-24 shows the new methods and their associated properties.

Listing 6-24. The `deviceorientation` Event Handler

```
/**
 * The current coordinate of the 'ball'.
 * @type {Coordinate}
 * @private
 */
this.currCoordinate_ = new Coordinate(0, 0);

/**
 * The index of the obstacle that the ball is currently
colliding with.
 * @type {number}
 * @private
 */
this.currentObstacleIndex_ = -1;

/**
 * The index of the target that the ball is currently
colliding with.
 * @type {number}
 * @private
 */
this.currentTargetIndex_ = -1;

/**
 * Gets an ordinate based on a Euler Angle.
 * @param {number} angle The orientation angle that is
inducing the change.
 * @param {number} ord The previous value of the ordinate.
 * @return {number} The new value of the ordinate.
 * @private
 */
this.getOrd_ = function(angle, ord) {
  var delta = Math.round(angle - (angle * 0.3));
  var tempVal = ord + delta;
  if (tempVal > 0) {
    ord = Math.min(192, tempVal);
  } else {
    ord = 0;
  }
  return ord;
};

/**
```

```
 * Handles a deviceorientation event from the window.
 * @param {DeviceOrientationEvent} event The device
orientation event object.
 * @private
 */
this.handleDeviceOrientation_ = function(event) {
  // Get the x and y positions and update the current
coordiate with them.
  this.currCoordinate_.x = this.getOrd_(event.gamma,
this.currCoordinate_.x);
  this.currCoordinate_.y = this.getOrd_(event.beta,
this.currCoordinate_.y);

  // Check for collisions.
  this.currentObstacleIndex_
= this.checkCollision_(this.currCoordinate_, 10,
      this.arrObstacles_);
  this.currentTargetIndex_
= this.checkCollision_(this.currCoordinate_, 10,
      this.arrTargets_);
};
```

This code adds some new constants: the current position `Coordinate` of the ball, which you'll manipulate with the event handler; and the indexes for the obstacle and target that are currently being collided with. You're using the `checkCollision` method to determine collisions, just like you did when generating the obstacles and targets.

Note The delta calculation in the `getOrd` method here has been optimized for the Euler Angles published by Safari Mobile. You might need to tweak the calculation for your particular browser/platform combination.

The Draw Cycle

Use the `DrawCycle` class you built in Chapter 5 for an easy way to manage a draw cycle using `requestAnimationFrame`.

The draw cycle method will need to do three things:

- Position the ball at the `Coordinates` stored in the

`currCoordinate` property by the `deviceorientation` event handler.

- Check for a collision with an obstacle, and if one is happening:
    - add the collision class to the container,
    - remove a ball from the user's remaining number and update the display, and
    - if there are no more balls, end the game with a loss.
- Check for a collision with a target, and if one is happening,
    - hide the target element, and
    - check to see if all elements have been collected—if so, end the game with a win.

Dealing with obstacle collisions is a little complicated, because you could have a collision happening with the same obstacle for multiple iterations of the draw cycle (e.g., imagine the player is being very careful and is moving the ball very slowly, so it is in collision with an obstacle for a second or so). A collision with an obstacle should only remove one ball from the user's total no matter how long the ball is in collision with that obstacle. To prevent problems like this, store a reference to the obstacle that is currently in collision when the collision first happens. Then on subsequent draw cycles, ignore further collisions with the same element. You'll clear the reference when the collision ends.

When a collision with a target happens, you need to hide the associated target element on the game field. The easiest way to do this is to store a DOM reference in the element's `Coordinate` when you generate it in the `drawGameField` method. This will require modifications to that method, as well as to the `Coordinate` class, as shown in Listing 6-25.

Listing 6-25. The Draw Cycle, the Associated Class Properties, and Updates to Class Methods

```
 /**
 * Reference to the 'gameover' DOM element.
 * @type {Element}
 * @private
 */
this.domGameOver_ = document.getElementById('gameover');
```

```
/**
 * Reference to the 'gamefield' DOM element.
 * @type {Element}
 * @private
 */
this.gameField_ = document.getElementById('game-field');

/**
 * Reference to the ball DOM element.
 * @type {Element}
 * @private
 */
this.ball_ = document.getElementById('ball');

/**
 * The number of targets that have been collected.
 * @type {number}
 * @private
 */
this.collectedTargets_ = 0;

/**
 * Reference to the current obstacle during a collision
event. Stored between
 * draw cycles to prevent firing multiple collisions.
 * @type {Coordinate}
 * @private
 */
this.currObstacle_ = new Coordinate(0, 0);

/**
 * Initializes the obstacles and targets and draws the UI.
 * @private
 */
this.drawGameField_ = function() {
  // Clear the game field.
  this.gameField_.innerHTML = '';

  // Fill up the obstacle and target arrays with random
coordinates.
  this.generateCoords_(this.numberOfObstacles_,
this.arrObstacles_);
  this.generateCoords_(this.numberOfTargets_,
```

```
this.arrTargets_);

  // Create a div that can be used as a template for
cloning.
  var templateDiv = document.createElement('div');

  // Add the obstacles to the playing field.
  this.arrObstacles_.forEach(function(currCoord) {
    var newObstacle = templateDiv.cloneNode();
    newObstacle.classList.add('obstacle');
    newObstacle.style.left = currCoord.x + 'px';
    newObstacle.style.top = currCoord.y + 'px';
    this.gameField_.appendChild(newObstacle);
  }, this);

  // Add the targets to the playing field.
  this.arrTargets_.forEach(function(currCoord, index) {
    var newTarget = templateDiv.cloneNode();
    newTarget.classList.add('target');
    newTarget.style.left = currCoord.x + 'px';
    newTarget.style.top = currCoord.y + 'px';
    this.gameField_.appendChild(newTarget);
    // Store a reference to the new element in the array, we will
need it
    // later.
    currCoord.element = newTarget;
    this.arrTargets_.splice(index, 1, currCoord);
  }, this);

    // Update the lives displayed.
  this.updateRemainingBalls_();
};

/**
 * Draws the screen for the game: positions the 'ball' and
updates the number
 * of lives as necessary. Registered in the draw cycle.
 * @private
 */
this.drawScreen_ = function() {
  // Move the "ball."
  this.ball_.style.top = this.currCoordinate_.y + 'px';
  this.ball_.style.left = this.currCoordinate_.x + 'px';

  // Check for obstacle collisions.
```

```
  if (this.currentObstacleIndex_ > -1) {
    // Yes, there is a collision active. Check to see if
it is a new
    // collision.
    var obstacle
= this.arrObstacles_[this.currentObstacleIndex_];
    if ((this.currObstacle_.x != obstacle.x) &&
        (this.currObstacle_.y != obstacle.y)) {
      // It is a new collision.
      // Add the collision class to the game field.
      this.gameField_.classList.add('collision');
      // Store the current obstacle for the next check.
      this.currObstacle_ = obstacle;
      // A collision with an obstacle costs a life.
      this.balls_--;
      this.updateRemainingBalls_();
      // If we're out of lives, the game is over.
      if (this.balls_ <= 0) {
        this.gameOver_(false);
      }
    }
  } else {
    // There is no collision active.
    // Remove the collision class from the game field.
    this.gameField_.classList.remove('collision');
    // Clear the current obstacle cache.
    this.currObstacle_ = new Coordinate(0, 0);
  }

  // Check for target collisions.
  if (this.currentTargetIndex_ > -1) {
    // A target has been hit! Get the reference to the DOM
element.
    var hitEl
= this.arrTargets_[this.currentTargetIndex_].element;
    // If the element is not hidden, we need to hide it.
    if (!hitEl.classList.contains('hidden')) {
      hitEl.classList.add('hidden');
      // Increment the collected targets counter.
      this.collectedTargets_++;
      // If that was the last target, the game is won!
      if (this.collectedTargets_ >= this. numberOfTargets_)
```

```
{
       this.gameOver_(true);
     }
   }
  }
};

/**
 * Ends the game.
 * @param {boolean} isWon Whether the game was won or
lost.
 * @private
 */
this.gameOver_ = function(isWon) {
  if (isWon) {
    this.domGameOver_.classList.remove('loser');
    this.domGameOver_.classList.add('winner');
    this.domGameOver_.innerHTML = 'Winner!';
  } else {
    this.domGameOver_.classList.remove('winner');
    this.domGameOver_.classList.add('loser');
    this.domGameOver_.innerHTML = 'Try Again!';
  }
  this.domGameOver_.classList.remove('hidden');
  };
};

/**
 * Coordinate class.
 * @param {number} xOrd The x ordinate of the coordinate.
 * @param {number} yOrd The y ordinate of the coordinate.
 * @param {Element=} element A reference to the DOM
element for these
 *     coordinates.
 * @constructor
 */
function Coordinate(xOrd, yOrd, element) {
  this.x = xOrd;
  this.y = yOrd;
  this.element = element;
}
```

The changes to the `drawGameField` method have been bolded to make them easier to see (the rest of the method is the same as before). You've also updated the `Coordinate` class to include an optional element property, which you use to store a reference to the element at those coordinates if the element is a target.

The `drawScreen` method behaves as outlined earlier, and calls the `gameOver` method if the user wins or loses the game. The `gameOver` method shows the game over DOM element and updates its content and styling to reflect the win or loss.

Initializing the Game

There are a few things missing from the `MobiDex` class:

- You need to register the `deviceorientation` event handler.
- You need to instantiate a `DrawCycle` object and start the animation.
- You need to publish a public method on the class that can be called to start the game.
- You need to have a way to reset the game so that it can be played again.

You can use the same public method to start and restart the game because those two code paths are almost identical. The main difference is that the first two actions (creating the `DrawCycle` object and registering the event handler) should only be done once, the first time the game is started. So you'll have to break those out into a separate method and only invoke that method once.

To reset the game, you'll need to revert several of the class properties to their default values. To play the game you'll need to draw the playing field and then start the draw cycle, as shown in Listing 6-26.

Listing 6-26. Game Initialization

```
/**
 * Whether or not the game has been initialized.
 * @type {boolean}
 * @private
 */
this.isInitialized_ = false;
```

```
/**
 * The draw cycle object for the game.
 * @type {DrawCycle}
 * @private
 */
this.drawCycle_ = new DrawCycle();

/**
 * Start the game. Initializes data structures, draws the
UI, and starts the
 * animation cycle.
 */
this.startGame = function() {
  // Reset the game variables.
  this.reset_();

  // Hide the game over message.
  this.domGameOver_.classList.add('hidden');

  // Draw a random game field.
  this.drawGameField_();

  if (!this.isInitialized_) {
    this.init_();
  }

  // Start the draw cycle.
  this.drawCycle_.startAnimation();
};

/**
 * Resets game variables to their base state.
 * @private
 */
this.reset_ = function() {
  this.balls_ = 3;
  this.arrObstacles_ = [];
  this.arrTargets_ = [];
  this.collectedTargets_ = 0;
  this.currCoordinate_ = new Coordinate(0, 0);
  this.currentObstacleIndex_ = -1;
  this.currentTargetIndex_ = -1;
};
```

```
/**
 * Initialize the game for the first time.
 * @private
 */
this.init_ = function() {
  // Register the device orientation event handler on the
window object.
  window.addEventListener('deviceorientation',
      this.handleDeviceOrientation_.bind(this),
      false);

  // Add the draw method to the draw cycle.
  this.drawCycle_.addAnimation(this.drawScreen_.bind(this));

  this.isInitialized_ = true;
};
```

Now there is one public method on the class, `startGame`, that can be called when you want to start a new game, whether it's the first or subsequent games. This method initializes the game if needed, updates the default values, draws the game field, and kicks off the animation.

Assuming you have saved both the `MobiDex` and `Coordinate` classes in the file `mobidex.js`, you can now load them into your HTML document, as shown in Listing 6-27.

Listing 6-27. The Finished Game

```
<!DOCTYPE html>
<html>
    <head>
      <meta name="viewport" content="width=device-width,
user-scalable=no">
       <title>The HTML5 Programmer's Reference</title>
       <style>
// [...]
       </style>
       <script src="../js-lib/drawcycle.js"></script>
       <script src="../js-lib/mobidex.js"></script>
    </head>
    <body>
      <h1>MobiDex</h1>
```

```
      <div id="remaining-balls"></div>
      <div id="container-field">
        <div id="game-field"></div>
        <div id="ball"></div>
        <div id="gameover" class="hidden"></div>
      </div>
      <script>
// Create a new instance of the game.
var myGame = new MobiDex();

// Attach an event handler to the game over message so
that the user can restart
// the game.
document.getElementById('gameover').addEventListener('clic
function() {
  myGame.startGame();
}, false);

// Start the game.
myGame.startGame();
      </script>
    </body>
</html>
```

Here again for the interests of saving space you have elided the CSS, which hasn't changed. It not only creates a new MobiDex instance and starts the game, it also fulfills your last requirement: that the user can tap on the game over element and start a new game.

Additional Exercises

This is just the beginning for the MobiDex game. Here are some modifications you can make:

- Add scoring: For each target award a point. Carry points over through subsequent rounds. The first loss ends the game with the final score. Save the final score in local storage.
- Add a timer: Add a global timer to the game that counts down on screen. The player has to complete as many rounds as possible before the timer runs out. Save the highest number of rounds in local storage.

- Add customization: Add sliders to customize the delta calculation and make the ball move faster or slower. Add sliders to control the number of obstacles and/or targets. Save customizations in local storage.

The Full Listing

Listing 6-28 provides the entire `MobiDex` and `Coordinate` classes.

Listing 6-28. Full Listing of the MobiDex and Coordinate Classes

```
/**
 * Creates a new game. Assumes that the required DOM
elements are present.
 * @constructor
 */
function MobiDex() {

  /**
   * Whether or not the game has been initialized.
   * @type {boolean}
   * @private
   */
  this.isInitialized_ = false;

  /**
   * Reference to the 'gameover' DOM element.
   * @type {Element}
   * @private
   */
  this.domGameOver_ = document.getElementById('gameover');

  /**
   * Reference to the 'gamefield' DOM element.
   * @type {Element}
   * @private
   */
  this.gameField_ = document.getElementById('game-field');

  /**
   * Reference to the ball DOM element.
```

```
   * @type {Element}
   * @private
   */
  this.ball_ = document.getElementById('ball');

  /**
   * Reference to the 'remaining-balls' DOM element.
   * @type {Element}
   * @private
   */
  this.remainingBalls_
= document.getElementById('remaining-balls');

  /**
   * The current coordinate of the 'ball'.
   * @type {Coordinate}
   * @private
   */
  this.currCoordinate_ = new Coordinate(0, 0);

  /**
   * The index of the obstacle that the ball is currently
colliding with.
   * @type {number}
   * @private
   */
  this.currentObstacleIndex_ = -1;

  /**
   * The index of the target that the ball is currently
colliding with.
   * @type {number}
   * @private
   */
  this.currentTargetIndex_ = -1;

  /**
   * Reference to the current obstacle during a collision
event. Stored between
   * draw cycles to prevent firing multiple collisions.
   * @type {Coordinate}
   * @private
   */
```

```
this.currObstacle_ = new Coordinate(0, 0);

/**
 * Array of obstacle coordinates.
 * @type {Array.<Coordinate>}
 * @private
 */
this.arrObstacles_ = [];

/**
 * Array of target coordinates.
 * @type {Array.<Coordinate>}
 * @private
 */
this.arrTargets_ = [];

/**
 * The number of targets that have been collected.
 * @type {number}
 * @private
 */
this.collectedTargets_ = 0;

/**
 * The number of 'lives' remaining.
 * @type {number}
 * @private
 */
this.balls_ = 3;

/**
 * The draw cycle object for the game.
 * @type {DrawCycle}
 * @private
 */
this.drawCycle_ = new DrawCycle();

/**
 * The number of obstacles to draw on the game field.
 * @type {number}
 * @private
 */
this.numberOfObstacles_ = 10;
```

```
  /**
   * The number of targets to draw on the game field.
   * @type {number}
   * @private
   */
  this.numberOfTargets_ = 10;

  /**
   * Start the game. Initializes data structures, draws
the UI, and starts the
   * animation cycle.
   */
  this.startGame = function() {
    // Reset the game variables.
    this.reset_();

    // Hide the game over message.
    this.domGameOver_.classList.add('hidden');

    // Draw a random game field.
    this.drawGameField_();

    if (!this.isInitialized_) {
      this.init_();
    }

    // Start the draw cycle.
    this.drawCycle_.startAnimation();
  };

  /**
   * Resets game variables to their base state.
   * @private
   */
  this.reset_ = function() {
    this.balls_ = 3;
    this.arrObstacles_ = [];
    this.arrTargets_ = [];
    this.collectedTargets_ = 0;
    this.currCoordinate_ = new Coordinate(0, 0);
    this.currentObstacleIndex_ = -1;
    this.currentTargetIndex_ = -1;
```

```
  };

  /**
   * Initialize the game for the first time.
   * @private
   */
  this.init_ = function() {
    // Register the device orientation event handler on
the window object.
    window.addEventListener('deviceorientation',
        this.handleDeviceOrientation_.bind(this),
        false);

    // Add the draw method to the draw cycle.
    this.drawCycle_.addAnimation(this.drawScreen_.bind(this

    this.isInitialized_ = true;
  };

  /**
   * Initializes the obstacles and targets and draws the
UI.
   * @private
   */
  this.drawGameField_ = function() {
    // Clear the game field.
    this.gameField_.innerHTML = '';

    // Fill up the obstacle and target arrays with random
coordinates.
    this.generateCoords_(this.numberOfObstacles_,
this.arrObstacles_);
    this.generateCoords_(this.numberOfTargets_,
this.arrTargets_);

    // Create a div that can be used as a template for
cloning.
    var templateDiv = document.createElement('div');

    // Add the obstacles to the playing field.
    this.arrObstacles_.forEach(function(currCoord) {
      var newObstacle = templateDiv.cloneNode();
      newObstacle.classList.add('obstacle');
```

```
      newObstacle.style.left = currCoord.x + 'px';
      newObstacle.style.top = currCoord.y + 'px';
      this.gameField_.appendChild(newObstacle);
    }, this);

    // Add the targets to the playing field.
    this.arrTargets_.forEach(function(currCoord, index) {
      var newTarget = templateDiv.cloneNode();
      newTarget.classList.add('target');
      newTarget.style.left = currCoord.x + 'px';
      newTarget.style.top = currCoord.y + 'px';
      this.gameField_.appendChild(newTarget);
      // Store a reference to the new element in the array,
we will need it
      // later.
      currCoord.element = newTarget;
      this.arrTargets_.splice(index, 1, currCoord);
    }, this);

      // Update the lives displayed.
    this.updateRemainingBalls_();
  };

  /**
   * Handles a deviceorientation event from the window.
   * @param {DeviceOrientationEvent} event The device
orientation event object.
   * @private
   */
  this.handleDeviceOrientation_ = function(event) {
    // Get the x and y positions and update the current
coordiate with them.
    this.currCoordinate_.x = this.getOrd_(event.gamma,
this.currCoordinate_.x);
    this.currCoordinate_.y = this.getOrd_(event.beta,
this.currCoordinate_.y);

    // Check for collisions.
    this.currentObstacleIndex_
= this.checkCollision_(this.currCoordinate_, 10,
        this.arrObstacles_);
    this.currentTargetIndex_
= this.checkCollision_(this.currCoordinate_, 10,
```

```
      this.arrTargets_);
  };

  /**
   * Draws the screen for the game: positions the 'ball'
and updates the number
   * of lives as necessary. Registered in the draw cycle.
   * @private
   */
  this.drawScreen_ = function() {
    // Move the "ball."
    this.ball_.style.top = this.currCoordinate_.y + 'px';
    this.ball_.style.left = this.currCoordinate_.x + 'px';

    // Check for obstacle collisisons.
    if (this.currentObstacleIndex_ > -1) {
      // Yes, there is a collision active. Check to see if
it is a new
      // collision.
      var obstacle
= this.arrObstacles_[this.currentObstacleIndex_];
      if ((this.currObstacle_.x != obstacle.x) &&
          (this.currObstacle_.y != obstacle.y)) {
        // It is a new collision.
        // Add the collision class to the game field.
        this.gameField_.classList.add('collision');
        // Store the current obstacle for the next check.
        this.currObstacle_ = obstacle;
        // A collision with an obstacle costs a life.
        this.balls_--;
        this.updateRemainingBalls_();
        // If we're out of lives, the game is over.
        if (this.balls_ <= 0) {
          this.gameOver_(false);
        }
      }
    } else {
      // There is no collision active.
      // Remove the collision class from the game field.
      this.gameField_.classList.remove('collision');
      // Clear the current obstacle stored in the this.
      this.currObstacle_ = new Coordinate(0, 0);
```

```
    }

    // Check for target collisions.
    if (this.currentTargetIndex_ > -1) {
      // A target has been hit! Get the reference to the
DOM element.
      var hitEl
= this.arrTargets_[this.currentTargetIndex_].element;
      // If the element is not hidden, we need to hide it.
      if (!hitEl.classList.contains('hidden')) {
        hitEl.classList.add('hidden');
        // Increment the collected targets counter.
        this.collectedTargets_++;
        if (this.collectedTargets_ >=
this.arrTargets_.length) {
          this.gameOver_(true);
        }
      }
    }
  };

  /**
   * Updates the number of remaining balls displayed.
   * @private
   */
  this.updateRemainingBalls_ = function() {
    // Clear the current lives.
    this.remainingBalls_.innerHTML = '';
    // Create a template that we can clone and use
multiple times.
    var lifeTemplate = document.createElement('div');
    lifeTemplate.classList.add('life');
    // Add an element for each life.
    for (var i = 0; i < this.balls_; i++) {
      var currLife = lifeTemplate.cloneNode();
      currLife.style.left = (i * 15) + 'px';
      this.remainingBalls_.appendChild(currLife);
    }
  };

  /**
   * Check to see if the specified coordinates collide
```

```
with an existing set of
   * coordinates.
   * @param {Coordinate} coordinate The coordinate to
check.
   * @param {number} sensitivity The sensitivity for
a collision. If coordinates
   *     are within sensitivity distance of a target
coordinate, a collision
   *     will be registered.
   * @param {Array.<Coordinate>} arrTargetCoords An array
of target coordinates
   *     to check against.
   * @return {number} The index of the member of the
target coordinates array
   *     that is being hit, or -1 if no collision is
detected.
   * @private
   */
  this.checkCollision_ = function(coordinate, sensitivity,
arrTargetCoords) {
    // Loop through each target coordinate and compare the
provided values.
    for (var i = 0; i < arrTargetCoords.length; i++) {
      var currObstacle = arrTargetCoords[i];
      var xcoll = false;
      var ycoll = false;

      // If the provided x coordinate is within range of
the obstacle coordinate,
      // then there is an x collision.
      if (((currObstacle.x - sensitivity) < coordinate.x)
&&
          (coordinate.x < (currObstacle.x + sensitivity)))
{
        xcoll = true;
      }

      // If the provided y coordinate is within range of
the obstacle coordinate,
      // Then there is a y collision.
      if (((currObstacle.y - sensitivity) < coordinate.y)
&&
```

```
        (coordinate.y < (currObstacle.y + sensitivity)))
{
       ycoll = true;
      }

     // If there is both an x and a y collision, then
return true.
     if (xcoll && ycoll) {
       return i;
      }
    }
    return -1;
  };

  /**
   * Gets an ordinate based on a Euler Angle.
   * @param {number} angle The orientation angle that is
inducing the change.
   * @param {number} ord The previous value of the
ordinate.
   * @return {number} The new value of the ordinate.
   * @private
   */
  this.getOrd_ = function(angle, ord) {
    var delta = Math.round(angle - (angle * 0.3));
    var tempVal = ord + delta;
    if (tempVal > 0) {
      ord = Math.min(192, tempVal);
    } else {
      ord = 0;
    }
    return ord;
  };

  /**
   * Generates a set of random coordinates and adds them
to the provided array.
   * Tries to avoid duplicating too closely any
previously-generated
   *     coordinates.
   * @param {number} numberOfCoords The number of
coordinates to generate.
```

```
   * @param {Array} targetArray The array to fill with the
new coordinates.
   * @private
   */
  this.generateCoords_ = function(numberOfCoords,
targetArray) {
    for (var i = 0; i < numberOfCoords; i++) {
      var newCoord = new
Coordinate(this.getRandomIntegerBetween_(10, 190),
          this.getRandomIntegerBetween_(10, 190));
      while (this.checkCollision_(newCoord, 15,
           this.arrObstacles_.concat(this.arrTargets_)) > -
1) {
        newCoord.x = this.getRandomIntegerBetween_(10,
190);
        newCoord.y = this.getRandomIntegerBetween_(10,
190);
      }
      targetArray.push(newCoord);
    }
  };

  /**
   * Returns a random integer between the specified
minimum and maximum values.
   * @param {number} min The lower boundary for the random
number.
   * @param {number} max The upper boundary for the random
number.
   * @return {number}
   * @private
   */
  this.getRandomIntegerBetween_ = function(min, max) {
    return Math.floor(Math.random() * (max - min + 1))
+ min;
  };

  /**
   * Ends the game.
   * @param {boolean} isWinner Whether the game was won or
lost.
   * @private
```

```
   */
  this.gameOver_ = function(isWon) {
    this.drawCycle_.stopAnimation();
    if (isWon) {
      this.domGameOver_.classList.remove('loser');
      this.domGameOver_.classList.add('winner');
      this.domGameOver_.innerHTML = 'Winner!';
    } else {
      this.domGameOver_.classList.remove('winner');
      this.domGameOver_.classList.add('loser');
      this.domGameOver_.innerHTML = 'Try Again!';
    }
    this.domGameOver_.classList.remove('hidden');
  };
};

/**
 * Coordinate class.
 * @param {number} xOrd The x ordinate of the coordinate.
 * @param {number} yOrd The y ordinate of the coordinate.
 * @param {Element} element A reference to the DOM element
for these
 *     coordinates.
 * @constructor
 */
function Coordinate(xOrd, yOrd, element) {
  this.x = xOrd;
  this.y = yOrd;
  this.element = element;
}
```

Summary

In this chapter I have discussed working with HTML5 in practical projects. I've covered:

- feature detection,
- dynamically responding to different levels of HTML5 support, and
- online resources for researching HTML5 support and locating

shims.

You have also built an entire HTML5 mobile game from scratch, starting with user stories and ending with working code.

This concludes the discussion chapters for the book. The next chapters will all be reference chapters for HTML5 features, starting with the HTML5 Element Reference.

PART II

HTML5 Reference

CHAPTER 7

HTML5 Element Reference

This chapter provides a detailed reference for all of the new HTML5 elements. The elements are grouped semantically, so all of the elements that provide sectioning semantics are together, all of the elements that provide grouping semantics are together, etc. Each element entry will have the following:

- A brief description of the element and its function.
- A Usage section that includes the syntax of the element and a brief example.
- A Properties section that lists all of the properties that can be set on the element.
- A table with all of the relevant standards for the element.

All of these same elements are covered in more detail in Chapter 2. You can find in-depth discussions there as well as many more examples and browser support at time of press.

Sections

HTML5 defines several new elements for marking up sections of content within a larger document. These sections are typically self-contained or distinct groups of content. Some sections are repeatable within a single document. Section elements help provide an overall structure to the document.

The `article` Element

The `article` element is used to denote a section of self-contained, stand-alone content within a larger document: a single blog post within a page of blog posts, a single story within a newspaper page, or a single advertisement on a larger page.

Usage

The element is used to denote a block section of content, and is rendered as a block element in the document flow (like a `div` or heading element). The closing tag is required.

Syntax

```
<article>...</article>
```

Listing 7-1. The article Element

```
<!DOCTYPE html>
<html>
    <head>
       <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
      <article>
        <h1>First Article Title</h1>
        <p>Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Vestibulum
          tempus in nisi id gravida. Nullam vitae velit
tincidunt, vulputate
          arcu nec, ullamcorper velit. In in nulla tellus.
</p>
        <p>Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Vestibulum
          tempus in nisi id gravida. Nullam vitae velit
tincidunt, vulputate
          arcu nec, ullamcorper velit. In in nulla tellus.
</p>
      </article>
      <article>
        <h1>Second Article Title</h1>
        <p>Lorem ipsum dolor sit amet, consectetur
```

```
adipiscing elit. Vestibulum
          tempus in nisi id gravida. Nullam vitae velit
tincidunt, vulputate
          arcu nec, ullamcorper velit. In in nulla tellus.
</p>
       <p>Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Vestibulum
          tempus in nisi id gravida. Nullam vitae velit
tincidunt, vulputate
          arcu nec, ullamcorper velit. In in nulla tellus.
</p>
     </article>
    </body>
</html>
```

Properties

- accesskey
- class
- classlist
- contenteditable
- contextmenu
- dataset
- dir
- draggable
- dropzone
- hidden
- id
- lang
- spellcheck
- style
- tabindex
- title

- translate

Table 7-1. *Standards for the* `article` *Element*

Specification	Status	URL
W3C	Candidate Recommendation	www.w3.org/TR/html5/sections.html#the-article-element
WHATWG	Living Standard	www.whatwg.org/specs/web-apps/current-work/multipage/semantics.html#the-article-element

The `aside` Element

The `aside` element denotes content that is tangential or loosely related to its containing content: a sidebar, note, or comment. Omission of the content within the `aside` element should not affect the meaning of the containing content.

Usage

The element is used to denote a block section of content, and is rendered as a block element in the document flow (like a `div` or heading element). The closing tag is required.

Syntax

```
<aside>...</aside>
```

Listing 7-2. The aside Element

```
<!DOCTYPE html>
<html>
    <head>
       <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
      <article>
        <h1>Article Title</h1>
        <p>Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Vestibulum
```

```
          tempus in nisi id gravida. Nullam vitae velit
tincidunt, vulputate
          arcu nec, ullamcorper velit. In in nulla tellus.
</p>
        <aside>
          <h2>Aside Title</h2>
          <p>Lorem ipsum dolor sit amet, consectetur nisi
id gravida.</p>
          <ul>
            <li>Item</li>
            <li>Item</li>
            <li>Item</li>
          </ul>
        </aside>
        <p>Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Vestibulum
          tempus in nisi id gravida. Nullam vitae velit
tincidunt, vulputate
          arcu nec, ullamcorper velit. In in nulla tellus.
</p>
      </article>
    </body>
</html>
```

Properties

This element can have any of the "global" properties, which are standard for all HTML elements. These properties include:

- accesskey
- class
- contenteditable
- contextmenu
- dataset
- dir
- draggable
- hidden

- id
- lang
- spellcheck
- style
- tabindex
- title

Table 7-2. *Standards for the* `aside` *Element*

Specification	Status	URL
W3C	Candidate Recommendation	www.w3.org/TR/html5/sections.html#the-aside-element
WHATWG	Living Standard	www.whatwg.org/specs/web-apps/current-work/multipage/semantics.html#the-aside-element

The `footer` Element

The `footer` element is used to denote content that comes at the end of the containing section. Footers typically provide information about their containing sections. Each section should have at most one `footer`.

Usage

The element is used to denote a block section of content, and is rendered as a block element in the document flow (like a `div` or heading element). The closing tag is required. The `footer` element may not contain `header`, `footer`, or `main` elements.

Syntax

```
<footer>...</footer>
```

Listing 7-3. The *footer* Element

```
<!DOCTYPE html>
<html>
```

```
    <head>
       <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
      <article>
        <h1>Article Title</h1>
        <p>Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Vestibulum
          tempus in nisi id gravida. Nullam vitae velit
tincidunt, vulputate
          arcu nec, ullamcorper velit. In in nulla tellus.
</p>
        <p>Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Vestibulum
          tempus in nisi id gravida. Nullam vitae velit
tincidunt, vulputate
          arcu nec, ullamcorper velit. In in nulla tellus.
</p>
        <footer>
          <p>Sitemap:</p>
          <ul>
            <li>link</li>
            <li>link</li>
            <li>link</li>
          </ul>
          <p>Copyright notice.</p>
        </footer>
      </article>
    </body>
</html>
```

Properties

This element can have any of the “global” properties, which are standard for all HTML elements. These properties include:

- accesskey
- class
- classlist
- contenteditable

- contextmenu
- dataset
- dir
- draggable
- dropzone
- hidden
- id
- lang
- spellcheck
- style
- tabindex
- title

Table 7-3. *Standards for the* `footer` *Element*

Specification	Status	URL
W3C	Candidate Recommendation	www.w3.org/TR/html5/sections.html#the-footer-element
WHATWG	Living Standard	www.whatwg.org/specs/web-apps/current-work/multipage/semantics.html#the-footer-element

The `header` Element

The `header` element is used to denote a set of introductory content at the beginning of a section. Each section should have at most one header.

Usage

The element is used to denote a block section of content, and is rendered as a block element in the document flow (like a `div` or heading element). The closing tag is required. The `header` element may not contain `header`, `footer`, or `main` elements.

Syntax

```
<header>...</header>
```

Listing 7-4. The header Element

```
<!DOCTYPE html>
<html>
    <head>
       <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
      <article>
        <header>
          <h1>Article Title</h1>
          <ul>
            <li>navlink 1</li>
            <li>navlink 2</li>
            <li>navlink 3</li>
          </ul>
        </header>
        <p>Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Vestibulum
          tempus in nisi id gravida. Nullam vitae velit
tincidunt, vulputate
          arcu nec, ullamcorper velit. In in nulla tellus.
</p>
        <p>Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Vestibulum
          tempus in nisi id gravida. Nullam vitae velit
tincidunt, vulputate
          arcu nec, ullamcorper velit. In in nulla tellus.
</p>
      </article>
    </body>
</html>
```

Properties

This element can have any of the "global" properties, which are standard for all HTML elements. These properties include:

- accesskey

- class
- classlist
- contenteditable
- contextmenu
- dataset
- dir
- draggable
- dropzone
- hidden
- id
- lang
- spellcheck
- style
- tabindex
- title

Table 7-4. *Standards for the* `header` *Element*

Specification	Status	URL
W3C	Candidate Recommendation	www.w3.org/TR/html5/sections.html#the-header-element
WHATWG	Living Standard	www.whatwg.org/specs/web-apps/current-work/multipage/semantics.html#the-header-element

The `nav` Element

The `nav` element is used to denote a navigation section with major navigation links to other pages or to content or sections within the current document.

Usage

The element is used to denote a block section of content, and is rendered as a block

element in the document flow (like a `div` or heading element). The closing tag is required.

Syntax

```
<nav>...</nav>
```

Listing 7-5. The `nav` Element

```
<!DOCTYPE html>
<html>
    <head>
       <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
      <article>
        <header>
          <h1>Article Title</h1>
          <nav>
            <ul>
              <li>navlink 1</li>
              <li>navlink 2</li>
              <li>navlink 3</li>
            </ul>
          </nav>
        </header>
        <p>Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Vestibulum
          tempus in nisi id gravida. Nullam vitae velit
tincidunt, vulputate
          arcu nec, ullamcorper velit. In in nulla tellus.
</p>
       <p>Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Vestibulum
          tempus in nisi id gravida. Nullam vitae velit
tincidunt, vulputate
          arcu nec, ullamcorper velit. In in nulla tellus.
</p>
     </article>
    </body>
</html>
```

Properties

This element can have any of the "global" properties, which are standard for all HTML elements. These properties include:

- `accesskey`
- `class`
- `classlist`
- `contenteditable`
- `contextmenu`
- `dataset`
- `dir`
- `draggable`
- `dropzone`
- `hidden`
- `id`
- `lang`
- `spellcheck`
- `style`
- `tabindex`
- `title`

Table 7-5. *Standards for the* `nav` *Element*

Specification	Status	URL
W3C	Candidate Recommendation	`www.w3.org/TR/html5/sections.html#the-nav-element`
WHATWG	Living Standard	`www.whatwg.org/specs/web-apps/current-work/multipage/semantics.html#the-nav-element`

The `section` Element

The `section` element is used to indicate a generic group of content, typically covering

one specific theme. Often, the theme of the section is denoted by a child header element.

Usage

The `usage` element is used to denote a block section of content, and is rendered as a block element in the document flow (like a `div` or heading element). The closing tag is required.

Syntax

```
<section>...</section>
```

Listing 7-6. The `section` Element

```
<!DOCTYPE html>
<html>
    <head>
       <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
      <article>
        <section>
          <h1>Introduction</h1>
          <p>Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Vestibulum
           tempus in nisi id gravida. Nullam vitae velit
tincidunt, vulputate
           arcu nec, ullamcorper velit. In in nulla tellus.
</p>
        </section>
        <section>
          <h1>First Section</h1>
          <p>Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Vestibulum
           tempus in nisi id gravida. Nullam vitae velit
tincidunt, vulputate
           arcu nec, ullamcorper velit. In in nulla tellus.
</p>
        </section>
        <section>
          <h1>Second Section</h1>
```

```
        <p>Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Vestibulum
          tempus in nisi id gravida. Nullam vitae velit
tincidunt, vulputate
          arcu nec, ullamcorper velit. In in nulla tellus.
</p>
      </section>
    </article>
  </body>
</html>
```

Properties

This element can have any of the "global" properties, which are standard for all HTML elements. These properties include:

- accesskey
- class
- classlist
- contenteditable
- contextmenu
- dataset
- dir
- draggable
- dropzone
- hidden
- id
- lang
- spellcheck
- style
- tabindex
- title

Table 7-6. *Standards for the* `section` *Element*

Specification	Status	URL
W3C	Candidate Recommendation	`www.w3.org/TR/html5/sections.html#the-section-element`
WHATWG	Living Standard	`www.whatwg.org/specs/web-apps/current-work/multipage/semantics.html#the-section-element`

Grouping

The HTML5 standard defines a new set of elements to group data together by type, as distinguished from the section elements, which are used to provide overall document structure.

The `figure` and `figcaption` Elements

The `figure` element is used to group together a self-contained group of data that is referenced as a single set from the main content of the document. Typical uses are for illustrations, diagrams, code samples, and so on. The `figcaption` element is optionally used to provide a caption for a figure element.

Usage

The elements are used to denote block sections of content, and are rendered as a block element in the document flow (like a `div` or heading element). The closing tag is required for both elements. The `figcaption` element is optional, but must be a child of a `figure` element.

Syntax

```
<figure>
    <figcaption>...</figcaption>
    ...
</figure>
```

Listing 7-7. The `figure` and `figcaption` Elements

```
<!DOCTYPE html>
<html>
    <head>
       <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
      <article>
        <h1>Title</h1>
        <figure>
          <figcaption>Caption: Source Information</figcaption>
          <img src="diagrams/diagram1-1.png" alt="Schematic
diagram" />
        </figure>
      </article>
    </body>
</html>
```

Properties

This element can have any of the “global” properties, which are standard for all HTML elements. These properties include:

- accesskey
- class
- classlist
- contenteditable
- contextmenu
- dataset
- dir
- draggable
- dropzone
- hidden
- id
- lang
- spellcheck

- `style`
- `tabindex`
- `title`

Table 7-7. *Standards for the* `figure and figcaption` *Elements*

Specification	Status	URL
W3C	Candidate Recommendation	www.w3.org/TR/html5/grouping-content.html#the-figure-element www.w3.org/TR/html5/grouping-content.html#the-figcaption-element
WHATWG	Living Standard	www.whatwg.org/specs/web-apps/current-work/multipage/semantics.html#the-figure-element www.whatwg.org/specs/web-apps/current-work/multipage/semantics.html#the-figcaption-element

The `main` Element

The `main` element is used as a container for the dominant contents of another element. The main element itself does not provide structure or contribute to the document's outline. It only provides a grouping container.

Usage

The element is used to denote a block section of content, and is rendered as a block element in the document flow (like a `div` or heading element). The closing tag is required.

Syntax

```
<main>...</main>
```

Listing 7-8. The `main` Element

```
<!DOCTYPE html>
<html>
    <head>
```

```
      <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
      <article>
        <h1>Article Title</h1>
        <main>
          <p>Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Vestibulum
            tempus in nisi id gravida. Nullam vitae velit
tincidunt, vulputate
            arcu nec, ullamcorper velit. In in nulla
tellus.</p>
          <p>Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Vestibulum
            tempus in nisi id gravida. Nullam vitae velit
tincidunt, vulputate
            arcu nec, ullamcorper velit. In in nulla
tellus.</p>
        </main>
        <aside>
          <p>Lorem ipsum dolor sit amet, consectetur
adipiscing elit.</p>
        </aside>
      </article>
    </body>
</html>
```

Properties

This element can have any of the "global" properties, which are standard for all HTML elements. These properties include:

- accesskey
- class
- classlist
- contenteditable
- contextmenu
- dataset

- dir
- draggable
- dropzone
- hidden
- id
- lang
- spellcheck
- style
- tabindex
- title

Table 7-8. *Standards for the* `main` *Element*

Specification	Status	URL
W3C	Candidate Recommendation	www.w3.org/TR/html5/grouping-content.html#the-main-element
WHATWG	Living Standard	www.whatwg.org/specs/web-apps/current-work/multipage/semantics.html#the-main-element

Semantics

The HTML5 standard specifies several new elements designed to provide more capabilities for defining the semantic purposes of portions of data. Because they are meant to provide semantic detail rather than structure, these tags are rendered as inline elements.

Unfortunately, there is little support for many of these tags in current browser implementations.

The `bdi` Element

The `bdi` element (`bdi` is an abbreviation for "bi-directional isolation") is used to isolate a portion of text that might be formatted in a different direction than the surrounding text—for example, when directly including Arabic text in an otherwise

English page.

Usage

The element is used to denote a portion of text contained within other text, and as such is rendered as an inline element.

Syntax

```
<bdi>...</bdi>
```

Listing 7-9. The `bdi` Element

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
      <article>
        <h1>Article Title</h1>
        <p>Lorem ipsum dolor sit amet, <bdi>consectetur
adipiscing elit</bdi>. Vestibulum
          tempus in nisi id gravida. Nullam vitae velit
tincidunt, vulputate
          arcu nec, ullamcorper velit. In in nulla tellus.
</p>
      </article>
    </body>
</html>
```

Properties

This element can have any of the “global” properties, which are standard for all HTML elements. These properties include:

- `accesskey`
- `class`

- classlist
- contenteditable
- contextmenu
- dataset
- dir
- draggable
- dropzone
- hidden
- id
- lang
- spellcheck
- style
- tabindex
- title

Table 7-9. *Standards for the* `bdi` *Element*

Specification	Status	URL
W3C	Candidate Recommendation	www.w3.org/TR/html5/text-level-semantics.html#the-bdi-element
WHATWG	Living Standard	www.whatwg.org/specs/web-apps/current-work/multipage/semantics.html#the-bdi-element

The `data` Element

The `data` element is used to denote machine-readable data embedded in a document. Typically the data will be embedded in the element using a type or data attribute.

Usage

This element is used to embed machine-readable data into a document. It is not typically rendered. A closing tag is not required.

Syntax

```
<data value="someval">
```

Listing 7-10. The `data` Element

```
<!DOCTYPE html>
<html>
    <head>
       <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
      <article>
        <h1>Article Title</h1>
        <ul>
          <li><data value="ser-123-456">Serial Number
1</li>
          <li><data value="ser-123-856">Serial Number
2</li>
          <li><data value="ser-123-204">Serial Number
3</li>
        </ul>
      </article>
    </body>
</html>
```

Properties

This element can have any of the "global" properties, which are standard for all HTML elements. These properties include:

- `accesskey`
- `class`
- `classlist`
- `contenteditable`
- `contextmenu`
- `dataset`

- dir
- draggable
- dropzone
- hidden
- id
- lang
- spellcheck
- style
- tabindex
- title

Table 7-10. *Standards for the* `data` *Element*

Specification	Status	URL
W3C	Candidate Recommendation	www.w3.org/TR/html5/text-level-semantics.html#the-data-element
WHATWG	Living Standard	www.whatwg.org/specs/web-apps/current-work/multipage/semantics.html#the-data-element

The `mark` Element

The `mark` element is used to denote occurrences within a set of data. The occurrence itself is context specific.

Usage

This element is used to denote portions of data within larger contents, and as such is rendered as an inline element. The closing tag is required.

Syntax

```
<mark>...</mark>
```

Listing 7-11. The `mark` Element

```
<!DOCTYPE html>
<html>
    <head>
       <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
      <article>
       <h1>Article Title</h1>
       <p>Lorem ipsum dolor sit amet,
<mark>consectetur</mark> adipiscing elit. Vestibulum
          tempus in nisi id gravida. Nullam vitae velit
tincidunt, vulputate
          arcu nec, ullamcorper velit. In in nulla tellus.
</p>
       <p>Lorem ipsum dolor sit amet,
<mark>consectetur</mark> adipiscing elit. Vestibulum
          tempus in nisi id gravida. Nullam vitae velit
tincidunt, vulputate
          arcu nec, ullamcorper velit. In in nulla tellus.
</p>
     </article>
    </body>
</html>
```

Properties

This element can have any of the "global" properties, which are standard for all HTML elements. These properties include:

- `accesskey`
- `class`
- `classlist`
- `contenteditable`
- `contextmenu`
- `dataset`
- `dir`

- draggable
- dropzone
- hidden
- id
- lang
- spellcheck
- style
- tabindex
- title

Table 7-11. *Standards for the* `mark` *Element*

Specification	Status	URL
W3C	Candidate Recommendation	www.w3.org/TR/html5/text-level-semantics.html#the-mark-element
WHATWG	Living Standard	www.whatwg.org/specs/web-apps/current-work/multipage/semantics.html#the-mark-element

The `ruby`, `rp`, and `rt` Elements

The `ruby`, `rp`, and `rt` elements are used for creating Ruby annotations, which are short runs of text presented next to main text. Typically Ruby annotations are used to indicate pronunciation in East Asian languages. For details about Ruby annotations, see www.w3.org/TR/ruby/ and http://en.wikipedia.org/wiki/Ruby_character.

Usage

A `ruby` element typically consists of a set of content surrounded by a `ruby` tag, with one or more `rp` or `rt` annotations.

Syntax

```
<ruby>base<rt>annotation</ruby>
```

```
<ruby><rb>base<rt>annotation</ruby>
```

Listing 7-12. The `ruby`, `rt`, and `rp` Elements

```
<!DOCTYPE html>
<html>
    <head>
       <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
      <article>
        <h1>Article Title</h1>
        <ruby>B<rt>a<rt>a</ruby><ruby>A<rt>a<rt>a</ruby>
        <ruby>S<rt>a<rt>a</ruby><ruby>E<rt>a<rt>a</ruby>
        <ruby>BASE<rt>annotation 1<rt>annotation 2</ruby>
      </article>
    </body>
</html>
```

Properties

These elements can have any of the "global" properties, which are standard for all HTML elements. These properties include:

- `accesskey`
- `class`
- `classlist`
- `contenteditable`
- `contextmenu`
- `dataset`
- `dir`
- `draggable`
- `dropzone`
- `hidden`
- `id`

- `lang`
- `spellcheck`
- `style`
- `tabindex`
- `title`

Table 7-12. *Standards for the* `ruby`, `rt`, *and* `rp` *Elements*

Specification	Status	URL
W3C	Candidate Recommendation	www.w3.org/TR/html5/text-level-semantics.html#the-ruby-element www.w3.org/TR/html5/text-level-semantics.html#the-rt-element www.w3.org/TR/html5/text-level-semantics.html#the-rp-element
WHATWG	Living Standard	www.whatwg.org/specs/web-apps/current-work/multipage/semantics.html#the-ruby-element www.whatwg.org/specs/web-apps/current-work/multipage/semantics.html#the-rt-element www.whatwg.org/specs/web-apps/current-work/multipage/semantics.html#the-rp-element

The `time` Element

Similar to the `data` element, the `time` element is used to denote machine-readable date/time data embedded in a document.

Usage

This element is used to embed machine-readable data into a document. A closing tag is required.

Syntax

```
<time>...</time>
```

Listing 7-13. The `time` Element

```
<!DOCTYPE html>
<html>
    <head>
       <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
      <article>
        <h1>Article Title</h1>
        <time>2011-11-12T14:54</time>
        <time>2011-11-12T14:54:39</time>
        <time>2011-11-12T14:54:39.929</time>
        <time>2011-11-12 14:54</time>
        <time>2011-11-12 14:54:39</time>
        <time>2011-11-12 14:54:39.929</time>
      </article>
    </body>
</html>
```

Properties

This element can have any of the “global” properties, which are standard for all HTML elements. These properties include:

- accesskey
- class
- classlist
- contenteditable
- contextmenu
- dataset
- dir
- draggable
- dropzone
- hidden
- id
- lang

- spellcheck
- style
- tabindex
- title

Table 7-13. *Standards for the* `time` *Element*

Specification	Status	URL
W3C	Candidate Recommendation	www.w3.org/TR/html5/text-level-semantics.html#the-time-element
WHATWG	Living Standard	www.whatwg.org/specs/web-apps/current-work/multipage/semantics.html#the-time-element

The `wbr` Element

The Word Break Opportunity tag is used to indicate a position in the document flow where the browser may initiate a line break though its internal rules might not otherwise do so. It has no effect on bidi-ordering, and if the browser does initiate a break at the tag, a hyphen is not used.

Usage

This element is used to indicate word break opportunities in text, and so is only rendered if a word break is needed. As such, it is expected to be contained within other block elements such as paragraphs. A closing tag is not required.

Syntax

```
<wbr>
```

Listing 7-14. The `wbr` Element

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
```

```
    </head>
    <body>
      <article>
        <h1>Article Title</h1>
        <p>Supercali<wbr>fragilistic<wbr>expialidocious and
          antidis<wbr>establishment<wbr>arianism.</p>
      </article>
    </body>
</html>
```

Properties

This element can have any of the "global" properties, which are standard for all HTML elements. These properties include:

- `accesskey`
- `class`
- `classlist`
- `contenteditable`
- `contextmenu`
- `dataset`
- `dir`
- `draggable`
- `dropzone`
- `hidden`
- `id`
- `lang`
- `spellcheck`
- `style`
- `tabindex`
- `title`

Table 7-14. *Standards for the* `wbr` *Element*

Specification	Status	URL
W3C	Candidate Recommendation	www.w3.org/TR/html5/text-level-semantics.html#the-wbr-element
WHATWG	Living Standard	www.whatwg.org/specs/web-apps/current-work/multipage/semantics.html#the-wbr-element

Audio and Video Content

The `audio` Element

The `audio` element is used to embed sound content (typically audio files) in web pages.

In the past, embedding audio content in documents typically required the use of a plug-in (most typically Flash). This had the benefit of being fairly ubiquitous because as long as the target browser had the plug-in installed, it would be able to play the content. All of the complexity around the user interface controls, handling different file formats, and special features like dynamic streaming were all handled by the plug-in software.

When implementing the ability to embed sound content, web browser manufacturers had to handle these issues themselves. As a result, the appearance and functionality of the user interface controls for the audio player vary from browser to browser.

Each browser also supports different file formats due to patent encumbrances, and some browsers support different file formats on different operating systems depending on locally installed software. An in-depth discussion of audio file formats, their patent issues, and operating system support is beyond the scope of this book, but you can find a great deal of information on the Web. Specifically:

- https://developer.mozilla.org/en-US/docs/Web/HTML/Supported_media_formats The Mozilla Developer Network has a good page that discusses the various audio formats and their support in major browsers.
- http://hpr.dogphilosophy.net/test/index.php is a page you can visit to test the support of various audio formats in your browsers. The page also has some good information on the various formats and the state of their support in the major browsers.
- www.jwplayer.com/html5/formats/ The JW Player is a proprietary audio/video player based on HTML5 technology (the

core of the player is open source). The company has an obvious interest in the state of HTML5 audio and video support, and they maintain their own statistics on the topic.

There are other sources available on the Web, but many of them seem to be out of date (or it wasn't possible to verify when they were last updated).

Usage

The element is used to embed sound content in documents. The content can be specified using either the `src` attribute or by using `source` elements contained within the `audio` element. For details on using `source` elements, see the next section.

The element can also contain zero or more `track` elements to specify time-based data for the audio content (such as captions). See the section on the `track` element for details.

Additionally, the element can optionally contain other elements that will be rendered if the browser does not support the audio element.

The `<audio>` tag is not self-closing so both the start and end tags are required.

Syntax

```
<audio></audio>
```

Listing 7-15. The `audio` Element

```
<!DOCTYPE html>
<html>
    <head>
       <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
      <p>Basic</p>
      <audio controls="true" src="../media/winamp-
llama.mp3">
        <p>Your browser does not support the HTML5 audio
tag.</p>
      </audio>
      <p>Using source Elements</p>
      <audio controls="true">
```

```
        <source src="testfile.mp3" type="audio/mpeg">
        <source src="testfile.ogg" type="audio/ogg">
        <p>Your browser does not support the HTML5 audio
tag.</p>
      </audio>
    </body>
</html>
```

Properties

The audio element supports the following properties:

- `autoplay`: This is a boolean flag that, when set (to anything, even `false`), will cause the browser to immediately begin playing the audio content as soon as it can without stopping for buffering.
- `controls`: If this attribute is set, then the browser will display its default user interface controls for the audio player (volume controls, progress meter/scrub bar, etc.).
- `loop`: If this attribute is set, the browser will loop playback of the specified file.
- `muted`: This attribute specifies that the playback should be muted by default.
- `preload`: This attribute is used to give the browser a hint for how to provide the best user experience for the specified content. It can take three values:
    - `none` specifies that the author wants to minimize the download of the audio content, possibly because the content is optional, or because the server resources are limited.
    - `metadata` specifies that the author recommends downloading the metadata for the audio content (duration, track list, tags, etc.) and possibly the first few frames of the content.
    - `auto` specifies that the browser can put the user's needs first without risk to the server. This means the browser can begin buffering the content, download all the metadata, and so on.

- `src`: This attribute specifies the source of the content, just as with an `img` element. If desired, this attribute can be omitted in favor of one or more `source` elements contained within the `audio` element.

In addition, the `audio` element supports the following global attributes:

- `accesskey`
- `class`
- `classlist`
- `contenteditable`
- `contextmenu`
- `dataset`
- `dir`
- `draggable`
- `dropzone`
- `hidden`
- `id`
- `lang`
- `spellcheck`
- `style`
- `tabindex`
- `title`

Table 7-15. *Standards for the* `audio` *Element*

Specification	Status	URL
W3C	Candidate Recommendation	`www.w3.org/TR/html5/embedded-content-0.html#the-audio-element`
WHATWG	Living Standard	`www.whatwg.org/specs/web-apps/current-work/multipage/semantics.html#the-video-element`

The `source` Element

The `source` element is used to specify a single content source for its parent element. As a result, source elements must always be contained within either audio or video elements. It does not represent anything on its own.

Usage

The `source` element is used to specify a single content source for an `audio` or `video` element. Multiple `source` elements are permitted. If multiple `source` elements are contained within an `audio` or `video` element, the browser will examine each one in order and fetch and play the first one that specifies content encoded in a manner it supports. This provides a workaround for the fragmented support for audio and video formats in various browsers: simply encode the desired content in the required formats, and specify the location of the different encodings using as many source elements as are required.

The `<source>` tag does not require a closing tag, and is not otherwise rendered in the document. All `source` elements must come before any `track` elements.

Syntax

```
<source src="testfile.mp3">
```

Listing 7-16. The `source` Element

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
      <audio controls="true">
        <source src="testfile.mp3" type="audio/mpeg">
        <source src="testfile.ogg" type="audio/ogg">
        <p>Your browser does not support the HTML5 audio
tag.</p>
     </audio>
     <video controls="true">
       <source src='video.mp4' type='video/mp4;
```

```
codecs="avc1.42E01E, mp4a.40.2"'>
       <source src='video.mp4' type='video/mp4;
codecs="avc1.58A01E, mp4a.40.2"'>
       <source src='video.3gp' type='video/3gpp;
codecs="mp4v.20.8, samr"'>
       <p>Your browser does not support the HTML5 video
tag.</p>
     </video>
   </body>
</html>
```

Properties

A `source` element has two properties:

- `src`: The `src` property is used to provide an address of a media resource appropriate for the containing element. This property is required.
- `type`: The `type` property is used to specify the MIME type of the media resource. This type attribute is used by the browser to determine if it can play the media resource. If it can't play the media resource, the browser will not attempt to fetch it and will move on to the next `source` element (if any). The `type` property may contain an optional `codec` parameter that specifies the codec(s) used to create the specified media. The syntax of the `codec` parameter is governed by RFC6381, "The `codecs` and `profiles` Parameters for Bucket Media Types."

Table 7-16. *Standards for the `source` Element*

Specification	Status	URL
Internet Engineering Taskforce (IETF)	Proposed Standard	http://tools.ietf.org/html/rfc6381
W3C	Language Reference	www.w3.org/TR/html-markup/source.html
WHATWG	Living Standard	www.whatwg.org/specs/web-apps/current-work/multipage/embedded-content.html#the-source-element

The `track` Element

The `track` element is used to specify time-based data for an audio or video element, such as closed captioning or subtitles. Like `source` elements, `track` elements do not define any content on their own, and must be children of audio or video elements. The time-based data specified by the `track` element could be formatted in any way supported by the browser; the most common format is the new WebVTT format.

WebVTT-Formatted Data

The Web Video Text Tracks Format (WebVTT) standard specifies a specific schema or format for a text file (UTF-8 encoded) that associates arbitrary data with points in time. Typically the data is captioning information, but it could be any data in any desired format, including XML, HTML, or even JSON.

A valid WebVTT file consists of the `WEBVTT` declaration, an optional description next to the declaration, and zero or more cues or comments. Thus the file

```
WEBVTT
```

is a valid WebVTT file.

A valid cue consists of an optional cue identifier, followed on the next line by cue timings (which may also include cue settings), followed on the next line by the contents of the cue. A simple example would look as shown in Listing 7-17.

Listing 7-17. A Sample WebVTT Closed Captioning File for King Lear

```
WEBVTT

1 - Act 1, Scene 1
00:00:1.000 --> 00:00:1.500
Scene: King Lear's Palace
Enter Kent, Gloucester, and Edmund.

00:00:1.500 --> 00:00:2.000 position:10%  size:50%
<v Kent> I thought the king had more affected the Duke of
Albany than Cornwall.

00:00:2.100 --> 00:00:3.500 position:10%  size:50%
<v Gloucester> It did always seem so to us: but now, in
the
```

```
division of the kingdom, it appears not which of
the dukes he values most; for equalities are so
weighed, that curiosity in neither can make choice
of either's moiety.
```

In this example there are three separate cues, each with a timestamp range. As the video plays, each cue is displayed at the appropriate time. The WebVTT standard includes the ability to format the resulting captions; in this example the dialog cue boxes are limited to 50% of the width of the video viewport and are positioned 10% of the total viewport width away from the left of the viewport.

The WebVTT standard is extensive, and I'm not going to cover it fully here. For details, see the W3C standard at `http://dev.w3.org/html5/webvtt/`. There is also a great tutorial at the HTML5 Doctor web site available at `http://html5doctor.com/video-subtitling-and-webvtt/`.

Usage

The `track` element is used to specify a file containing time-based data for the containing audio or video element. Multiple `track` elements are permitted for the same audio or video element.

The `track` tag is self-closing and does not require a closing tag, and `track` tags must come after any source tags.

Syntax

```
<track src="karaoki.vtt">
```

Listing 7-18. The `track` Element

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
      <audio controls="true">
        <source src="testfile.mp3" type="audio/mpeg">
        <source src="testfile.ogg" type="audio/ogg">
        <track src="karaoki.vtt" kind="captions"
label="Karaoki Cues">
```

```
      <p>Your browser does not support the HTML5 audio
tag.</p>
    </audio>
    <video controls="true">
      <source src='thetwotowers.mp4' type='video/mp4;
codecs="avc1.42E01E, mp4a.40.2"'>
      <source src='thetwotowers.mp4' type='video/mp4;
codecs="avc1.58A01E, mp4a.40.2"'>
      <source src='thetwotowers.3gp' type='video/3gpp;
codecs="mp4v.20.8, samr"'>
      <track src="closed-captioning.vtt" kind="captions"
label="Closed Captioning">
      <track src="peter_fran_philippa.vtt"
kind="subtitles" src="en" label="Director and
       Writer Scene Notes">
      <p>Your browser does not support the HTML5 video
tag.</p>
    </video>
  </body>
</html>
```

Properties

The `track` element has the following properties:

- `default`: Indicates that this is the default track for the content and should be the one that is displayed unless overridden by the user.
- `kind`: Indicates what kind of data is contained within the file. Valid values are:
    - `captions`: Indicates that the data is a transcription or translation of the content (e.g., closed captioning).
    - `chapters`: Indicates that the data is a set of chapter titles or other sectional information used when navigating the content.
    - `descriptions`: The data is a description of the video or audio content, suitable for people who are blind (in the case of video) or deaf (in the case of audio).
    - `metadata`: The data is meant to be used by scripts and not

shown directly to the user.

    - `subtitles`: Subtitles are additional content for the parent content, such as scene information, extra narrative background, and so forth. If this `kind` is specified, the `srclang` attribute must also be specified for the content.

- `label`: A user-readable label for the track that can be presented when the user is browsing available tracks.
- `src`: The URI for the content.
- `srclang`: The language of the track data, in a BCP 47 language tag (see the BCP 47 standard at http://tools.ietf.org/html/bcp47). If the `kind` is set to subtitles, this attribute must be specified.

Table 7-17. *Standards for the* `track` *Element*

Specification	Status	URL
IETF	Best Current Practice	http://tools.ietf.org/html/bcp47
W3C	Editor's Draft	http://dev.w3.org/html5/webvtt/
W3C	Language Reference	www.w3.org/html/wg/drafts/html/master/embedded-content.html#the-track-element
WHATWG	Living Standard	www.whatwg.org/specs/web-apps/current-work/multipage/embedded-content.html#the-track-element

The `video` Element

The `video` element is used to embed video content (typically video files) in web pages.

In the past, embedding video content in documents typically required the use of a plug-in (most typically Flash). This had the benefit of being fairly ubiquitous because as long as the target browser had the plug-in installed, it would be able to play the content. All of the complexity around the user interface controls, handling different file formats, and special features like dynamic streaming were all handled by the plug-in software.

When implementing the ability to embed video content, web browser manufacturers had to handle these issues themselves. As a result, the appearance and functionality of

the user interface controls for the video player vary from browser to browser.

Each browser also supports different file formats due to patent encumbrances, and some browsers support different file formats on different operating systems depending on locally installed software. An in-depth discussion of video file formats, their patent issues, and operating system support is beyond the scope of this book, but you can find a great deal of information on the Web. Specifically:

- `https://developer.mozilla.org/en-US/docs/Web/HTML/Supported_media_formats` The Mozilla Developer Network has a good page that discusses the various video formats and their support in major browsers.
- `blog.zencoder.com/2013/09/13/what-formats-do-i-need-for-html5-video/` Zencoder is a cloud-based transcoding service provider. Since their bread and butter is video transcoding, they have a good understanding of what formats are necessary for various levels of support.
- `www.jwplayer.com/html5/formats/` The JW Player is a proprietary audio/video player based on HTML5 technology (the core of the player is open source). The company has an obvious interest in the state of HTML5 audio and video support, and they maintain their own statistics on the topic.

There are other sources available on the Web, but many of them seem to be out of date (or it wasn't possible to verify when they were last updated).

Usage

The element is used to embed video content in documents. The content can be specified using either the `src` attribute or by using `source` elements contained within the `video` element. For details on using source elements, see the `source` Element section in this chapter.

The element can also contain zero or more `track` elements to specify time-based data for the video content (such as captions). For details on using track elements, see the `track` Element section in this chapter.

Additionally, the element can optionally contain other elements that will be rendered if the browser does not support the audio element.

The `<video>` tag is not self-closing so both the start and end tags are required.

Syntax

```
<video></video>
```

Listing 7-19. The `video` Element

```
<!DOCTYPE html>
<html>
    <head>
       <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
      <p>Basic</p>
      <video controls="true"
src="../media/lotr_thetwotowers.mp4">
       <p>Your browser does not support the HTML5 video
tag.</p>
     </video>
     <p>Using source Elements</p>
     <video controls="true">
       <source src="../media/video-1.mp4"
type="video/mp4">
       <source src="../media/video-1.ogv"
type="video/ogg">
       <p>Your browser does not support the HTML5 video
tag.</p>
     </video>
    </body>
</html>
```

Properties

The video element supports the following properties:

- `autoplay`: This is a boolean flag that when set (to anything, even `false`) will cause the browser to immediately begin playing the video content as soon as it can without stopping for buffering.
- `controls`: If this attribute is set, then the browser will display its default user interface controls for the video player (volume controls, progress meter/scrub bar, etc.).

- `height`: This attribute can be used to specify the height, in pixels, of the video player.
- `loop`: If this attribute is set, the browser will loop playback of the specified file.
- `muted`: This attribute specifies that the playback should be muted by default.
- `poster`: This attribute can be used to specify a URL to a poster to display before the video is played. If no poster is specified, then the player will show the first frame of the video by default, once it has loaded.
- `preload`: This attribute is used to provide to the browser a hint for how to provide the best user experience for the specified content. It can take the following values:
    - `none`: specifies that the author wants to minimize the download of the video content, possibly because the content is optional, or because the server resources are limited.
    - `metadata`: specifies that the author recommends downloading the metadata for the video content (duration, track list, tags, etc.) and possibly the first few frames of the content.
    - `auto`: specifies that the browser can put the user's needs first without risk to the server. This means the browser can begin buffering the content, download all the metadata, etc.
- `src`: This attribute specifies the source of the content. If desired, this attribute can be omitted in favor of one or more `<source>` tags contained within the `<video>` tag.
- `width`: This attribute can be used to specify the width of the video player, in pixels.

In addition, the `video` element supports the following global attributes:

- `accesskey`
- `class`
- `classlist`
- `contenteditable`

- contextmenu
- dataset
- dir
- draggable
- dropzone
- hidden
- id
- lang
- spellcheck
- style
- tabindex
- title

Table 7-18. *Standards for the* `video` *Element*

Specification	Status	URL
W3C	Candidate Recommendation	www.w3.org/TR/html5/embedded-content-0.html#the-video-element
WHATWG	Living Standard	www.whatwg.org/specs/web-apps/current-work/multipage/semantics.html#the-video-element

Interactive Elements

The `details` and `summary` Elements

The `details` and `summary` elements are used to provide a basic solution for progressive disclosure. By clicking on the contents of a `summary` element, the content within the associated `details` element is shown (or hidden). This particular kind of user interface widget is often referred to as an "expando" and is a common UI component across both web-based and native applications.

There is currently no support for these elements in Internet Explorer, IE Mobile (see http://status.modern.ie/detailssummary), Firefox, or Firefox for

Android (see https://bugzilla.mozilla.org/show_bug.cgi?id=591737). However, the IE team is considering implementation, and the Firefox team is actively developing the feature. Otherwise the feature is well supported in Chrome, Chrome for Android, Android Browser, Safari, and Safari Mobile.

Usage

The tags are meant to be used together. The `details` tag is the parent tag, with the `summary` tag as the first element. Content that appears after the `summary` tag (but still inside the `details` tag) will be shown or hidden as the user clicks on the contents of the `summary` tag.

Both tags are rendered as block elements. The tags are not self-closing, so the closing tag is required for both elements.

Syntax

```
<details>
    <summary>...</summary>
    ...
</details>
```

Listing 7-20. The `details` and `summary` Elements

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
  </head>
  <body>
    <article>
      <h1>Using the &lt;summary&gt; and &lt;details&gt;
tags</h1>
      <details>
        <summary>Item 1</summary>
        <p>Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Phasellus
          accumsan orci nec justo rhoncus facilisis.
Integer pellentesque
          ipsum vitae semper lacinia. Quisque non nisl
rutrum, porta est at,
```

```
        ultrices neque. Aenean consequat, lacus vulputate
vestibulum
        faucibus, turpis magna mollis quam, a congue
neque lorem at
        justo.</p>
      </details>
      <details>
        <summary>Item 2</summary>
        <p>Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Phasellus
        accumsan orci nec justo rhoncus facilisis.
Integer pellentesque
        ipsum vitae semper lacinia. Quisque non nisl
rutrum, porta est at,
        ultrices neque. Aenean consequat, lacus vulputate
vestibulum
        faucibus, turpis magna mollis quam, a congue
neque lorem at
        justo.</p>
      </details>
      <details open>
        <summary>Item 3--this one will be open by
default</summary>
        <p>Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Phasellus
        accumsan orci nec justo rhoncus facilisis.
Integer pellentesque
        ipsum vitae semper lacinia. Quisque non nisl
rutrum, porta est at,
        ultrices neque. Aenean consequat, lacus vulputate
vestibulum
        faucibus, turpis magna mollis quam, a congue
neque lorem at
        justo.</p>
      </details>
      <details>
        <summary>Item 3</summary>
        <p>Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Phasellus
        accumsan orci nec justo rhoncus facilisis.
Integer pellentesque
        ipsum vitae semper lacinia. Quisque non nisl
```

```
rutrum, porta est at,
         ultrices neque. Aenean consequat, lacus vulputate
vestibulum
         faucibus, turpis magna mollis quam, a congue
neque lorem at
         justo.</p>
      </details>
    </article>
  </body>
</html>
```

Properties

The details element has an `open` attribute, which when present (even when set to false) will cause the associated content to be visible by default. Otherwise, both elements support the following standard global attributes:

- `accesskey`
- `class`
- `classlist`
- `contenteditable`
- `contextmenu`
- `dataset`
- `dir`
- `draggable`
- `dropzone`
- `hidden`
- `id`
- `lang`
- `spellcheck`
- `style`
- `tabindex`
- `title`

Table 7-19. *Standards for the* `details` *and* `summary` *Elements*

Specification	Status	URL
W3C	Candidate Recommendation	www.w3.org/html/wg/drafts/html/master/interactive-elements.html#the-details-element www.w3.org/html/wg/drafts/html/master/interactive-elements.html#the-summary-element
WHATWG	Living Standard	www.whatwg.org/specs/web-apps/current-work/multipage/forms.html#the-details-element www.whatwg.org/specs/web-apps/current-work/multipage/forms.html#the-summary-element

Form Elements

The `datalist` Element

The `datalist` element provides a way of associating a list of data with a standard input element. As the user begins to type in the input field, the list appears beneath, and as the user continues to type the choices narrow. At any time the user can use the arrow keys to select an item from the list.

This sort of input field is typically referred to as a "combobox," and is a common user interface element found in web and native applications.

Note that support for this feature is currently quite limited. Internet Explorer lists it as "shipped" but there are significant bugs in the implementation (see http://playground.onereason.eu/2013/04/ie10s-lousy-support-for-datalists/ for a discussion and example). Safari currently does not support the feature either on desktop or mobile.

Usage

The `datalist` element provides a way of associating a filterable list of default items for an input field. As the user types in the field, the list appears and narrows in choices to those that match the characters that have been entered. The user can select an item from the list at any time, or keep typing to enter a custom option.

The `datalist` element takes as children a set of `option` elements, rather like the form `select` element. In browsers that support the element, a `datalist` element is not rendered in the document and can appear anywhere in the markup. To associate a

given `datalist` with a particular input field, set the input's list attribute to match the `id` attribute of the desired `datalist` element.

Syntax

```
<datalist id="example">
    <option value="val1">
    <option value="val2">
    ...
</datalist>
<input list="example" />
```

Listing 7-21. The `datalist` Element

```
<!DOCTYPE html>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
  </head>
  <body>
    <!-- Note the datalist can be anywhere -->
    <datalist id="browsers">
      <option value="Chrome">
      <option value="Firefox">
      <option value="Internet Explorer">
      <option value="Opera">
      <option value="Safari">
    </datalist>
    <article>
      <h1>Using the &lt;datalist&gt; tag</h1>
      <input list="browsers" />
    </article>
  </body>
</html>
```

Properties

The `datalist` element supports the following properties:

- `accesskey`

- class
- classlist
- contenteditable
- contextmenu
- dataset
- dir
- draggable
- dropzone
- hidden
- id
- lang
- spellcheck
- style
- tabindex
- title

Table 7-20. *Standards for the* `datalist` *Element*

Specification	Status	URL
W3C	Candidate Recommendation	www.w3.org/html/wg/drafts/html/master/forms.html#the datalist-element
WHATWG	Living Standard	www.whatwg.org/specs/web-apps/current-work/multipage/forms.html#the-datalist-element

The `meter` Element

The `meter` element provides a visual meter or gauge as a filled bar on the page. The bar is meant to model a measurement with a known range, or a fraction of a known range (e.g., disk usage or volume loudness). It should not be used to show progress; for that use the `progress` element.

Usage

The meter element provides a way of modeling a measurement, and as such has attributes that allow you to define the current value as well as minimum and maximum values and even ranges. The appearance of the bar will vary depending on these settings.

The <meter> tag is not self-closing and the closing tag is required. The <meter> tag can contain other content that will be rendered if the browser does not support the <meter> tag.

Syntax

```
<meter></meter>
```

Listing 7-22. The meter Element

```
<!DOCTYPE html>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
  </head>
  <body>
    <article>
      <h1>Using the &lt;meter&gt; tag</h1>
      <p>Simple meter from 1 to 100, value set to 25:<br>
        <meter min="1" max="100" value="25"></meter>
      </p>
      <p>Simple meter from 1 to 100, low range from 1 to
25, high range from
         75 to 100, value set to 90:<br>
        <meter min="1" max="100" low="25" high="75"
value="90"></meter>
      </p>
      <p>Simple meter from 1 to 100, low range from 1 to
25, high range from
         75 to 100, value set to 10:<br>
        <meter min="1" max="100" low="25" high="75"
value="10"></meter>
      </p>
      <p>Simple meter from 1 to 100, low range from 1 to
25, high range from
         75 to 100, optimum set to 10, value set to 10:<br>
        <meter min="1" max="100" low="25" high="75"
```

```
optimum="10" value="10"></meter>
     </p>
     <p>Simple meter from 1 to 100, low range from 1 to
25, high range from
        75 to 100, optimum set to 10, value set to 10:<br>
       <meter min="1" max="100" low="25" high="75"
optimum="10" value="90"></meter>
     </p>
   </article>
  </body>
</html>
```

Attributes

The `meter` element supports the following attributes:

- `value`: The current value to be displayed. This value must be within the `min` and `max` values, if specified. If no `value` is set, or if it is malformed, the browser will default to 0. If specified but the value is greater than the `max` attribute, the value will be set to the value of the `max` attribute. And if the value is less than the min attribute, the value will be set to the value of the `min` attribute.
- `min`: The minimum value of the range. Defaults to 0 if not specified.
- `max`: The maximum value of the range. Must be greater than the value of the `min` attribute (if specified). Defaults to 1.
- `low`: The highest value of the low range. When the value attribute is within the low range, the bar will render yellow by default.
- `high`: The lowest value of the high range, which ranges from this value to the value of the `max` attribute. When the value attribute is within the high range, the bar will render yellow by default.
- `optimum`: Indicates an optimum value for the range. The value must be between the `min` and `max` values of the range. If the `low` and `high` ranges are used, specifying an `optimum` value within one of them will indicate which of those ranges is preferred. When the value is within the preferred range, the bar will render green. When it is in the other range, it will render red.

In addition, the meter element supports the following global attributes:

- accesskey
- class
- classlist
- contenteditable
- contextmenu
- dataset
- dir
- draggable
- dropzone
- hidden
- id
- lang
- spellcheck
- style
- tabindex
- title

Table 7-21. *Standards for the* `meter` *Element*

Specification	Status	URL
W3C	Candidate Recommendation	www.w3.org/html/wg/drafts/html/master/forms.html#the meter-element
WHATWG	Living Standard	www.whatwg.org/specs/web-apps/current-work/multipage/forms.html#the-meter-element

The `output` Element

The `output` element provides a way of indicating the output of a calculation done as part of a form (e.g., an interest calculation). The element has no special capabilities and is merely a way of semantically indicating the output of a calculation.

Usage

The element is rendered as an inline element by default. It is not self-closing and the closing tag is required.

Syntax

```
<output>...</output>
```

Listing 7-23. The `output` Element

```
<!DOCTYPE html>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
  </head>
  <body>
    <input name="operand1" id="operand1" /> +
    <input name="operand2" id="operand2" /> =
    <output></output><br>
    <button>Add</button>
    <script>
// Get references to the elements we will need.
var myOutput = document.querySelector('output');
var in1 = document.getElementById('operand1')
var in2 = document.getElementById('operand2')
var myButton = document.querySelector('button');

// Add a click event handler to the button that adds the
contents of the two
// fields. We'll use parseFloat to cast the value to
a number; experiment by
// entering various values including numbers, characters,
and combinations of
// characters and numbers. Especially try combinations
that start with numbers.
myButton.addEventListener('click', function() {
  myOutput.innerText = parseFloat(in1.value)
+ parseFloat(in2.value);
}, false);
    </script>
```

```
  </body>
</html>
```

Properties

The `output` element supports the following properties:

- `accesskey`
- `class`
- `classlist`
- `contenteditable`
- `contextmenu`
- `dataset`
- `dir`
- `draggable`
- `dropzone`
- `hidden`
- `id`
- `lang`
- `spellcheck`
- `style`
- `tabindex`
- `title`

Table 7-22. *Standards for the* `output` *Element*

Specification	Status	URL
W3C	Candidate Recommendation	www.w3.org/html/wg/drafts/html/master/forms.html#the-output-element
WHATWG	Living Standard	www.whatwg.org/specs/web-apps/current-work/multipage/forms.html#the-output-element

The `progress` Element

The `progress` element is used to provide a progress meter on the page. It is used to indicate progression or completion of a task, and provides the user with an idea of how much has been done and what still remains. It should not be used for visualizing a measurement within a known range; for that use the `meter` element.

Usage

The `progress` element provides a way of modeling the completion of an ongoing process, and as such has attributes that allow you to define the current value as well as a maximum value. The appearance of the bar will vary depending on these settings.

The `<progress>` tag is not self-closing and the closing tag is required. The `<progress>` tag can contain other content that will be rendered if the browser does not support the `<progress>` tag.

Syntax

```
<progress></progress>
```

Listing 7-24. The `progress` Element

```
<!DOCTYPE html>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
  </head>
  <body>
    <article>
      <h1>Using the &lt;progress&gt; tag</h1>
      <p>Downloading file1<br>
        <progress max="100" value="10">10/100</progress>
10%</p>
    </article>
  </body>
```

Properties

The `progress` element supports the following properties:

- `max`: The maximum value of the activity. This value must be a valid positive floating point number. If `max` is not specified, the maximum value defaults to 1.
- `value`: The current value of the progress. This value must be a valid floating point number between 0 and `max` (if specified) or 1 (if `max` is not specified). If `value` is not specified, then the progress bar is considered indeterminate, meaning the activity it is modeling is ongoing but gives no indication of how much longer it will take to complete.

In addition, the `progress` element supports the standard global properties:

- `accesskey`
- `class`
- `classlist`
- `contenteditable`
- `contextmenu`
- `dataset`
- `dir`
- `draggable`
- `dropzone`
- `hidden`
- `id`
- `lang`
- `spellcheck`
- `style`
- `tabindex`
- `title`

Table 7-23. *Standards for the progress Element*

Specification	Status	URL

W3C	Candidate Recommendation	www.w3.org/html/wg/drafts/html/master/forms.html#the progress-element
WHATWG	Living Standard	www.whatwg.org/specs/web-apps/current-work/multipage/forms.html#the-progress-element

CHAPTER 8

HTML5 API Reference

This chapter provides a detailed reference for all of the new HTML5 JavaScript APIs. For a detailed discussion of these APIs, including examples and support level at time of press, see Chapter 3.

Server-sent Events

The Server-sent Events API enables an HTML5 client to subscribe to an event service published by a server. The server can then transmit events to the HTML5 client.

The API for server-sent events is an `EventSource` constructor in the global JavaScript scope. `EventSource` objects implement the `EventTarget` interface, similar to DOM elements (so events can be published on them, and event handlers registered on them). When a new `EventSource` is instantiated, a URL for an event service is specified. This instructs the browser to establish a connection to the specified URL and begin polling it regularly for new events. When an event is received, the `EventSource` object will publish an event containing the data that was transmitted.

The server can publish events to the service by providing a standard HTTP response to a polling query using the `text/event-stream` MIME type (if that MIME type is not used, the `EventSource` object associated with the service will publish an error event).

The JavaScript API for the `EventSource` constructor is:

```
constructor EventSource(DOMString url)
interface EventSource implements EventTarget: {
  readonly DOMString url;
  readonly unsigned short readyState;
  EventHandler onopen;
  EventHandler onmessage;
```

```
  EventHandler onerror;
  void close();
}
```

Syntax

```
var myEventSource = new
EventSource('http://www.example.com:8030/event-stream/');
```

The `EventSource` constructor takes a single parameter of a valid URL indicating an event service. The resulting interface has the following properties:

- `url`: The URL for the service.
- `readyState`: The current ready state of the interface in the form of an integer:
  - 0: Connecting to the service.
  - 1: Connected to the service and actively listening for events.
  - 2: Closed (as in after the `close` method is called, or a fatal error has occurred in the connection).
- `onopen`: The `open` event interface.
- `onmessage`: The `message` event interface.
- `onerror`: The `error` event interface.
- `close()`: The `close` method. Calling this method will close the connection to the service.

Listing 8-1 shows a basic example of using the `EventSource` constructor (note that you will need to run this example from a server, rather than just loading it directly into the browser).

Listing 8-1. Using the `EventSource` Constructor

```
<!DOCTYPE html>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
  </head>
  <body>
```

```
    <h1>Server-sent Events Reference</h1>
    <script>
/**
 * Handles message events published by the EventSource.
 * @param {EventSourceEvent} event
 */
function handleMessage(event) {
  // Handle message.
  console.log('A message was sent from the server: ',
event.data);
}

/**
 * Handles error events published by the EventSource.
 * @param {EventSourceEvent} event
 */
function handleError(event) {
  // Handle an error.
  console.error('An error happened on the EventSource: ',
event.data);
}

/**
 * Handles an open event published by the EventSource.
 * @param {EventSourceEvent} event
 */
function handleOpen(event) {
  // Handle the open event.
  console.log('The connection is now open.');
}

// Create a new connection to the server.
var targetUrl = 'http://www.service.com/my-event-service';
var myEventSource = new EventSource(targetUrl);

// Attach event handlers. Here we are using the
addEventListener method.
// You could also directly attach the event handlers using
the event interfaces,
// e.g. myEventSource.onmessage = handleMessage.
myEventSource.addEventListener('message', handleMessage);
myEventSource.addEventListener('error', handleError);
myEventSource.addEventListener('open', handleOpen);
```

```
    </script>
  </body>
</html>
```

An event sent from the server takes the form of a simple HTTP response sent with the text/event-stream MIME type. Events consist of multiline `key: value` pairs, and are terminated by a double line feed. Valid keys are as follows:

- `data`: Specifies a line of arbitrary data to be sent to the client, which will receive it as the data property of the `event` object. Terminating a `data` value with a double line feed (`'\n\n'`) signifies the end of a particular event. Multiple `data` values are permitted in a single event; just terminate each one with a single line feed (`'\n'`) and the last with a double line feed.
- `event`: Specifies an arbitrary event type associated with this server-sent event. This will cause an event of the same name to be dispatched from the associated `EventTarget` object, thus enabling arbitrary events beyond `open`, `message`, and `error` to be fired from the server. If no event type is specified, the event will just trigger a `message` event on the `EventTarget`.
- `id`: Specifies an arbitrary ID to associate with the event sequence. Setting an ID on an event stream enables the browser to keep track of the last event fired, and if the connection is dropped it will send a `last-event-ID` HTTP header to the server.
- `retry`: Specifies the number of milliseconds before the browser should requery the server for the next event. By default this is set to 3000 (three seconds). This enables the server resource to throttle browser queries and prevent itself from being swamped.

Any arbitrary text can be transmitted as a server-sent event: HTML, CSS, XML, JSON, and so on. A single response can contain multiple events, and a given event can contain multiple data attributes. For example:

```
event: watch\n
data: {\n
data: "type":"flash flood",\n
data: "counties":"['Jefferson', 'Arapahoe', 'Douglas',
'Broomfield']",\n
data: "from":"12:30 pm June 12, 2015",\n
data: "to":"7:00 am June 13, 2015",\n
```

```
data: "details":"The National Weather Service has issued
a flash flood watch."\n
data: }\n
event: warning\n
data: {\n
data: "type":"severe thunderstorm",\n
data: "counties":"['Jefferson']",\n
data: "from":"12:30 pm June 12, 2015",\n
data: "to":"1:00 pm June 12, 2015",\n
data: "details":"The National Weather Service has issued
a severe thunderstorm warning."\n
data: }\n\n
```

This single server-sent event would trigger both a `watch` event and a `warning` event on the associated `EventTarget` object. The data for the `watch` event would be the JSON-formatted text:

```
{
  "type":"flash flood",
  "counties":"['Jefferson', 'Arapahoe', 'Douglas',
'Broomfield']",
  "from":"12:30 pm June 12, 2015",
  "to":"7:00 am June 13, 2015",
  "details":"The National Weather Service has issued
a flash flood watch."\n
}
```

And the data for the `warning` event would be the JSON-formatted text:

```
{
  "type":" severe thunderstorm ",
  "counties":"['Jefferson']",
  "from":"12:30 pm June 12, 2015",
  "to":"1:00 pm June 12, 2015",
  "details":" The National Weather Service has issued
a severe thunderstorm warning."\n
}
```

Table 8-1. *Standards for Server-sent Events*

Specification	Status	URL
W3C	Draft Living	http://dev.w3.org/html5/eventsource/

WebSockets

The WebSockets API provides a way of full duplex communication between client and server through a maintained network connection.

The API for WebSockets is a `WebSocket` constructor in the global JavaScript scope. `WebSocket` objects implement the `EventSource` interface, similar to DOM elements (meaning that events can be dispatched on them, and event handlers registered on them). The constructor requires a URL (the protocol for which must be either `ws://` or `wss://`), and may also take an optional protocol parameter. The protocol is either a string or an array of strings, each string representing the name of a protocol.

When a new `WebSocket` is instantiated, the client immediately sends a standard HTTP 1.1 `GET` request to the server, and the server then upgrades the connection from HTTP to the WebSocket network protocol. The connection is then ready to send and receive data.

`Strings`, `Blobs`, and `ArrayBuffers` may all be transmitted through the socket. Communication from the server dispatches events on the `EventTarget` interface. Communication to the server is done via the `send` method of the `WebSocket` object.

The definition of the API is:

```
constructor WebSocket(DOMString url, optional (DOMString
or DOMString[]) protocols)
interface WebSocket implements EventTarget {
  readonly DOMString url;
  readonly unsigned short readyState;
  readonly unsigned long bufferedAmount;
  EventHandler onopen;
  EventHandler onerror;
  EventHandler onclose;
  readonly DOMString extensions;
  readonly DOMString protocol;
  void close(optional unsigned short code, optional
USVString reason);
  EventHandler onmessage;
  BinaryType binaryType;
  void send(USVString|Blob|ArrayBuffer data);
};
```

Syntax

```
// Create a web socket without specifying protocols.
var myWebSocket = new WebSocket('ws://www.example.com/');

// Create a web socket and specify one or more protocols.
var myChatWebSocket = new
WebSocket('ws://www.example.com/', chat');
var myWebSocket = new WebSocket('ws://www.example.com/',
['chat', 'json']);
```

The properties of the interface are:

- `url`: The URL of the service, set when the `WebSocket` object was constructed.
- `readyState`: An integer value representing the communication state of the connection:
    - 0: The client is still in the process of connecting to the service.
    - 1: The connection is open and ready to use.
    - 2: The connection is closing.
    - 3: The connection is closed and no longer active.
- `bufferedAmount`: The number of bytes that are queued for sending back to the server, but haven't been sent yet.
- `onopen`: The open event interface.
- `onerror`: The error event interface.
- `onclose`: The close event interface.
- `onmessage`: The message event interface.
- `extensions`: The name of any file extensions in use by the server (e.g., zip)
- `protocol`: The name of the protocol that is in use.
- `binaryType`: What type of data is being transmitted (e.g. `'blob'` or `'arraybuffer'`).
- `send`: The `send` method, used for transmitting data back to the server. Takes one parameter, which is the data to be sent.

- `close`: The `close` method, which closes the connection. Can take two optional parameters, which are typically defined by the protocol in use:
    - `code`: An optional number representing the closing code.
    - `reason`: A string containing the reason for closing the connection.

Listing 8-2 shows a simple implementation of a WebSocket, including stubbed event handlers (note that you will need to run this example from a server).

Listing 8-2. Using the `WebSocket` Constructor

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
  </head>
  <body>
    <h1>Web Sockets Reference</h1>
    <script>
// Create a new web socket connection.
var socketUrl = 'ws://www.fgjkjk4994sdjk.com/';
var validProtocols = ['chat', 'json'];
var myWebSocket = new WebSocket(socketUrl,
validProtocols);

/**
 * Handles an error event on the web socket object.
 */
function handleError() {
  console.log('An error occurred on the web socket.');
}

/**
 * Handles a close event on the web socket object.
 * @param {CloseEvent} event The close event object.
 */
function handleClose(event) {
  console.log('The web socket connection was closed
because ', event.reason);
}
```

```
/**
 * Handles an open event on the web socket object.
 * @param {OpenEvent} event The open event object.
 */
function handleOpen(event) {
  console.log('The web socket connection is open.');
}

/**
 * Handles a message event on the web socket object.
 * @param {MessageEvent} event The message event object.
 */
function handleMessage(event) {
  console.log('A message event has been sent.');

  // The event object contains the data that was
transmitted from the server.
  // That data is encoded either using the chat protocol
or the json protocol,
  // so we need to deterine which protocol is being used.
  if (myWebSocket.protocol === validProtocols[0]) {
    console.log('The chat protocol is active.');
    console.log('The data the server transmitted is: ',
event.data);
    // etc...
  } else {
    console.log('The json protocol is active.');
    console.log('The data the server transmitted is: ',
event.data);
    // etc...
  }
}

// Register the event handlers on the web socket.
myWebSocket.addEventListener('error', handleError);
myWebSocket.addEventListener('close', handleClose);
myWebSocket.addEventListener('open', handleOpen);
myWebSocket.addEventListener('message', handleMessage);
    </script>
  </body>
</html>
```

Tip Building a WebSocket server from scratch is a complex task. There are, however, several open source web socket servers available that you can use in your projects. If you would like to tackle building one from scratch, see Sections 4, 5, and 6 in the WebSocket Protocol RFC.

Table 8-2. *Standards for WebSockets*

Specification	Status	URL
W3C	Candidate Recommendation	www.w3.org/TR/websockets/
WHATWG	Living Standard	https://html.spec.whatwg.org/multipage/comms.html#ne
RFC	Complete	https://tools.ietf.org/html/rfc6455

Cross-Document Messaging/Web Messaging

Browsers will allow you to open documents from different origins in iframes, but if a script from one origin attempts to interact with the content from another origin, the browser will throw an error. The Cross-Document Messaging API (also known as Web Messaging) defines a secure way for scripts from one origin in one frame to communicate with scripts from another origin in another frame. This allows scripts from multiple origins to more safely interact with one another.

The Cross-Document Messaging specification defines a both a new method and a new event on the `window` object. The new method is `postMessage`, and it takes three parameters:

- `message`: The message you want to transmit from the current context to the target context. The message is serialized using the structured clone algorithm, unless you specify that the objects should instead be transferred using the `transfer` parameter.
- `origin`: The origin you expect the resources in the target context to have. If the resources in the target context do not have the specified origin, the method will have no effect.
- `transfer`: an array of objects that are part of the message that should have their ownership transferred to the new context.

Transferring ownership means that the objects will be bound to the origin of the target context. Transferring ownership is limited to `ArrayBuffer` and `MessagePort` objects.

Note The structured clone algorithm is defined as part of the HTML5 specification. You can read it at `www.w3.org/TR/html5/infrastructure.html#safe-passing-of-structured-data`. Basically this algorithm allows you to transmit just about anything from one context to another. The exceptions are functions, DOM elements, and `Error` objects, which will throw a `DATA_CLONE_ERROR` if you attempt to transmit them.

The new event is the `message` event, which is dispatched on the `window` object when the `postMessage` method is used to transmit a message. The resulting event object will have two important attributes:

- `data`: This attribute will contain the message that was sent from the other context.
- `source`: This attribute will contain the origin of the sending context. You should always double-check the origin of message sources to prevent accidentally capturing and processing events from unexpected (and possibly malicious) origins.

Syntax

```
var targetIframe = document.getElementById('my-iframe');
targetIframe.contentWindow.postMessage('hello world',
'apress.com');

window.addEventListener('message', function(event) {
    if (event.source === 'apress.com') {
        console.log('A message was received: ',
event.data);
    }
});
```

To demonstrate using the API, you need two pages served from different origins: a host page and a target page. The host page will contain an iframe that will load the target page. The host page will dispatch events to the target page, and the target page will listen for message events and alert their contents. Listing 8-3 is the host page.

Listing 8-3. The Host Page

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
  </head>
  <body>
    <h1>Cross-Domain Messaging</h1>
    <iframe id="target-iframe" src="target-page.html">
</iframe>
    <p><button id="clickme">Click to send a message to the
iframe.</button></p>
    <script>
// Create some objects to transfer.
var testBlob = new Blob(['some data']);
var testBuffer = new ArrayBuffer(100);
var testBuffer2 = new ArrayBuffer(8);

// To transfer multiple objects, we need to wrap them in
a single carrier. The
// names of the properties don't matter, they're just
serving as a place to
// store references to the buffer objects.
var transferObject = {
    buffer1: testBuffer,
    buffer2: testBuffer2
};

var targetFrame = document.getElementById('target-
iframe');

// Reference to the button.
var clickme = document.getElementById("clickme");

// Add a click event handler to the button.
clickme.addEventListener("click", function() {
  // Send a simple text string to the target frame.
  targetFrame.contentWindow.postMessage('hello world',
'*');
  // Send a Blob to the target frame.
  targetFrame.contentWindow.postMessage(testBlob, '*');
  // Transfer multiple array buffers to the target frame.
```

```
  targetFrame.contentWindow.postMessage(transferObject,
'*',
      [transferObject.buffer1, transferObject.buffer2]);
});
    </script>
  </body>
</html>
```

Note that the iframe element's `src` is set to load the target page from the same context. If you have access to a different domain (or even another web server on the same domain running on a different port) you can serve the target page from there and thus fully demonstrate that the API allows for sending messages across origins.

Listing 8-4 contains the target page.

Listing 8-4. The Target Page

```
<!DOCTYPE html>
<html>
    <head>
      <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
      <h1>Target iframe</h1>
      <script>
/**
 * Handles a message event on the window object.
 * @param {MessageEvent} event A message event object.
 */
function handleMessage(event) {
  // Create a string for alerting.
  var strAlert = "Target iframe:\n";

  if (event.data.buffer1) {
    // The two buffers have been transferred.
    strAlert += event.data.buffer1 + '\n';
    strAlert += event.data.buffer2 + '\n';
  } else {
    // Just alert the data.
    strAlert += event.data;
  }
  alert(strAlert);
}
```

```
// Register the event handler.
window.addEventListener("message", handleMessage, false);
     </script>
    </body>
</html>
```

To run the example, click the button. The host page will send three messages to the target page, resulting in three alerts.

Table 8-3. *Standards for Cross-Document Messaging*

Specification	Status	URL
W3C	Candidate Recommendation	www.w3.org/TR/webmessaging/
WHATWG	Living Standard	www.whatwg.org/specs/web-apps/current-work/multipage/web-messaging.html#crossDocumentMessages

Web Storage

The new Web Storage API specifies a new way to store information on the client. Before Web Storage, the standard way of storing information on the client was with HTTP cookies, which was messy and inconvenient. Web Storage provides an easier-to-use storage feature.

Web Storage defines two new interface objects in the global context: `sessionStorage` and `localStorage`. The `sessionStorage` interface is for storing data for a single browsing session. When the user closes their browser, the data will automatically be deleted. The `localStorage` interface is for storing data across sessions. Even if the user closes their browser, the data stored in localStorage will persist.

Caution All browsers have implemented some form of "private browsing." When using this feature, `localStorage` data is removed when the user ends the session. In addition, many browsers now have features that will automatically clear `localStorage` when the browser is closed even for regular sessions. Your application should not assume that any data stored in `localStorage` will always be available, and should respond appropriately if the expected data is not present.

The API also defines the `storage` event, which is dispatched on the `window` object of all documents that are the same as the document where the storage change occurred, but not the window document where the change occurred. For other DOM events, if you have a page loaded and an event is dispatched, it is dispatched on the current page. The `storage` event does not dispatch on the current page. If you have multiple versions of the same page open in tabs, the event will dispatch on every window object except the one that is currently active.

Caution Currently Internet Explorer dispatches the storage event in all documents, not just inactive ones. There is a bug filed for the behavior at https://connect.microsoft.com/IE/feedback/details/774798/localstorage-event-fired-in-source-window that is currently postponed.

The API definition is:

```
interface Storage {
  readonly unsigned long length;
  DOMString? key(unsigned long index);
  getter DOMString? getItem(DOMString key);
  setter creator void setItem(DOMString key, DOMString
value);
  deleter void removeItem(DOMString key);
  void clear();
};
interface WindowSessionStorage {
  readonly attribute Storage sessionStorage;
};
interface WindowLocalStorage {
  readonly attribute Storage localStorage;
};
```

Both `localStorage` and `sessionStorage` implement the `Storage` interface, and thus have the same methods:

- `getItem(key)`: Returns the data associated with the specified key.
- `removeItem(key)`: Removes the data associated with the specified key.
- `setItem(key, data)`: Stores the data in storage with the

specified key.

- `clear()`: Clears the storage of all contents.

Whenever `localStorage` is changed using any of those methods, a `storage` event is dispatched on the `window` object of any document that is the same as the current document. The associated `event` object is a `StorageEvent` object, and it has the following properties:

- `target`: The `target` property is a reference to the DOM element on which the event was dispatched. In this case, that is the `window` object.
- `type`: The `type` property is set to `storage`.
- `key`: The `key` property contains the key that had its associated data changed.
- `oldValue`: The `oldValue` property contains the previous value of the data.
- `newValue`: The `newValue` property contains the new value of the data.
- `url`: The `url` property contains the URL of the hosting document.
- `storageArea`: The `storageArea` property will be a reference to the actual `localStorage` object.

Listing 8-5 demonstrates using Web Storage.

Listing 8-5. Using Web Storage

```
<!DOCTYPE html>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
  </head>
  <body>
    <h1>Web Storage Example</h1>
    <script>

/**
 * Handles a storage event.
 * @param {StorageEvent} event The storage event object.
```

```
 */
function handleStorageEvent(event) {
  var alertMsg = 'Storage event!\n';
  alertMsg += 'key: ' + event.key + '\n';
  alertMsg += 'oldValue: ' + event.oldValue + '\n';
  alertMsg += 'newValue: ' + event.newValue + '\n';
  alert(alertMsg);
}

// Register the event handler on the window object.
window.addEventListener('storage', handleStorageEvent,
false);

// Check to see if we've visited this page before.
var myValue = localStorage.getItem('myKey');
if (myValue == null) {
  alert('This is the first time you loaded this page! Now
reload this page.');
  localStorage.setItem('myKey', 'true');
} else {
  alert('You have loaded this page before!');
  localStorage.removeItem('myKey');
}
    </script>
  </body>
</html>
```

The first time you load the example, it will tell you this is the first time you have loaded the page. When you reload, it will detect the stored information and then delete it, thus resetting the test. If you open the example in two tabs you will see the alerts resulting from the storage events being dispatched on the inactive tabs.

Table 8-4. *Standards for Web Storage*

Specification	Status	URL
W3C	Candidate Recommendation	www.w3.org/TR/webstorage/
WHATWG	Living Standard	www.whatwg.org/specs/web-apps/current-work/multipage/webstorage.html

Drag and Drop

The new HTML5 Drag and Drop specification provides a native API for handling drag-and-drop interactions in the browser. The API is event driven, and using it involves these steps:

- Declare one or more objects as `draggable`, and attach desired event handlers.
- Attach `drop` event handlers to target elements.
- As the user drags items and drops them on targets, the various events are dispatched.

Specifying Draggable Elements: The `draggable` Property

The `draggable` property is a new property for DOM elements that indicates the availability of the element for being a drag target. The property can be set to three values:

- `true`: Indicates that the element is draggable.
- `false`: Indicates that the element is not draggable.
- `auto`: The browser's default rules apply. For most elements, the default rule is `false`. (The exception is selected text, which can always initiate a drag interaction.)

Handling the Interactions: Drag-and-Drop Events

The API specifies several new events that occur on either the dragging element or the elements it is dragged over:

- `dragstart`: Dispatched from the element being dragged.
- `dragenter`: Dispatched from any element when a draggable item is dragged into it.
- `dragover`: Dispatched continuously from any element as long as a draggable item is over it. Note that this event fires continuously regardless of whether or not the draggable item is moving.

- `dragleave`: Dispatched from an element when a draggable item leaves its boundary.
- `drag`: Dispatched from the element being dragged throughout the drag sequence. Like `dragover` this event is fired continuously regardless of whether the pointer is being moved.
- `dragend`: Dispatched from the element being dragged when the mouse button is released.
- `drop`: Dispatched from an element when the user drops a draggable item on it by releasing the mouse button.

Specifying Drop Targets

The API specifies a `dropzone` attribute that is supposed to indicate that an element can be a drop target. However, the `dropzone` attribute is not widely supported, so the only way to specify a given element is a valid target is through the event handlers.

Generally speaking, the majority of elements in the DOM should not be valid drop targets, so the default action of the `dragover` event is to cancel drops. As a result, to indicate a valid drop target you have to cancel the default action of the `dragover` event by calling the `preventDefault()` method on the `event` object within the event handler.

The `dataTransfer` Object

All of the drag-and-drop events can be handled with standard event handlers, and those event handlers will receive an `event` object as a parameter. One of the properties on drag-and-drop `event` objects is the `dataTransfer` object. This object is used to control the appearance of the drag-and-drop helper (the ghosted visual element that follows the cursor during the drag-and-drop operation), to indicate what the drag-and-drop process is doing, and to easily transfer data from the `dragstart` event to the `drop` event.

The `dataTransfer` object has the following methods:

- `Event.dataTransfer.addElement(HtmlElement)`: Specify the source element of the drag sequence. This affects where the `drag` and `dragend` events are fired from. This is set automatically at the beginning of the drag interaction, so you probably won't need to change it.

- `Event.dataTransfer.clearData(opt_DataType)`: Clear the data associated with a specific `DataType` (see `setData` in this list). If the `DataType` is not specified, all data is cleared.
- `Event.dataTransfer.getData(DataType)`: Get the data associated with a specific `DataType` (see `setData`, next).
- `Event.dataTransfer.setData(DataType, data)`: Associates the specified `data` with the `DataType`. Valid `DataTypes` depend on the browser. Internet Explorer only supports `DataTypes` of `text` and `url`. Other browsers support standard MIME types and even arbitrary types. The `data` has to be a simple string but could conceivably be a JSON-formatted serialized object. Note that Firefox requires the `dataTransfer` object to be initialized with data during the `dragstart` event in order for drag and drop events to fire correctly.
- `Event.dataTransfer.setDragImage(HtmlElement, opt_offsetX, opt_offsetY)`: Sets the drag helper image to the specified HTML element. By default the upper left corner of the helper image is placed under the mouse pointer, but that can be offset by specifying the optional parameters `opt_offsetX` and `opt_offsetY`, in pixels. This method is not available in Internet Explorer and apparently never will be; see http://connect.microsoft.com/IE/feedback/details/8 datatransfer-prototype-setdragimage-method.

The `dataTransfer` object also has the following properties:

- `Event.dataTransfer.dropEffect`: The drop effect that is being performed by the drag-and-drop sequence. Valid values are `copy`, `move`, `link`, and `none`. This value is automatically initialized in the `dragenter` and `dragover` events based on what interaction the user has requested through a combination of mouse actions and modifier keys (e.g., Ctrl-drag, Shift-drag, Option-drag, etc.). These are platform dependent. Only values specified by `effectAllowed` (see next) will actually initiate drag-and-drop sequences.
- `Event.dataTransfer.effectAllowed`: Specifies which `dropEffects` are permitted for this drag-and-drop sequence. Valid values and the effects they permit are:

        - `copy`: Allow a copy `dropEffect`.
        - `move`: Allow a move `dropEffect`.
        - `link`: Allow a link `dropEffect`.
        - `copyLink`: Allow both a copy and a link `dropEffect`.
        - `copyMove`: Allow both a copy and a move `dropEffect`.
        - `linkMove`: Allow both a link and a move `dropEffect`.
        - `all`: All `dropEffects` are permitted. This is the default value.
        - `none`: No `dropEffects` are permitted (the item cannot be dropped).
    - `Event.dataTransfer.files`: Contains a list of all the files available on the data transfer. Will only have values if files are being dragged from the desktop to the browser.
    - `Event.dataTransfer.types`: Contains a list of all the `DataTypes` that have been added to the `dataTransfer` object, in the order in which they were added.

Listing 8-6 demonstrates the Drag and Drop API.

Listing 8-6. The Drag and Drop API at Work

```
<!DOCTYPE html>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
    <style type="text/css">
#drag-target,
#drop-target {
  float: left;
  padding: 10px;
  margin: 10px;
  box-sizing: border-box;
}
#drag-target {
  background-color: #008000;
```

```
  width:75px;
  height:75px;
}
#drop-target {
  background-color: #0000FF;
  width:150px;
  height:150px;
}
.drag-over {
  border: 5px solid #FF0000;
}
</style>
  </head>
  <body>
    <h1>Drag and Drop Example</h1>
    <div id="drop-target">Target</div>
    <div id="drag-target" draggable="true">Drag me!</div>
    <script>
/**
 * Handles a dragStart event.
 * @param {DragEvent} event The event object.
 */
function handleDragStart(event) {
  // Set the data in the dataTransfer object to the id of
the element being
  // dragged.
  event.dataTransfer.setData("Text",
event.target.getAttribute('id'));
}

/**
 * Handles a dragenter event.
 * @param {DragEvent} event The event object.
 */
function handleDragEnter(event) {
  // Apply a class to the element.
  event.target.classList.add('drag-over');
}

/**
 * Handles a dragleave event.
 * @param {DragEvent} event The event object.
```

```
 */
function handleDragLeave(event) {
  // Remove the class from the element.
  event.target.classList.remove('drag-over');
}

/**
 * Handles a dragover event.
 * @param {DragEvent} event The event object.
 */
function handleDragOver(event) {
  // Indicates this element is a valid drop target.
  event.preventDefault();
}

/**
 * Handles a drop event.
 * @param {DragEvent} event The event object.
 */
function handleDrop(event) {
  // Get a reference to the dragging element and append it
to the drop target.
  var src = event.dataTransfer.getData("Text");
  event.target.appendChild(document.getElementById(src));
  event.preventDefault();
}

// Register event handlers.
var dragTarget = document.getElementById('drag-target');
dragTarget.addEventListener('dragstart', handleDragStart);

var dropTarget = document.getElementById('drop-target');
dropTarget.addEventListener('dragenter', handleDragEnter);
dropTarget.addEventListener('dragleave', handleDragLeave);
dropTarget.addEventListener('dragover', handleDragOver);
dropTarget.addEventListener('drop', handleDrop);
    </script>
  </body>
</html>
```

When you run this example, you'll be able to drag the drag target (labeled "Drag me!") into the drop target (labeled "Target"). It sets the data in the `dataTransfer` object to the ID of the drag target during the `dragstart` event, and then retrieves it

during the `drop` event and uses it to fetch a reference to the element and move it in the DOM. This is a very common use case for the Drag and Drop API.

Table 8-5. *Standards for Drag and Drop*

Specification	Status	URL
W3C	Recommendation	`www.w3.org/TR/html5/editing.html#dnd`
WHATWG	Living Standard	`www.whatwg.org/specs/web-apps/current-work/multipage/dnd.html`

Web Workers

The Web Workers API enables you to create threaded JavaScript applications by creating (or "spawning") subprocesses to handle certain tasks. Each of these workers runs its own JavaScript context and performs whatever tasks you set to it. Web Workers can also spawn other Web Workers.

Communication between Web Worker contexts and the main JavaScript thread is done via a `postMessage` interface similar to that used by Web Messaging. This enables you to pass data into and out of Web Worker contexts, but because all contexts are independent, any data passed between contexts is copied unless you specifically transfer it. (See the "Cross-Document Messaging/Web Messaging" section earlier for more details on sending and transferring data using `postMessage`).

When you create a new Web Worker, you specify a JavaScript file for it to load and run. To start it, send a message to it (any message will do). The worker can post messages back to the parent context or any other Web Workers it has access to.

Web Workers have some important limitations, which are designed to help avoid the usual pitfalls inherent in writing multithreaded applications:

- A Web Worker runs in its own independent JavaScript context. It has no direct access to anything in any of the other execution contexts like other Web Workers, or the main JavaScript thread.
- Communication between Web Worker contexts and the main JavaScript thread is done via a `postMessage` interface similar to that used by Web Messaging. This enables you to pass data into and out of Web Worker contexts, but because all contexts are independent, any data passed between contexts is copied, not shared.
- A Web Worker cannot access the DOM. The only DOM methods

available to a Web Worker are `atob`, `btoa`, `clearInterval`, `clearTimeout`, `dump`, `setInterval`, and `setTimeout`.

- Web Workers are bound by the Same Origin Policy, so you cannot load a worker script from a different origin than the original script.

The Web Workers API takes the form of a new `WebWorker` constructor in the global JavaScript scope:

```
constructor WebWorker(DOMstring url)
interface WebWorker implements EventTarget {
  readonly WorkerLocation location;
  void terminate();
  OnErrorEventHandler onerror;
  EventHandler onlanguagechange;
  EventHandler onoffline;
  EventHandler ononline;
  EventHandler onmessage;
};
```

Syntax

```
var myWorker = new WebWorker('worker-script.js');
```

The constructor returns a `WebWorker` object, which implements the `EventTarget` interface. The properties are:

- `location`: The `location` property is similar to the `document.location` object but contains information specific to the Web Worker (see hereafter for details).
- `terminate()`: The `terminate` method will end the thread for the worker. Once a worker has been terminated it is not possible to restart it.
- `onerror`: The `onerror` event handler is called when an `error` event is dispatched on the worker.
- `onlanguagechange`: The `onlanguagechange` handler is dispatched on the worker when the user changes their preferred language in the browser.
- `onoffline`: The `onoffline` event handler is called when an `offline` event is dispatched on the worker. This occurs when the

browser loses network connectivity and the value of `navigator.onLine` is changed to `false`.

- `ononline`: The `ononline` event handler is called when an `online` event is dispatched on the worker. This occurs when the browser regains network connectivity.
- `onmessage`: The `onmessage` event handler is called when a `message` event is dispatched on the worker.

The execution context inside of a Web Worker is significantly different than the global execution context. Web Workers have no access to the DOM, but they do have access the following properties and methods:

- The DOM methods `atob`, `btoa`, `clearInterval`, `clearTimeout`, `dump`, `setInterval`, and `setTimeout`.
- The `XMLHttpRequest` constructor, so Web Workers can perform asynchronous network tasks.
- The `WebSocket` constructor, so Web Workers can create and manage Web Sockets (as of this writing, Firefox does not enable `WebSocket` for Web Workers; however, this feature is being implemented and you can track its status at https://bugzilla.mozilla.org/show_bug.cgi?id=504553)
- The `Worker` constructor, so Web Workers can spawn their own workers (which are referred to as "subworkers"). As of this writing, Chrome and Safari do not implement the `Worker` constructor for Web Workers. There is a bug filed for Chrome at https://code.google.com/p/chromium/issues/detail?id=31666 and for Safari's WebKit at https://bugs.webkit.org/show_bug.cgi?id=22723. Internet Explorer does support subworkers as of version 10.
- The `EventSource` constructor, so Web Workers can subscribe to Server-sent Event streams. This appears to be a nonstandard feature, but seems to be available in all major browsers as of this writing.
- A special subset of the `Navigator` properties, available through the `navigator` object:

    - `navigator.language`: Returns the current language the browser is using.
    - `navigator.onLine`: Returns a boolean indicating whether or not the browser is online.
    - `navigator.platform`: Returns a string indicating the platform of the host system.
    - `navigator.product`: Returns a string with the name of the current browser.
    - `navigator.userAgent`: Returns the user agent string for the browser.

      The implementation of these properties varies from browser to browser, so it might be better to pass needed `Navigator` information into the Web Worker from the main thread.
- A special subset of `Location` properties, available on the `location` object:
    - `location.href`: The full URL of the script being executed by the Web Worker.
    - `location.protocol`: The protocol scheme of the URL of the script being executed by the Web Worker, including the final ":".
    - `location.host`: The host part of the URL (the hostname and port) of the script being executed by the Web Worker.
    - `location.hostname`: The hostname part of the URL of the script being executed by the Web Worker.
    - `location.port`: The port part of the URL of the script being executed by the Web Worker.
    - `location.pathname`: The initial '/' followed by the path of the script being executed by the Web Worker.
    - `location.search`: The initial '?' followed by the parameters (if any) of the URL of the script being executed by the Web Worker.
    - `location.hash`: The initial '#' followed by the

fragment identifier (if any) of the URL of the script being executed by the Web Worker.

In addition, Web Workers have one special method available only to them: `importScripts`. The method takes either a single URL or a comma-delimited list of URLs of JavaScript files to load and execute in order. The `importScripts` method is a blocking method and is bound by the Same Origin Policy.

Syntax

```
importScripts('test.js');
importScripts('polymer.js', 'custom-element.js',
'jquery.js');
```

When a Web Worker is started, it follows these steps:

- It executes the script from start to finish, including any asynchronous tasks (such as `XMLHttpRequest` calls).
- If part of its execution was to register a `message` event handler, it then goes into a `wait` loop for incoming messages. The first message it receives will be the message that was posted to start the worker. The worker will remain in wait mode until you manually terminate it, or it terminates itself.
- If no `message` event handlers were registered, the worker thread will terminate automatically.

To demonstrate a Web Worker you will need two files: a host page that will create and run the worker, and a stand-alone JavaScript script for the worker to execute. Listing 8-7 shows a basic host page.

Listing 8-7. Creating and Using a Web Worker

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
      <h1>Web Workers</h1>
      <div id="message-box"></div>
```

```
    <script>

/**
 * Handles an error event from web worker.
 * @param {WorkerErrorEvent} event The error event object.
 */
function handleWorkerError(event) {
  console.warn('Error in web worker: ', event.message);
}

/**
 * Handles a message event from a web worker.
 * @param {WorkerMessageEvent} event The message event
object.
 */
function handleWorkerMessage(event) {
  displayMessage('Message received from worker: '
+ event.data);
}

/**
 * Displays a message in the message box.
 * @param {string} message The message to display.
 */
function displayMessage(message) {
  // Get a reference to the target element.
  var messageBox = document.getElementById('message-box');

  // Create a new paragraph and set its content to the
message.
  var newParagraph = document.createElement('p');
  newParagraph.innerHTML = message;

  // Append the new paragraph to the target element.
  messageBox.appendChild(newParagraph);
}

// Create a new worker.
var myNewWorker = new Worker('example8-8.js');

// Register error and message event handlers on the
worker.
myNewWorker.addEventListener('error', handleWorkerError);
```

```
myNewWorker.addEventListener('message',
handleWorkerMessage);

// Start the worker.
myNewWorker.postMessage('begin');
     </script>
    </body>
</html>
```

Listing 8-8 is a very basic stand-alone script for a Web Worker.

Listing 8-8. A Simple Web Worker Script

```
/**
 * Handles a message event from the main context.
 * @param {WorkerMessageEvent} event The message event.
 */
function handleMessageEvent(event) {
  // Do something with the message.
  console.log('Worker received message:', event.data);

  // Send the message back to the main context.
  self.postMessage('Your message was received.');
}

// Register the message event handler.
self.addEventListener('message', handleMessageEvent);

// Dispatch 10 events to the host document.
var counter = 0;
var timer = setInterval(function() {
  counter++;
  self.postMessage('Message #' + counter);
  if (counter == 10) {
    // Stop the timer.
    clearInterval(timer);

    // Throw an error.
    throw new Error();
  }
}, 1000);
```

Note The files for this example will need to be served on a regular server for the code to function. If you simply load the host file in the browser from the filesystem the browser will throw a cross origin violation error.

Table 8-6. *Standards for Web Workers*

Specification	Status	URL
W3C	Recommendation	http://dev.w3.org/html5/workers/
WHATWG	Living Standard	www.whatwg.org/specs/web-apps/current-work/multipage/workers.html

CHAPTER 9

Canvas Reference

This chapter will provide a detailed reference for the `canvas` element and the 2D drawing context API. For detailed discussions about these features and more examples, see Chapter 4.

The `canvas` Element

The HTML5 `canvas` element enables you to draw bitmaps on web pages by providing a blank "canvas" to work on. The `canvas` element itself is a block-level DOM element. To use a `canvas` element for drawing, you have to fetch a drawing context reference from the element. The context exposes an extensive API for drawing that you can use in your scripts.

The API definition for the `canvas` element itself is:

```
Interface HTMLCanvasElement implements HTMLElement {
  unsigned long width;
  unsigned long height;
  renderingContext? getContext(DOMString contextId);
  DOMString toDataURL(optional DOMString type);
}
```

Syntax

```
<canvas id="myCanvas" width="100" height="100"></canvas>

<script>
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');
```

```
</script>
```

The properties are

- `width`: The layout width of the element.
- `height`: The layout height of the element.
- `getContext()`: Returns the requested rendering context. All browsers support the `'2D'` context, and many support the `'webgl'` (in older browsers, `'experimental-webgl'`) context.
- `toDataURL()`: Returns a data URI representation of the canvas bitmap. The optional type parameter is used to specify the format of the encoded data. Valid options are `"image/jpeg"`, `"image/gif"`, or `"image/png"`. If no type parameter is specified, the default is `"image/png"`. The resolution of the encoded image is 96dpi.

Tip The data URI scheme is a way of encoding data directly into a document. You can encode anything in a data URI, but it is most commonly used to encode images. When an image is encoded as a data URI you can then use that data URI where you would use a regular URL (e.g., in the `src` attribute for an image tag). Data URIs are defined in RFC 2397, at `tools.ietf.org/html/rfc2397`.

You need to specify the width and height of a canvas element using the `width` and `height` properties on the tag itself and not with CSS. If you use CSS, the aspect ratio of the drawing context will be incorrect (unless you are specifying the default size of the canvas element, which is 200 pixels high by 400 pixels wide).

Listing 9-1 shows a basic implementation of a `canvas` element.

Listing 9-1. A `canvas` Element

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
canvas {
  border: 1px solid #000;
```

```
}
       </style>
    </head>
    <body>
      <canvas id="myCanvas" width="200" height="200">
</canvas>
     <script>
// Get a DOM reference to the canvas element.
var myCanvas = document.getElementById('myCanvas');

// Get a reference to the 2d drawing context from the
canvas element.
var myContext = myCanvas.getContext('2d');
     </script>
    </body>
</html>
```

Table 9-1. *Standards for the* `canvas` *Element*

Specification	Status	URL
W3C	Candidate Recommendation	www.w3.org/TR/2dcontext/
WHATWG	Living Standard	www.whatwg.org/specs/web-apps/current-work/multipage/the-canvas-element.html

The Drawing Context

Once you have created a `canvas` element and retrieved the drawing context from it, you can use the API on the drawing context to begin drawing. The 2d drawing context is the most commonly used drawing context, and its API is what this chapter will cover. The commands provided by the API are simple but provide all the tools you need to create complex drawings.

The 2d drawing context uses a pen metaphor for drawing, meaning that most commands to draw something take the form of "From the current position of the pen, draw this item" or "Draw this item from this position to the current position of the pen." The 2d context also employs the concept of paths. A *path* is an invisible representation of the item you have just drawn, whether it be a line, a circle, or complex drawing made up of multiple items. Paths can be stroked (meaning the path is drawn as if a pen stroke followed it exactly) or filled (meaning all the area contained by the path is filled). Paths can be stroked and filled with solid colors, gradients, or patterns generated from

images. Paths can be open (different starting and ending points) or closed (same starting and ending points). They need not be continuous; you can have a single path that has several “pieces” that do not connect. The 2d drawing context only supports having one path active at a time.

Defining Paths

The 2d drawing context provides a few simple commands for defining paths.

The `beginPath` Method

This command specifies that you are defining a new path. The previous path will be cleared from the drawing context.

Syntax

```
Context.beginPath();
```

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

myContext.beginPath();
```

The `closePath` Method

This command closes the current path. If the beginning point and the ending point of the current path are not identical (in other words, if the current path isn’t already closed visually), this command will close the path by extending it along a straight line between the two points.

Syntax

```
Context.closePath();
```

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

myContext.closePath();
```

The `moveTo` Method

This command moves the pen to the specified coordinates. This provides a way to create noncontiguous paths.

Syntax

```
Context.moveTo(x, y);
```

Table 9-2. *Parameters for the* `moveTo` *Method*

Parameter	Type	Explanation
x	Number	The x coordinate of the new location.
y	Number	The y coordinate of the new location.

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

myContext.moveTo(10, 10);
```

Listing 9-2 demonstrates creating and managing paths in the drawing context.

Listing 9-2. Managing Paths

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
canvas {
  border: 1px solid #000;
}
        </style>
```

```
    </head>
    <body>
      <canvas id="myCanvas" width="200" height="200">Did
You Know: Every time
       you use a browser that doesn't support HTML5,
somewhere a kitten
       cries. Be nice to kittens, upgrade your browser!
      </canvas>
      <script>
// Get the context we will be using for drawing.
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

// Set the stroke style.
myContext.strokeStyle = '#000000';
myContext.lineWidth = 5;

// Create a closed path.
myContext.beginPath();
// Start the path at (30, 10).
myContext.moveTo(30, 10);
// Draw a line from the current pen location at (30, 10)
to (50, 50).
myContext.lineTo(50, 50);
// Draw a line from current pen location at (50, 50) to
(10, 50);
myContext.lineTo(10, 50);
// The pen is now currently at (10, 50). Closing the path
will draw a straight
// line from (10, 50) back to the beginning point of the
path at (30, 10).
myContext.closePath();
// We can't see the path without stroking it.
myContext.stroke();

// Create a new path.
myContext.beginPath();
// Start the path at (60, 10).
myContext.moveTo(60, 10);
// Draw a line from the current pen location at (60, 10)
to (100, 10).
myContext.lineTo(100, 10);
```

```
// Move the pen from its current location at (100, 10) to
(100, 50).
myContext.moveTo(100, 50);
// Draw a line from the current pen location at (100, 50)
to (60, 50).
myContext.lineTo(60, 50);
// Give the new shape a different color.
myContext.strokeStyle = '#ff0000';
myContext.stroke();

// Creating a new path will clear the current path from
memory without closing
// the previous one. We can demonstrate this by changing
the stroke style and
// calling stroke again. The previous shape should remain
red.
myContext.beginPath();
myContext.strokeStyle = '#00ff00';
myContext.stroke();
     </script>
   </body>
</html>
```

Basic Drawing Commands

The 2d drawing context provides a set of methods for drawing curves: lines, arcs, and so forth.

The `lineTo` Method

This command draws a path from the current pen position to the specified coordinates.

Syntax

```
Context.lineTo(x, y);
```

Table 9-3. *Parameters for the* `lineTo` *Method*

Parameter	Type	Explanation

x	Number	The x coordinate of the desired endpoint.
y	Number	The y coordinate of the desired endpoint.

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

myContext.lineTo(10, 10);
```

The `arc` Method

Draws an arc from `startAngle` to `endAngle` along a circle centered at coordinates `(x, y)` with radius `radius`.

Syntax

```
Context.arc(x, y, radius, startAngle, endAngle,
opt_isAnticlockwise);
```

Table 9-4. *Parameters for the `arc` Method*

Parameter	Type	Explanation
x	Number	The x coordinate of the desired center.
y	Number	The y coordinate of the desired center.
radius	Number	The radius of the arc, in pixels.
startAngle	Number	The start angle in radians.
endAngle	Number	The end angle in radians.
opt_isAnticlockwise	Boolean	If `true` the arc will be drawn anticlockwise. Optional; if not provided the default is `false`.

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

// Draw an arc starting from 0 to 3 radians.
myContext.arc(50, 50, 10, 0, 3);
```

The `quadraticCurveTo` Method

This command draws a quadratic curve starting at the current pen location and ending at the coordinates `(x, y)`, with the control point at `(cp1x, cp1y)`.

Syntax

```
Context.quadraticCurveTo(cplx, cply, x, y);
```

Table 9-5. *Parameters for the `quadraticCurveTo` Method*

Parameter	Type	Explanation
cplx	Number	The x coordinate of the control point.
cply	Number	The y coordinate of the control point.
x	Number	The x coordinate of the end of the curve.
y	Number	The y coordinate of the end of the curve.

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

myContext.quadraticCurveTo(50, 50, 10, 10);
```

The `bezierCurveTo` Method

Draws a bezier curve starting at the current pen location and ending at the coordinates `(x, y)`, with control point 1 specified by `(cp1x, cp1y)` and control point 2 specified by `(cp2x, cp2y)`.

Syntax

```
Context.bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y);
```

Table 9-6. *Parameters for the `bezierCurveTo` Method*

Parameter	Type	Explanation
cp1x	Number	The x coordinate of control point 1.

cp1y	Number	The y coordinate of control point 1.
cp2x	Number	The x coordinate of control point 2.
cp2y	Number	The y coordinate of control point 2.
x	Number	The x coordinate of the end point.
y	Number	The y coordinate of the end point.

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

myContext.bezierCurveTo(50, 50, 10, 10);
```

The `rect` Method

This comand draws a rectangle starting at coordinates `(x, y)` with the `width` and `height` specified.

Syntax

```
Context.rect(x, y, width, height);
```

Table 9-7. *Parameters for the `rect` Method*

Parameter	Type	Explanation
x	Number	The x coordinate of the upper left corner.
y	Number	The y coordinate of the upper left corner.
width	Number	The width of the rectangle, in pixels.
height	Number	The height of the rectangle, in pixels.

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

myContext.rect(50, 50, 10, 10);
```

Stroking and Filling Paths

As mentioned, the basic drawing commands create paths on the canvas that aren't visible. To make paths visible, you have to use the commands for stroking or filling. The 2d drawing context also provides a set of properties for defining the styles of strokes and fills.

The `strokeStyle` Property

This property specifies the style that should be applied in subsequent calls to the `stroke` method. This property can take any valid CSS color string (e.g., `'red'`, `'#ff0000'`, `'rgb(255, 0, 0)'`, etc.), a `Gradient` object, or a `Pattern` object. (See hereafter for how to define `Gradient` and `Pattern` objects.)

Syntax

```
Context.strokeStyle = StrokeStyleValue;
```

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

myContext.strokeStyle = '#FF0000';
```

The `stroke` Method

This command strokes the current path with the currently set stroke style.

Syntax

```
Context.stroke();
```

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');
```

```
myContext.strokeStyle = '#FF0000';
myContext.strokePath();
```

The `fillStyle` Property

This property specifies the style that should be applied in subsequent calls to the `fill` method. The property can take any valid CSS color string (e.g., `'red'`, `'#ff0000'`, `'rgb(255, 0, 0)'`, etc.), a `Gradient` object, or a `Pattern` object. (See hereafter for how to define `Gradient` and `Pattern` objects.)

Syntax

```
Context.fillStyle = FillStyleValue;
```

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

myContext.fillStyle = '#FF0000';
```

The `fill` Method

This command fills the current path with the currently set fill style.

Syntax

```
Context.fill();
```

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

myContext.fillStyle = '#FF0000';
myContext.fillPath();
```

The `lineWidth` Property

This property defines the thickness in units of the stroke applied to paths. If not set this property defaults to 1.

Syntax

```
Context.lineWidth = Number;
```

Table 9-8. *Values for the* `lineCap` *Property*

Value	Explanation
butt	The line ends are squared off and end precisely at the specified endpoint. This is the default value.
round	The line ends are rounded and end slightly over the specified endpoint.
square	The line ends are squared by adding a box to the end of the line whose width is equal to the width of the line and whose height is half of the width of the line.

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

myContext.lineWidth = 2;
```

The `lineCap` Property

This property defines how lines are capped. Valid values are `'butt'`, `'round'`, or `'square'`.

Syntax

```
Context.lineCap = LineCapValue;
```

Table 9-9. *Values for the* `lineJoin` *Property*

Value	Explanation
bevel	The joints are beveled.
miter	The joints are mitered.

round	The joints are rounded.

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

myContext.lineCap = 'round';
```

The `lineJoin` Property

This property defines how connecting lines are joined together. Valid values are `'bevel'`, `'miter'`, or `'round'`.

Syntax

```
Context.lineJoin = LineCapValue;
```

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

myContext.lineJoin = 'round';
```

Listing 9-3 provides an example of using the basic drawing commands and fill and stroke commands to create functions that will easily draw circles.

Listing 9-3. Circles Yay

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
canvas {
  border: 1px solid #000;
}
        </style>
    </head>
```

```
    <body>
      <canvas id="myCanvas" width="200" height="200">Did
You Know: Every time
       you use a browser that doesn't support HTML5,
somewhere a kitten
       cries. Be nice to kittens, upgrade your browser!
      </canvas>
      <script>
// Get the context we will be using for drawing.
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

/**
 * Draws a circle of the specified dimensions at the
target coordinates and
 * fills it with the current fill style.
 * @param {number} x The x coordinate of the center of the
circle.
 * @param {number} y The y coordinate of the center of the
circle.
 * @param {number} radius The radius of the circle.
 */
function fillCircle(x, y, radius) {
  myContext.beginPath();
  myContext.arc(x, y, radius, 0, 6.3);
  myContext.fill();
  myContext.closePath();
}

/**
 * Draws a circle of the specified dimensions at the
target coordinates and
 * strokes it with the current stroke style.
 * @param {number} x The x coordinate of the center of the
circle.
 * @param {number} y The y coordinate of the center of the
circle.
 * @param {number} radius The radius of the circle.
 */
function strokeCircle(x, y, radius) {
  myContext.beginPath();
  myContext.arc(x, y, radius, 0, 6.3);
```

```
  myContext.stroke();
  myContext.closePath();
}

// Set a fill style and draw a filled circle.
myContext.fillStyle = 'rgb(0, 0, 0)';
fillCircle(65, 65, 50);

// Set a stroke style and draw a stroked circle.
myContext.strokeStyle = 'rgb(0, 0, 0)';
myContext.lineWidth = 2;
strokeCircle(135, 135, 50);
     </script>
   </body>
</html>
```

Drawing Rectangles

In addition to basic paths, the 2d drawing context has a few functions for drawing simple rectangles. You could draw these with the basic path commands, but these convenience methods make it easier.

The `fillRect` Method

This command draws a rectangle at the specified coordinates and with the specified width and height, filled with the current fill style.

Syntax

```
Context. fillRect(x, y, width, height);
```

Table 9-10. *Parameters for the* `fillRect` *Method*

Parameter	Type	Explanation
x	Number	The x coordinate of the upper left corner.
y	Number	The y coordinate of the upper left corner.
width	Number	The width of the rectangle in pixels.
height	Number	The height of the rectangle in pixels.

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

myContext.fillStyle = '#000000';
myContext.fillRect(50, 50, 10, 10);
```

The `strokeRect` Method

This command draws a rectangle at the specified coordinates and with the specified width and height stroked with the current stroke style.

Syntax

```
Context. strokeRect(x, y, width, height);
```

Table 9-11. *Parameters for the* `strokeRect` *Method*

Parameter	Type	Explanation
x	Number	The x coordinate of the upper left corner.
y	Number	The y coordinate of the upper left corner.
width	Number	The width of the rectangle in pixels.
height	Number	The height of the rectangle in pixels.

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

myContext.strokeStyle = '#000000';
myContext.strokeRect(50, 50, 10, 10);
```

The `clearRect` Method

This command clears the specified rectangular area of any other drawing.

Syntax

```
Context. clearRect(x, y, width, height);
```

Table 9-12. *Parameters for the* `clearRect` *Method*

Parameter	Type	Explanation
x	Number	The x coordinate of the upper left corner.
y	Number	The y coordinate of the upper left corner.
width	Number	The width of the rectangle in pixels.
height	Number	The height of the rectangle in pixels.

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

myContext.clearRect(50, 50, 10, 10);
```

Listing 9-4 demonstrates drawing rectangles.

Listing 9-4. Random Rectangles

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
canvas {
  border: 1px solid #000;
}
        </style>
    </head>
    <body>
      <canvas id="myCanvas" width="200" height="200">Did
You Know: Every time
        you use a browser that doesn't support HTML5,
somewhere a kitten
        cries. Be nice to kittens, upgrade your browser!
      </canvas>
      <script>
// Get the context we will be using for drawing.
var myCanvas = document.getElementById('myCanvas');
```

```
var myContext = myCanvas.getContext('2d');

// Create a loop that will draw a random rectangle on the
canvas.
var cycles = 10,
    i = 0;
for (i = 0; i < cycles; i++) {
  var randX = getRandomIntegerBetween(0, 150);
  var randY = getRandomIntegerBetween(0, 150);
  var randWidth = getRandomIntegerBetween(10, 100);
  var randHeight = getRandomIntegerBetween(10, 100);
  myContext.beginPath();
  myContext.strokeRect(randX, randY, randWidth,
randHeight);
  randStroke();
  myContext.closePath();
}

/**
 * Returns a random integer between the specified minimum
and maximum values.
 * @param {number} min The lower boundary for the random
number.
 * @param {number} max The upper boundary for the random
number.
 * @return {number}
 */
function getRandomIntegerBetween(min, max) {
  return Math.floor(Math.random() * (max - min + 1))
+ min;
}

/**
 * Returns a random color formatted as an rgb string.
 * @return {string}
 */
function getRandRGB() {
  var randRed = getRandomIntegerBetween(0, 255);
  var randGreen = getRandomIntegerBetween(0, 255);
  var randBlue = getRandomIntegerBetween(0, 255);
  return 'rgb(' + randRed + ', ' + randGreen + ', '
+ randBlue + ')';
```

```
}

/**
 * Performs a randomized stroke on the current path.
 */
function randStroke() {
  myContext.lineWidth = getRandomIntegerBetween(1, 10);
  myContext.strokeStyle = getRandRGB();
  myContext.stroke();
}
     </script>
    </body>
</html>
```

Gradients and Patterns

Canvas has great support for gradients and patterns. Both patterns and gradients are represented by objects returned from construction functions. These objects can then be used as the values for fill or stroke styles.

The `createLinearGradient` Method

This method creates a linear gradient starting at coordinates `(startX, startY)` and ending at coordinates `(endX, endY)`. Returns a `Gradient` object that can be used as a stroke or fill style.

Syntax

```
Context.createLinearGradient(startX, startY, endX, endY);
```

Table 9-13. *Parameters for the `createLinearGradient` Method*

Parameter	Type	Explanation
startX	Number	The x coordinate of the start of the gradient.
startY	Number	The y coordinate of the start of the gradient.
endX	Number	The x coordinate of the end of the gradient.
endY	Number	The y coordinate of the end of the gradient.

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

var myGradient = myContext.createLinearGradient(50, 50,
10, 10);
```

The `createRadialGradient` Method

This method creates a radial gradient consisting of two circles, the first one centered at `(x, y)` with radius `r`, and the other centered at `(x1, y1)` with radius `r1`. Returns a `Gradient` object that can be used as a stroke or fill style.

Syntax

```
Context.createRadialGradient(x, y, r, x1, y1, r1);
```

Table 9-14. *Parameters for the `createRadialGradient` Method*

Parameter	Type	Explanation
x	Number	The x coordinate of the center of the first circle.
y	Number	The y coordinate of the center of the first circle.
r	Number	The radius of the first circle.
x1	Number	The x coordinate of the center of the second circle.
y1	Number	The y coordinate of the center of the second circle.
r1	Number	The radius of the second circle.

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

var myGradient = myContext.createRadialGradient(50, 50,
50, 50, 50, 100);
```

The `addColorStop` Method

This command adds a color stop to a `Gradient`. The position parameter must be between 0 and 1 and defines the relative position within the gradient of the color stop. The color can be any valid CSS color value. You can add as many color stops as you want to a particular `Gradient`.

Syntax

```
Gradient.addColorStop(position, color);
```

Table 9-15. *Parameters for the `lineTo` Method*

Parameter	Type	Explanation
position	Number	The position of the color stop.
color	CssColorValue	The color of the color stop.

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

var myGradient = myContext.createRadialGradient(50, 50,
50, 50, 50, 100);
myGradient.addColorStop(0, '#FF0000');
myGradient.addColorStop(1, '#000000');
```

The `createPattern` Method

This command creates a `Pattern` object that can be used as a fill or stroke style. The `Image` parameter must be any valid `Image` (see "Images" section next for details). The `repeat` parameter specifies how the pattern image is repeated, and must be one of `'repeat'`, `'repeat-x'`, `'repeat-y'`, or `'no-repeat'`.

Syntax

```
Gradient.createPattern(position, color);
```

Table 9-16. *Parameters for the `createPattern` Method*

Parameter	Type	Explanation

image	Image	The image to use to create the pattern.
repeat	string	How to repeat the image to create the pattern.

Table 9-17. *Valid Values for the Repeat Parameter*

Value	Explanation
repeat	Tile the image both horizontally and vertically.
repeat-x	Repeat the image only horizontally.
repeat-y	Repeat the image only vertically.
no-repeat	Do not repeat the image at all.

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

var myImage = document.getElementById('myImage');

var myPattern = myContext.createPattern(myImage,
'repeat');
```

Listing 9-5 demonstrates using radial gradients to fill random circles.

Listing 9-5. Generating and Using Radial Gradients to Fill Shapes

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
canvas {
  border: 1px solid #000;
}
        </style>
    </head>
    <body>
      <canvas id="myCanvas" width="200" height="200">Did
You Know: Every time
        you use a browser that doesn't support HTML5,
somewhere a kitten
        cries. Be nice to kittens, upgrade your browser!
```

```
    </canvas>
    <script>
// Get the context we will be using for drawing.
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

/**
 * Draws a circle of the specified dimensions at the
target coordinates and
 * fills it with the current fill style.
 * @param {number} x The x coordinate of the center of the
circle.
 * @param {number} y The y coordinate of the center of the
circle.
 * @param {number} radius The radius of the circle.
 */
function fillCircle(x, y, radius) {
  myContext.beginPath();
  myContext.arc(x, y, radius, 0, 6.3);
  myContext.fill();
  myContext.closePath();
}

/**
 * Returns a random integer between the specified minimum
and maximum values.
 * @param {number} min The lower boundary for the random
number.
 * @param {number} max The upper boundary for the random
number.
 * @return {number}
 */
function getRandomIntegerBetween(min, max) {
  return Math.floor(Math.random() * (max - min + 1))
+ min;
}

/**
 * Returns a random color formatted as an rgb string.
 * @return {string}
 */
function getRandRGB() {
```

```
  var randRed = getRandomIntegerBetween(0, 255);
  var randGreen = getRandomIntegerBetween(0, 255);
  var randBlue = getRandomIntegerBetween(0, 255);
  return 'rgb(' + randRed + ', ' + randGreen + ', '
+ randBlue + ')';
}

// Create a loop that will draw a random circle on the
canvas.
var cycles = 10,
    i = 0;
for (i = 0; i < cycles; i++) {
  // Get a random set of coordinates for the new circle.
  var randX = getRandomIntegerBetween(50, 150);
  var randY = getRandomIntegerBetween(50, 150);
  // Get a random radius.
  var randRadius = getRandomIntegerBetween(10, 50);
  // Create a gradient object based on the coordinates we
just generated.
  var randGrad = myContext.createRadialGradient(randX,
randY, 0, randX, randY,
      randRadius);
  // Create some random colors and add them as color stops
to the gradient.
  var randColor1 = getRandRGB();
  var randColor2 = getRandRGB();
  randGrad.addColorStop(0, randColor1);
  randGrad.addColorStop(1, randColor2);
  // Set the fill style and draw the circle.
  myContext.fillStyle = randGrad;
  fillCircle(randX, randY, randRadius);
}
      </script>
    </body>
</html>
```

Images

The 2d drawing context can also load and manipulate images. Valid image sources are an `img` element, a `video` element, or another `canvas` element. An image source doesn't have to be rendered as part of the DOM, so you can dynamically create tags and

load content as needed without necessarily having to attach them to the DOM. Once an image is loaded into a `canvas`, you can also draw on it with the drawing commands.

`Canvas` has one method for drawing images, `drawImage`, but it can take many different parameters and thus has multiple capabilities.

Drawing an Image

When you provide drawImage with an image source, an x coordinate, and a y coordinate, it will draw the image at the coordinates.

Syntax

```
Context.drawImage(image, x, y);
```

Table 9-18. *Parameters for the `drawImage` Method When Simply Drawing an Image*

Parameter	Type	Explanation
image	CanvasImageSource	A valid canvas image source.
x	Number	The x coordinate of the image.
y	Number	The y coordinate of the image.

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');
var myImage = document.getElementById('myImage');

myContext.drawImage(myImage, 10, 10);
```

Scaling an Image

When you provide `drawImage` with an image source, an x coordinate, a y coordinate, a width, and a height, it will draw the image at the coordinates and scale the image to the specified `width` and `height`.

Syntax

```
Context.drawImage(image, x, y, width, height);
```

Table 9-19. *Parameters for the* `drawImage` *Method When Scaling an Image*

Parameter	Type	Explanation
image	CanvasImageSource	A valid canvas image source.
x	Number	The x coordinate of the image.
y	Number	The y coordinate of the image.
width	Number	The desired width of the image, in pixels.
Height	Number	The desired height of the image, in pixels.

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');
var myImage = document.getElementById('myImage');

myContext.drawImage(myImage, 10, 10, 50, 50);
```

Drawing a Slice of an Image

You can also select a specific area on an image (a “slice”) and draw just that on the canvas.

Syntax

```
Context.drawImage(image, sliceX, sliceY, sliceWidth,
sliceHeight, x, y);
```

Table 9-20. *Parameters for the* `drawImage` *Method When Drawing a Slice of an Image*

Parameter	Type	Explanation
image	CanvasImageSource	A valid canvas image source.
sliceX	Number	The x coordinate on the image of the upper left corner of the slice.
sliceY	Number	The y coordinate of the image of the upper left corner of the slice.
sliceWidth	Number	The desired width of the slice, in pixels.
sliceHeight	Number	The desired height of the slice, in pixels.

x	Number	The x coordinate at which to draw the image slice.
y	Number	The y coordinate at which to draw the image slice.

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');
var myImage = document.getElementById('myImage');

myContext.drawImage(myImage, 10, 10, 50, 50, 0, 0);
```

Listing 9-6 demonstrates loading an image into our basic canvas template.

Listing 9-6. Loading an Image into a `canvas` Element

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
canvas {
  border: 1px solid #000;
}
        </style>
    </head>
    <body>
      <canvas id="myCanvas" width="200" height="200">Did
You Know: Every time
        you use a browser that doesn't support HTML5,
somewhere a kitten
        cries. Be nice to kittens, upgrade your browser!
      </canvas>
      <script>
// Get the context we will be using for drawing.
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

// Create a new image element and fill it with a kitten.
var myImage = new Image();
myImage.src = 'http://lorempixel.com/g/200/200/cats';

// We can't do anything until the image has successfully
```

```
loaded.
myImage.onload = function() {
  myContext.drawImage(myImage, 0, 0);
};
      </script>
    </body>
</html>
```

Here you're simply loading a random placeholder image into the canvas at position `(0, 0)`. Listing 9-7 shows a more complex manipulation of an image.

Listing 9-7. Manipulating an Image Using Canvas

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
canvas {
  border: 1px solid #000;
}
        </style>
    </head>
    <body>
      <canvas id="myCanvas" width="200" height="200">Did
You Know: Every time
        you use a browser that doesn't support HTML5,
somewhere a kitten
        cries. Be nice to kittens, upgrade your browser!
      </canvas>
      <script>
// Get the context we will be using for drawing.
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

// Create a new image element and fill it with a kitten.
var myImage = new Image();
myImage.src = 'http://lorempixel.com/g/300/300/cats';

// We can't do anything until the image has successfully
loaded.
myImage.onload = function() {
```

```
  myContext.drawImage(myImage, 25, 25, 150, 150, 0, 0,
150, 50);
};
     </script>
    </body>
</html>
```

Here you're loading a 300 × 300 placeholder image, but slicing only a 75 × 75 portion of it starting at `(25, 25)`. Then you take that slice and render it in the `canvas`, scaling it to be 150 × 50.

Text

The 2d drawing context can also be used to render text.

The `fillText` Method

This method fills the specified text on the canvas starting at the specified coordinates with the current fill style. If the optional `maxWidth` parameter is specified, and the rendered text would exceed that width, the browser will attempt to render the text in such a way as to fit it within the specified width (use a condensed font face if available, use a smaller font size, etc.).

Syntax

```
Context.fillText(textString, x, y, opt_maxWidth);
```

Table 9-21. *Parameters for the `fillText` Method When Scaling an Image*

Parameter	Type	Explanation
textString	string	A text string.
x	Number	The x coordinate at which the text should be rendered.
y	Number	The y coordinate at which the text should be rendered.
opt_maxWidth	Number	A maximum width, in pixels.

Example

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

```
var myContext = myCanvas.getContext('2d');

myContext.fillText('Hello world!', 10, 10, 200);
```

The `measureText` Method

This method measures the width that would result if the specified text were to be rendered using the current style. Returns a `TextMetrics` object that has a `width` property that contains the value. This provides a way for you to test how well text will fit in a given area without actually having to render it.

Syntax

```
Context.measureText(textString);
```

Table 9-22. *Parameters for the `measureText` Method When Scaling an Image*

Parameter	Type	Explanation
textString	string	A text string.

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

var textMetric = myContext.measureText('Hello world!');
var calculatedWidth = textMetric.width;
```

The `strokeText` Method

This method strokes the specified text on the canvas starting at specified coordinates with the current stroke style. If the optional `maxWidth` parameter is specified, and the rendered text would exceed that width, the browser will attempt to render the text in such a way as to fit it within the specified width (use a condensed font face if available, use a smaller font size, etc.).

Syntax

```
Context.strokeText(textString, x, y, opt_maxWidth);
```

Table 9-23. *Parameters for the* `strokeText` *Method When Scaling an Image*

Parameter	Type	Explanation
textString	string	A text string.
x	Number	The x coordinate at which the text should be rendered.
y	Number	The y coordinate at which the text should be rendered.
opt_maxWidth	Number	A maximum width, in pixels.

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

myContext.strokeText('Hello world!', 10, 10, 200);
```

The `font` Property

This property defines the font that the text will be rendered in. Any valid CSS font string is permitted, but note that the user has to have the specified font installed on their system.

Syntax

```
Context.font = CssFontString;
```

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

myContext.font = 'arial, helvetica, sans-serif';
```

The `textAlign` Property

This property defines how the text is aligned when it is rendered. Valid values are `'left'`, `'right'`, `'center'`, `'start'`, and `'end'`.

Syntax

```
Context.textAlign = AlignValue;
```

Table 9-24. *Values for the* `textAlign` *Property*

Value	Explanation
left	Left-align the text.
right	Right-align the text.
center	Center the text.
start	Align the text at the starting side for the current locale (that is, left for left-to-right languages and right for right-to-left languages). This is the default value.
end	Align the text at the ending side for the current locale.

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

myContext.textAlign = 'center';
```

The `textBaseline` Property

This property defines the baseline of the text when it renders. Valid values are `'alphabetic'`, `'bottom'`, `'hanging'`, `'ideographic'`, `'middle'`, and `'top'`.

Syntax

```
Context.textAlign = AlignValue;
```

Table 9-25. *Values for the* `textBaseline` *Property*

Value	Explanation
alphabetic	Use the normal alphabetic baseline for the text. This is the default value.
bottom	The baseline is the bottom of the em square.
hanging	Use the hanging baseline for the text.
ideographic	Use the bottom of the body of characters (assuming they protrude beneath the alphabetic baseline).

middle	The text baseline is the middle of the em square.
top	The text baseline is the top of the em square.

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

myContext.textAlign = 'center';
```

Listing 9-8 demonstrates rendering text using the text commands.

Listing 9-8. Rendering Text in Canvas

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
canvas {
  border: 1px solid #000;
}
        </style>
    </head>
    <body>
      <canvas id="myCanvas" width="200" height="200">Did
You Know: Every time
        you use a browser that doesn't support HTML5,
somewhere a kitten
        cries. Be nice to kittens, upgrade your browser!
      </canvas>
      <script>
// Get the context we will be using for drawing.
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

myContext.font = '35px sans-serif';
myContext.strokeStyle = '#000';
myContext.lineWidth = 2;
myContext.textAlign = 'center';
myContext.strokeText('Hello World', 100, 100);
```

```
    </script>
  </body>
</html>
```

Shadows

The `canvas` element can also cast shadows based on the elements drawn upon it. This is most often used with text, but it also works with shapes and paths. If you're already familiar with CSS drop shadows, the parameters for canvas shadows will be very familiar.

The `shadowBlur` Property

This property defines the size of the blurring effect. Valid values are 0 (no blur, which is the default) or any positive integer.

Syntax

```
Context.shadowBlur = ShadowBlurValue;
```

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

myContext.shadowBlur = 5;
```

The `shadowColor` Property

This property defines the color of the shadow. Any CSS color string is a valid value. The default is `'rgba(0, 0, 0, 0)'`.

Syntax

```
Context.shadowColor = CssColorValue;
```

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

myContext.shadowColor = '#00FF00';
```

The `shadowOffsetX` Property

This property defines the x-offset of the shadow. Valid values are any positive or negative integer, or 0 (which is the default).

Syntax

```
Context.shadowOffsetX = OffsetValue;
```

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

myContext.shadowOffsetX = 5;
```

The `shadowOffsetY` Property

This property defines the y-offset of the shadow. Valid values are any positive or negative integer, or 0 (which is the default).

Syntax

```
Context.shadowOffsetY = OffsetValue;
```

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

myContext.shadowOffsetY = 5;
```

Listing 9-9 demonstrates creating a drop shadow on text.

Listing 9-9. Drop Shadows in Canvas

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
canvas {
  border: 1px solid #000;
}
        </style>
    </head>
    <body>
      <canvas id="myCanvas" width="200" height="200">Did
You Know: Every time
        you use a browser that doesn't support HTML5,
somewhere a kitten
        cries. Be nice to kittens, upgrade your browser!
      </canvas>
      <script>
// Get the context we will be using for drawing.
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

// Define a shadow.
myContext.shadowBlur = 2;
myContext.shadowColor = 'rgba(0, 100, 0, 0.5)';
myContext.shadowOffsetX = 5;
myContext.shadowOffsetY = 5;

myContext.font = '35px sans-serif';
myContext.strokeStyle = '#000';
myContext.lineWidth = 2;
myContext.textAlign = 'center';
myContext.strokeText('Hello World', 100, 100);
      </script>
    </body>
</html>
```

Compositing

Whenever you draw a new element on the canvas, the compositor looks at what is already present on the canvas. This current content is referred to as the *destination*. The new content is referred to as the *source*. Then the compositor draws the source in reference to the destination according to the currently active compositor.

The `globalCompositeOperation` Property

This property specifies which compositor is currently active.

Syntax

```
Context.globalCompositeOperation = CompositorValue;
```

Table 9-26. *Values for the* `globalCompositeOperation` *Property*

Value	Explanation
source-over	Draw source content over destination content. This is the default compositor.
source-atop	Source content is only drawn where it overlaps the destination content.
source-in	Source content is only drawn where both source and destination content overlap. Everything else is made transparent.
source-out	Source content is only drawn where it does not overlap destination content. Everything else is made transparent.
destination-over	Source content is drawn underneath destination content.
destination-atop	Source content is only kept where it overlaps the destination content. The destination content is drawn underneath the source. Everything else is made transparent.
destination-in	Source content is only kept where it overlaps with the destination content. Everything else is made transparent.
destination-out	Source content is only kept where it does not overlap with the destination content. Everything else is made transparent.
copy	Only draws the destination content. Everything else is made transparent.
lighter	Where destination content and source content overlap, the color is determined by adding the values of the two contents.
xor	The destination content is rendered normally except where it overlaps with source content, in which case both are rendered transparent.

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

myContext.globalCompositeOperation = 'source-atop';
```

Listing 4-16 in Chapter 4 provides a dynamic example of all of these properties.

Clipping

You can limit the drawing area of the canvas to any closed path that you have defined. This is referred to as *clipping*.

The `clip` Method

This command allows you to create a clipping area based on the current path. Only content contained within the clipping area will display. To reset the clipping area, you can do one of three things:

- You can define a path that encompasses the entire canvas, and then clip to that.
- You can restore to a previous drawing state with a different clipping area. This is the most common solution. You can save the drawing state before clipping, then restore it when you're done. See the "Saving and Restoring Canvas State" section hereafter for details on how to save state.
- You can reset the entire canvas by resizing it.

Syntax

```
Context.clip();
```

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

myContext.clip();
```

Listing 9-10 demonstrates creating a clipping area and using it to clip off the corners of a square.

Listing 9-10. Creating a Clipping Area

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
canvas {
  border: 1px solid #000;
}
        </style>
    </head>
    <body>
      <canvas id="myCanvas" width="200" height="200">Did
You Know: Every time
        you use a browser that doesn't support HTML5,
somewhere a kitten
        cries. Be nice to kittens, upgrade your browser!
      </canvas>
      <script>
// Get the context we will be using for drawing.
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

// Create a circular clipping area.
myContext.beginPath();
myContext.arc(100, 100, 50, 0, 7);
myContext.clip();

// Draw a square in the canvas and fill it. Only the
portion within the clipping
// area will be visible, so the corners will be cut off.
myContext.beginPath();
myContext.rect(60, 60, 80, 80);
myContext.fillStyle = 'black';
myContext.fill();
      </script>
    </body>
</html>
```

Transformations

The 2d drawing context supports various types of transformations. Once a transformation is set it will be applied to everything that is rendered from that point on.

The `translate` Method

This method moves the origin of the canvas from its current position to the new position specified by the coordinates.

Syntax

```
Context.translate(translateX, translateY);
```

Table 9-27. *Parameters for the* `translate` *Method When Scaling an Image*

Parameter	Type	Explanation
translateX	Number	The new x coordinate of the origin.
translateY	Number	The new y coordinate of the origin.

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

myContext.translate(10, 50);
```

The `rotate` Method

This method rotates the canvas clockwise around the origin by the specified angle in radians.

Syntax

```
Context.rotate(angle);
```

Table 9-28. *Parameters for the* `translate` *Method When Scaling an Image*

Parameter	Type	Explanation
angle	Number	The angle of the rotation, in radians.

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

myContext.rotate(2);
```

The `scale` Method

This method scales the canvas units by `scaleX` horizontally and `scaleY` vertically.

Syntax

```
Context.scale(scaleX, scaleY);
```

Table 9-29. *Parameters for the `scale` Method When Scaling an Image*

Parameter	Type	Explanation
scaleX	Number	The amount to scale the x axis.
scaleY	Number	The amount to scale the y axis.

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

myContext.translate(10, 50);
```

The `transform` Method

This method allows you to specify a generic transformation matrix:

$$\begin{bmatrix} scaleX & skewY & translateX \\ scaleX & skewY & translateX \\ 0 & 0 & 1 \end{bmatrix}.$$

The `rotate`, `translate`, and `scale` shorthand methods all map to transformation matrices and thus calls to the `transform` method. For example, `Context.translate(translateX, translateY)` maps to `Context.transform(1, 0, 0, 1, translateX, translateY)` and `Context.scale(scaleX, scaleY)` maps to `Context.transform(scaleX, 0, 0, scaleY, 0, 0)`.

Syntax

```
Context.transform(scaleX, skewX, skewY, scaleY,
translateX, translateY);
```

Table 9-30. *Parameters for the `bezierCurveTo` Method*

Parameter	Type	Explanation
scaleX	Number	The amount to scale the x axis.
skewX	Number	The amount to skew the x axis.
skewY	Number	The amount to skew the y axis.
scaleY	Number	The amount to scale the y axis.
translateX	Number	The x coordinate of the new origin.
translateY	Number	The y coordinate of the new origin.

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

// Reset all transformations.
myContext.transform(0, 0, 0, 0, 0, 0);
```

Listing 9-11 demonstrates using the `scale` and `translate` transforms.

Listing 9-11. Using the `scale` and `translate` Transforms

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
        <style>
```

```
canvas {
  border: 1px solid #000;
}
      </style>
    </head>
    <body>
      <canvas id="myCanvas" width="200" height="200">Did
You Know: Every time
       you use a browser that doesn't support HTML5,
somewhere a kitten
       cries. Be nice to kittens, upgrade your browser!
      </canvas>
      <script>
// Get the context we will be using for drawing.
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

/**
 * Draws a 100x100 square at (0, 0) in the specified
color. Indicates the origin
 * corner with a small black square.
 * @param {string} color A valid CSS color string.
 */
function drawSquare(color) {
  myContext.fillStyle = color;
  myContext.beginPath();
  myContext.rect(0, 0, 100, 100);
  myContext.fill();
  myContext.fillStyle = '#000';
  myContext.beginPath();
  myContext.rect(0, 0, 5, 5);
  myContext.fill();
}

// Draw a square, fill it with red.
drawSquare('rgba(255, 0, 0, 0.5)');

// Translate the canvas.
myContext.translate(20, 40);

// Scale the canvas.
myContext.scale(1, 1.5);
```

```
// Draw the same square again, fill it with blue.
drawSquare('rgba(0, 0, 255, 0.5)');

// Translate the canvas again.
myContext.translate(50, -20);

// Scale the canvas again.
myContext.scale(1.5, 1);

// Draw the same square again, fill it with green.
drawSquare('rgba(0, 255, 0, 0.5)');
     </script>
   </body>
</html>
```

Saving and Restoring Canvas State

The 2d drawing context includes a basic state management system. A given state is made up of the following properties in the context:

- The current value for `globalAlpha`
- The current `strokeStyle` and `fillStyle`
- The current line settings in `lineCap`, `lineJoin`, `lineWidth`, and `miterLimit`
- The current shadow settings in `shadowBlur`, `shadowColor`, `shadowOffsetX`, and `shadowOffsetY`
- The current compositing operation set in `globalCompositeOperation`
- The current clipping path
- Any transformations that have been applied to the drawing context

State is saved in a last-in first-out stack, so the last state you saved will be the first one available for retrieving. There is no way to skip around in the stack.

The `save` Method

This command saves the current context to the stack.

Syntax

```
Context.save();
```

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

// Saves an initial "blank" canvas state before anything
has been drawn or set.
myContext.save();
```

The `restore` Method

This command removes the most recently stored state from the stack and restores it to the context.

Syntax

```
Context.restore();
```

Example

```
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

// Saves an initial "blank" canvas state before anything
has been drawn or set.
myContext.restore();
```

Listing 9-12 demonstrates saving and restoring state.

Listing 9-12. Saving and Restoring Drawing Context State

```
<!DOCTYPE html>
<html>
   <head>
      <title>The HTML5 Programmer's Reference</title>
      <style>
```

```
canvas {
  border: 1px solid #000;
}
      </style>
    </head>
    <body>
      <canvas id="myCanvas" width="200" height="210">Did
You Know: Every time
      you use a browser that doesn't support HTML5,
somewhere a kitten
      cries. Be nice to kittens, upgrade your browser!
     </canvas>
     <script>
// Get the context we will be using for drawing.
var myCanvas = document.getElementById('myCanvas');
var myContext = myCanvas.getContext('2d');

// Create an array of colors to load into the stack.
var allTheColors = ['#ff0000', '#ff8800', '#ffff00',
'#00ff00', '#0000ff',
    '#4b0082', '#8f00ff'];

// Load the colors and stroke style into the stack.
for (var i = 0; i < allTheColors.length; i++) {
  myContext.strokeStyle = allTheColors[i];
  myContext.lineWidth = 30;
  myContext.save();
}

// Restore colors from the stack and draw.
for (var i = 0; i < 8; i++) {
  myContext.restore();
  myContext.beginPath();
  myContext.moveTo(0, ((30 * i) + 15));
  myContext.lineTo(200, ((30 * i) + 15));
  myContext.stroke();
}
     </script>
    </body>
</html>
```

APPENDIX A

JavaScript Tips and Techniques

JavaScript is the de facto programming language of the Web, and is the primary tool you'll be using to interact with the HTML5 features covered in this book. In this chapter I'll cover a few tips on organizing the JavaScript for your applications, and some more powerful techniques you can use to simplify your scripts. This brief chapter isn't meant to be a full JavaScript reference—for that, you can consult other Apress titles, such as *JavaScript Programmer's Reference*.

Code Formatting

Generally speaking, I avoid holy wars about code formatting styles such as bracketing and indentation. Usually the individual choices don't matter because what is more important is consistency throughout your code. You should always use the same bracketing style, comment style, indentation choice, and so on because it will help keep your code readable. Even if you're the only one who will ever look at your code, it's still important.

With JavaScript, however, these choices can matter. Bracket placement, for example, matters in certain cases in JavaScript. Consider this code snippet:

```
function test1tbs() {
  return {
    objectLiteral: 'value',
    isExpected: true
  };
}
```

This example follows the so-called One True Brace Style (occasionally abbreviated as 1TBS), in which the opening bracket of a function definition (and an object literal) is

on the same line as its declaration. This is the style I've used in the examples throughout the book. However, the same example could be written using Allman-style bracket placement:

```
function testAllman()
{
  return
    {
      objectLiteral: 'value',
      isExpected: false
    };
}
```

These two functions will have vastly different results. The `test1tbs` function will return the object literal that is defined inline as part of the return statement, while code containing the `testAllman` function won't even run (the JavaScript engine will throw an error). In other languages these two functions would be identical.

In addition, the ECMAScript standard that governs the behavior of JavaScript also defines rules about how missing semicolons should be interpreted. This is called Automatic Semicolon Insertion (ASI) and is a feature of the language, but it can occasionally lead to surprising results. This is why the de facto bracketing style in JavaScript code is the 1TBS style and not the more spacious Allman style.

JavaScript Rewards Verbosity

As you've read the examples throughout this book, you've probably noticed that the code style is fairly verbose. There are expansive comments, variable and function names tend to be long and descriptive, and so forth. This is partially because the code samples are designed to be easy to read and understand, but overall this level of verbosity isn't that much higher than the code I write on a daily basis.

JavaScript has several convenience features such as ASI, type coercion (which comes into play when you compare two variables of different types), and so forth. Sometimes these features can cause surprising results. Verbose code helps avoid these surprises by reminding you what your variables and functions are supposed to be doing, and by providing documentation of logical flow and expected behaviors. It also makes debugging easier, and if you're collaborating with other people it will help them learn your code more quickly.

Comment Annotations

Also, in the book's examples I've been annotating code using a specific comment format. If you're familiar with either JSDoc or JavaDoc, these comments will look familiar, because the format is derived from JSDoc. If you're not familiar with JSDoc, it's a standard for comments in JavaScript code that not only provides a great way of explaining the code, but also allows you to use parsing tools to generate actual documentation from the comments. In the examples, I haven't been using the full capabilities of JSDoc—I've just been focusing on providing type annotations. If you're familiar with annotating code for the Closure JavaScript compiler, these comments will be quite familiar.

Tip To learn more about annotating code for the Closure compiler, and using the Closure compiler in your projects, see https://developers.google.com/closure/compiler/docs/js-for-compiler. You can learn more about JSDoc and automatic documentation generators at www.usejsdoc.org/.

For every function, the comments specify the following:

- A description of what the function is supposed to do.
- The expected data type of each parameter for the function (if any).
- The data type of the return value (if any).
- Whether or not the function is meant to be private (only applicable for members of a class).

For example, consider this function from Chapter 4:

```
/**
 * Returns a random integer between the specified minimum
and maximum values.
 * @param {number} min The lower boundary for the random
number.
 * @param {number} max The upper boundary for the random
number.
 * @return {number}
 */
```

```
function getRandomIntegerBetween(min, max) {
  return Math.floor(Math.random() * (max - min + 1))
+ min;
}
```

You may be wondering what the value is of defining a parameter's expected type in a dynamically typed language like JavaScript. Obviously you can pass in any values you want and the JavaScript engine will coerce the values as best it can, which could possibly cause the function to return an unexpected result. By defining expected data types in the function definition, you not only indicate what the function needs in order to avoid unexpected results—you also make it easier to use the function later after you have forgotten the details yourself. Further, if you're collaborating with a team, you make it easier for them to use the function. In the particular case of this book, my intention is that the type annotations will help make the examples easier to understand.

JavaScript doesn't directly support the concept of public or private properties or methods. Specifying the `@private` tag on a property or method helps define class structures in a more traditional way and can make JavaScript code a little more palatable to people used to languages with more rigorous encapsulation features. I also find it helpful to keep in mind what interfaces I intend to be public or private, because if I then find myself needing to change those decisions it often indicates that the underlying class structure needs revision.

In this book I've used these tags:

- `@private`: Indicates that the property or method is considered private to its context. Typically this isn't enforced in any way and only helps clarify intent.
- `@constructor`: Indicates that the function is a JavaScript constructor, which will instantiate and return the specified object type when used in conjunction with the `new` keyword.
- `@param`: Indicates a parameter for a function or method. A `@param` definition will include a type definition in brackets, the name of the parameter as used throughout the function, and an optional description of what the parameter is. Optional parameters are indicated with the Optional operator (see hereafter for details).
- `@return`: Indicates that the function returns a value. A `@return` tag will include a type definition in brackets, indicating the data type of the return value.
- `@type`: Indicates the type of a variable or property as it is being defined.

Type definitions are an important part of the annotations. All type definitions are enclosed in curly brackets. Since JavaScript is dynamically typed, type annotations can specify multiple data types, with each data type separated by a vertical slash. For example:

```
{boolean}
```

specifies a boolean type, while

```
{boolean|number}
```

specifies that the type can be either a boolean or a number.

Compound types are specified using angular brackets. For example:

```
{Array<boolean>}
```

specifies that the type is an array of booleans, while

```
{Object<string, number>}
```

specifies that the object has keys that are strings and the associated values are numbers.

There are also a few operators used in type definitions:

- Nullable: The `?` operator indicates that the type can be the specified data type or null. Thus `{?Object}` is the equivalent of `{Object|null}`. I haven't used the nullable operator in the examples in this book; instead I've made the assumption that all types are nullable by default unless specified otherwise using the non-nullable operator.
- Non-nullable: The `!` operator indicates that the type cannot be null. For example, `{!Array<!string>}` specifies that the type must be an array of strings. An empty array, or an array of other types, is not permitted.
- Optional: The = operator used in a `@param` type definition indicates that the parameter is optional. I amplify this by prepending the prefix `opt_` to the name of all optional parameters. For example, `@param {boolean=} opt_isActive` specifies that the parameter `opt_isActive` is optional, but if it is present it must be a boolean (or null).

Using Objects as Event Handlers

One of my favorite little-known features of the DOM is the `EventListener` interface. We all know how to attach an event listener to a DOM element using `Element.addEventListener` method, which takes three parameters:

- `eventType`: a string that indicates the type of event
- `handler`: a function to execute when the event happens
- `bubble`: whether or not to execute the function during the bubble phase

What's little known is that the DOM specifies you can use any object as the handler, as long as it implements the `EventListener` interface. According to the DOM Level 2 standard:

> *The EventListener interface is the primary method for handling events. Users implement the EventListener interface and register their listener on an EventTarget using the AddEventListener method. The users should also remove their EventListener from its EventTarget after they have completed using the listener.*

An `EventListener` interface is defined as a method called `handleEvent` on any object:

```
interface EventListener {
  void handleEvent(in Event evt);
};
```

This means that any object that implements a `handleEvent` method can be used as an event handler, as demonstrated in Listing A-1.

Listing A-1. Using an Object As an Event Handler

```
<!DOCTYPE html>
<html>
    <head>
        <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
```

```
      <p id="targetElement">Click me!</p>
      <script>
var targetElement
= document.getElementById('targetElement');

var eventObject = {
  handleEvent: function(event) {
    console.log(event.type);
  }
};

targetElement.addEventListener('click', eventObject,
true);
      </script>
    </body>
</html>
```

In this example you've created a simple `eventObject` that implements the `EventListener` interface in the form of a method called `handleEvent`. You then bind it to the target element's click event, and when you click on the "click me" text you will see "click" appear in the console.

This technique is useful for encapsulating event handlers in objects and classes, rather than having them as separate functions. You can even create a single event handler object that has multiple event handlers on it, and use the EventListener interface to delegate activity as needed. For example, recall the WebSockets example (Listing 3-7) in Chapter 3, which had separate functions for handling `error`, `close`, `open`, and `message` events, all bound to the WebSocket interface. You could easily create all of those event handlers as methods on a single object, as shown in Listing A-2.

Listing A-2. Rewriting Listing 3-7 to use a Generic EventListener Interface

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>The HTML5 Programmer's Reference</title>
  </head>
  <body>
    <h1>Web Sockets Demonstration</h1>
    <script>
// Create a new web socket connection to the chat service.
var chatUrl = 'ws://www.fgjkjk4994sdjk.com/chat';
```

```
var validProtocols = ['chat', 'json'];
var chatSocket = new WebSocket(chatUrl, validProtocols);

/**
 * Creates an error handling class that implements the
EventListener interface.
 * @constructor
 * @returns {Object}
 */
function CreateWebSocketEventObject() {
  /**
   * Handles an error event on the chat socket object.
   * @private
   */
  this.handleError_ = function() {
    console.log('An error occurred on the chat
connection.');
  };

  /**
   * Handles a close event on the chat socket object.
   * @param {CloseEvent} event The close event object.
   * @private
   */
  this.handleClose_ = function(event) {
    console.log('The chat connection was closed because ',
event.reason);
  };

  /**
   * Handles an open event on the chat socket object.
   * @param {OpenEvent} event The open event object.
   * @private
   */
  this.handleOpen_ = function(event) {
    console.log('The chat connection is open.');
  };

  /**
   * Handles a message event on the chat socket object.
   * @param {MessageEvent} event The message event object.
   * @private
   */
```

```
  this.handleMessage_ = function(event) {
    console.log('A message event has been sent.');

    // The event object contains the data that was
transmitted from the server.
    // That data is encoded either using the chat protocol
or the json protocol,
    // so we need to deterine which protocol is being
used.
    if (chatSocket.protocol === validProtocols[0]) {
      console.log('The chat protocol is active.');
      console.log('The data the server transmitted is: ',
event.data);
      // etc...
    } else {
      console.log('The json protocol is active.');
      console.log('The data the server transmitted is: ',
event.data);
      // etc...
    }
  };

  /**
   * Implements the EventListener interface for the object
and invokes the
   * correct handler based on the event type.
   * @param {SocketEvent} event
   */
  this.handleEvent = function(event) {
    switch (event.type) {
      case 'error':
        this.handleError_();
        break;
      case 'close':
        this.handleClose_(event);
        break;
      case 'open':
        this.handleOpen_(event);
        break;
      case 'message':
        this.handleMessage_(event);
        break;
```

```
      default:
        console.warn('Unknown event of type ', event.type);
    }
  };
}

// Create a new event object using the constructor.
var eventHandlerObject = new CreateWebSocketEventObject();

// Bind the event object to the chat socket.
chatSocket.addEventListener('error', eventHandlerObject);
chatSocket.addEventListener('close', eventHandlerObject);
chatSocket.addEventListener('open', eventHandlerObject);
chatSocket.addEventListener('message',
eventHandlerObject);
   </script>
  </body>
</html>
```

In this version of the example you have built a constructor function that returns an object that implements the `EventListener` interface. Within that interface method it checks the incoming event's type property and invokes the correct handler method. This gives you better encapsulation of the event handlers, and opens up the possibility of easily opening multiple Web Sockets and using the same constructor to build event handlers for all of them.

Promises

Asynchronous activities are quite common in JavaScript applications, and the standard way of handling them is with callback functions. As an example, consider the dynamic script loading that you were doing in Chapter 6. Listing A-13 had a function that dynamically loaded a specified script and executed either a success or error callback function depending on the result:

```
/**
 * Dynamically loads a script and invokes an optional
callback.
 * @param {string} srcUrl The URL of the script file to
load.
 * @param {function=} opt_onLoadCallback An optional
```

```
function to call when the
 *     script is loaded.
 * @param {function=} opt_onErrorCallback An optional
function to call if the
 *     script fails to load.
 */
function loadScript(srcUrl, opt_onLoadCallback,
opt_onErrorCallback) {

  // Create a script tag.
  var newScript = document.createElement('script');

  // Apply the load callback, if one was provided.
  if (opt_onLoadCallback) {
    if (newScript.readyState) {
      // Internet explorer.
      newScript.onreadystatechange = function() {
        if (newScript.readyState == 'loaded' ||
            newScript.readyState == 'complete') {
          newScript.onreadystatechange = null;
          opt_onLoadCallback.call();
        }
      };
    } else {
      // Every other browser in the universe.
      newScript.onload = opt_onLoadCallback;
    }
  }

  // Apply the error callback, if one was provided.
  if (opt_onErrorCallback) {
    newScript.onerror = opt_onErrorCallback;
  }

  newScript.src = srcUrl;
  document.querySelector('head').appendChild(newScript);
}
```

This function takes three parameters: the URL of the script it needs to load, and the success and error callback functions.

The problem with using callbacks is that they result in convoluted code. And if you have nested callbacks—for example, if the success callback function also executes another asynchronous task—the callbacks can become difficult to manage and the code

difficult to read.

Promises provide a different way of handling asynchronous actions in JavaScript code. A Promise is an object that represents the result of an asynchronous action. The actual result (success or failure) doesn't need to be known at the time that the Promise is created; instead the asynchronous action will return a Promise object like any other synchronous action. This allows you to simplify your asynchronous code and reduce or even eliminate your need for nested callbacks.

A Promise object is in one of four states:

- fulfilled: The asynchronous action that the Promise represents has finished and was successful.
- rejected: The asynchronous action that the Promise represents has finished but resulted in an error.
- pending: This is the initial state of the Promise when it is created. A Promise in the pending state is neither fulfilled nor rejected.
- settled: The Promise is no longer pending and has either fulfilled or rejected.

Once a Promise has entered a fulfilled or rejected state it cannot change, so a fulfilled promise can never later become rejected and vice versa.

A Promise is created using the Promise constructor:

```
var myPromise = new Promise(executor)
```

The `executor` is a function with two parameters: `resolve` and `reject`. When you create the new Promise, these `resolve` and `reject` parameters will be placeholders for functions you'll be specifying later. Typically they're functions that you will be calling as the asynchronous action succeeds or fails.

A Promise object exposes an API for accessing the state of the asynchronous action and anything it might return.

- `Promise.then(resolve, reject)`: The `then` method enables you to specify the `resolve` and `reject` functions that will be called when the Promise is settled.
- `Promise.catch(reject)`: The `catch` method allows you to specify just the `reject` function that will be called when the promise is rejected.

To demonstrate creating a basic Promise and then assigning resolve and reject

handlers, Listing A-3 shows Listing A-13 rewritten to use a Promise.

Listing A-3. Using a Promise to Represent Dynamically Loading a Script

```
<!DOCTYPE html>
<html>
    <head>
       <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
      <script src="../js-lib/detect-support.js"></script>
      <script>
/**
 * Dynamically loads a script and invokes an optional
callback.
 * @param {string} srcUrl The URL of the script file to
load.
 * @return {Promise<null>}
 */
function loadScript(srcUrl) {
  var myPromise = new Promise(function(resolve, reject) {
    var newScript = document.createElement('script');
    if (newScript.readyState) {
      // Internet explorer.
      newScript.onreadystatechange = function() {
        if (newScript.readyState == 'loaded' ||
            newScript.readyState == 'complete') {
          newScript.onreadystatechange = null;
          resolve();
        }
      };
    } else {
      // Every other browser in the universe.
      newScript.onload = resolve;
    }
    newScript.onError = reject;
    newScript.src = srcUrl;
    document.querySelector('head').appendChild(newScript);
  });
  return myPromise;
}
```

```
// Test for supported features.
var supportedFeatures = new DetectHTML5Support();

if (!supportedFeatures.localStorage) {
  // The Web Storage is not supported, so load a shim. The
loadScript function
  // now returns a Promise.
  loadScript('../js-lib/webstorage-
shim.js').then(function() {
    initApplication();
  }, function() {
    console.log('Script failed to load.');
  });
} else {
  // Web Storage was supported, so continue with the
application.
  initApplication();
}

/**
 * Hypothetical function for initializing the application.
 */
function initApplication() {
  console.log('Application continues...');
  // Etc.
}
    </script>
  </body>
</html>
```

You'll notice that the `loadScript` function now constructs and returns a Promise using placeholders for the `resolve` and `reject` functions that will be specified later. When the function is called, the code applies the `resolve` and `reject` functions using the `Promise.then` method.

Chaining Promises

Promises provide a lot of flexibility for dealing with situations involving multiple asynchronous actions. For example, if you return a Promise from the `Promise.then` method, you can chain Promises together. To illustrate this, you can use the `loadScript` function to load three different scripts one after the other. Since the

`loadScript` function returns a Promise, you can simply chain the calls to `Promise.then` together. To do that you'll need to start the chain with an empty Promise that will always succeed:

```
var promiseChain = Promise.resolve();
promiseChain.then(function() {
  return loadScript('script1.js');
}).then(function() {
  return loadScript('script2.js');
}).then(function() {
  return loadScript('script3.js');
}).catch(function() {
  console.log('An error occurred when loading the
scripts.');
});
```

If you use the feature registry pattern mentioned in the "Working with Broken or Missing HTML5 Implementations" section of Chapter 6, you can reduce this even further to a simple `for` loop. Listing A-4 demonstrates using this technique to rewrite Listing A-14.

Listing A-4. Chaining Promises to Load Multiple Shims in Sequence

```
<!DOCTYPE html>
<html>
    <head>
       <title>The HTML5 Programmer's Reference</title>
    </head>
    <body>
      <script src="../js-lib/detect-support.js"></script>
      <script>
// Create a registry of HTML features that we need and
shims to apply if they
// are not present. The registry will be an array of
objects; each object will
// consist of a feature name and a path to a shim to apply
if that feature is
// not supported.
var featureRegistry = [
  {
    'featureName' : 'sessionStorage',
    'shim' : '../js-lib/webstorage-shim.js'
```

```
  },
  {
    'featureName' : 'requestAnimationFrame',
    'shim' : '../js-lib/animationframe-shim.js'
  }
];

/**
 * Dynamically loads a script and invokes an optional
callback.
 * @param {string} srcUrl The URL of the script file to
load.
 * @return {Promise<null>}
 */
function loadScript(srcUrl) {
  var myPromise = new Promise(function(resolve, reject) {
    var newScript = document.createElement('script');
    if (newScript.readyState) {
      // Internet explorer.
      newScript.onreadystatechange = function() {
        if (newScript.readyState == 'loaded' ||
            newScript.readyState == 'complete') {
          newScript.onreadystatechange = null;
          resolve();
        }
      };
    } else {
      // Every other browser in the universe.
      newScript.onload = resolve;
    }
    newScript.onError = reject;
    newScript.src = srcUrl;
    document.querySelector('head').appendChild(newScript);
  });
  return myPromise;
}

// Test for supported features.
var supportedFeatures = new DetectHTML5Support();

// Go through the registry and for each item load a shim
if it isn't supported.
```

```
var promiseChain = Promise.resolve();
featureRegistry.forEach(function(currFeature) {
  if (!supportedFeatures[currFeature.featureName]) {
    promiseChain = promiseChain.then(function() {
      return loadScript(currFeature.shim);
    });
  }
});

promiseChain.then(function() {
  initApplication();
}, function() {
  console.log('A shim failed to load.');
});

/**
 * Hypothetical function for initializing the application.
 */
function initApplication() {
  console.log('Application continues...');
  // Etc.
}

    </script>
  </body>
</html>
```

The first thing you will probably notice about this example is it is considerably more compact than the original example in Chapter 6. Simpler code is one of the benefits of using Promises.

In this example you start by creating a fulfilled Promise as the first link in your Promise chain. You then loop through the feature registry and test each feature. Unsupported features have shims loaded, and their Promises added to the chain. When then use the `Promise.then` method on the chain. If all the required features are supported, then the chain will consist of just the initial fulfilled promise, so the `resolve` handler will be invoked immediately. If there are unsupported features, then the chain will consist of multiple Promises, each of which will execute in turn.

Returning Values from Promises

In my examples so far, the asynchronous actions haven't returned any values. They've

just succeeded or failed. Many asynchronous actions will return a value that you'll need to access within your `resolve` and `reject` methods. To do this, you can specify parameters for your `resolve` and `reject` methods. For example, imagine a situation where you fetch data from a server using a `fetchData` method that returns a promise:

```
function fetchData() {
  var myPromise = new Promise(function(resolve, reject) {
    var client = new XMLHttpRequest();
    client.open('POST',
'http://www.fakeservice.com/myservice');
    client.send();

    client.onload = function () {
      if (this.status == 200) {
        // Successfully fetched information from the
service. Resolve the
        // promise with the information.
        resolve(this.response);
      } else {
        // Did not successfully fetch information from the
service. Reject the
        // promise with the error message.
        reject(this.statusText);
      }
    };
    client.onerror = function () {
      reject(this.statusText);
    };
  });
  return myPromise;
}

fetchData().then(function(serviceResponse) {
  console.log('The service returned ', serviceResponse);
}, function(errorMessage) {
  console.error(errorMessage);
});
```

In this example function you create a new `XMLHttpRequest` object to asynchronously fetch data from a server, but return a Promise that wraps the response value. You can then access the response value in the `Promise.then` callbacks.

Browser Support for Promises

Promises are a relatively new feature for JavaScript. As of this writing, all browsers except Internet Explorer support Promises, and Internet Explorer Edge will have full support when it is released. In the meantime there is a good shim for Promises available at https://github.com/jakearchibald/es6-promise.

Further Reading

This is just a brief introduction to Promises. There's a lot more you can do with them. Many of the examples in this book can be rewritten using Promises, resulting in simpler code.

To learn more about Promises, check out these resources:

- The Promises/A+ specification at https://promisesaplus.com/
- The Mozilla Developer Network reference for Promises at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Pr
- The Promises tutorial at HTML5Rocks, www.html5rocks.com/en/tutorials/es6/promises/

Index

A

Animation
- draw cycle constructor
- JavaScript-based
- with canvas

Animation timing
- API methods
- benefits
- brute-force method
- draw cycle manager
- frames
- global context
- JavaScript implementation of a timer-based draw cycle
- requestAnimationFrame
- startAnimation

APIs. *See* Application programming interfaces (APIs)

Application programming interfaces (APIs)
- audio and video tags
- cross document messaging/web messaging
  - handleMessage event handler
  - main and target page
  - postMessage method
  - RFC 6454
  - Same Origin Policy
- DOM
- Drag and Drop (*see* Drag-and-Drop interactions, APIs)
- Server-sent events
  - application
  - EventSource object
  - handleUpdates method
  - JSON-formatted text
  - one-way communication
  - origin limitations
  - polling script
  - race condition
  - security
  - stock ticker
  - stock values
  - text/event-stream MIME type
  - timer
  - timer-based polling scripts
  - XHR object
  - XMLHttpRequest

WebSockets (*see* WebSocket, APIs)
Web storage (*see* Web storage, APIs)
web worker
asynchronous activity
computation-intensive activity
creation
divide and conquer
image processing
location object and properties
multithreading
navigator object and property
web applications
XMLHttpRequest
Article element
properties
standards
usage
ASI. *See* Automatic Semicolon Insertion (ASI)
Aside element
properties
standards
usage
Audio and video content, HTML5
codecs parameter
multimedia
Shadow DOM APIs
software capabilities
src attribute
Audio element
global attributes
information, web
properties
standards
usage
Automatic semicolon insertion (ASI)

B

bdi element
properties
standards
usage
beginPath method
Browser support
broken/missing HTML5 implementations
array featureRegistry
checking single feature
Internet Explorer
loading shims
multiple shims and tracking process
onLoadCallback function

readystatechange event handler
registering callbacks, loading process
unsupported feature
verifyAllFeatures function
Can I Use database
canvas element
feature detection
feature detection script
Chrome, Firefox, and Internet Explorer
getFailedTestResults
getTestResults
listing
HTML5 Rocks website
interface, HTML Element
Internet Explorer
JavaScript properties and methods
detection
event interfaces detection
HTML5 features detection
nonexistent property
non supporting browsers
undefined
undefined *vs*. null
MobiDex (*see* MobiDex)
mobile browsers
Modernizr
Promises
testing
attribute support
supported elements
Webshim library
Browser wars

C

Callback functions
Can I Use database
Canvas
animation
clipping
compositing
drawing commands
arc method
bezierCurveTo method
Context.fill()
Context.fillStyle
Context.lineCap
Context.lineJoin
Context.lineWidth
Context.stroke()
Context.strokeStyle

curves
declaring paths
drawing rectangles
lineCap property
lineJoin property
lineTo method
quadraticCurveTo method
random circle generator
rect method
drawing context
alternate content, older browsers
API
pen metaphor
small victories
syntax
width and height
drawing mode
gradients (*see* Gradients, HTML5 canvas element)
gradients and patterns
addColorStop method
createLinearGradient method
createPattern method
createRadialGradient method
images (*see* Images, canvas; Images, HTML5 canvas element)
implementation
interaction
drawing with mouse
event handler
requestAnimationFrame method
patterns (*see* Patterns, HTML5 canvas element)
properties
restore method
save method
saving
saving and restoring, properties
saving drawing state
and restoring
properties
shadows
shadowBlur property
shadowColor property
shadowOffsetX property
shadowOffsetY property
standards
stroking/filling
clearRect method
drawing rectangles
fill method
fillRect method
fillStyle property
lineCap property

lineJoin property
lineWidth property
stroke method
strokeRect method
strokeStyle property
text (*see* Text, 2d drawing context; Text, canvas)
Cascading style sheets (CSS)
Clipping
clip method
creation
definition
effects of
resetting
closePath method
Code formatting
Comment annotations
@constructor
@param
@private
@return
@type
type definitions
Compositing
demonstration
destination
globalCompositeOperation property
gradient
photograph

D

Data element
properties
standards
usage
Datalist element
properties
usage
2D drawing context
API
defining paths
beginPath method
closePath method
moveTo method
drawing commands
arc method
bezierCurveTo method
lineTo method
quadraticCurveTo method
rect method
drawing rectangles

clearRect method
fillRect method
strokeRect method
path
pen metaphor
rendering text
fillText method
transformations (*see* Transformations)
Details and summary elements
properties
usage
Device orientation
Chrome and Firefox
compassneedscalibration event
devicemotion event
deviceorientation event
Dexterity puzzles (*see* MobiDex)
Drag-and-Drop interactions, APIs
CSS class
dataTransfer object
dataTransfer.setDragImage()
draggable property
event.preventDefault()
preventDefault()
updation, script
visual clipboard

E

Embedded audio content
autoplay
Chrome browser
controls
loop
muted
preload
src
web page
Embedded video content
autoplay
controls
height
loop
muted
poster
preload
src
web page
width

F

Feature detection
Figure and figcaption elements
    properties
    standards
    usage
Footer element
    properties
    standards
    usage
Form element attributes
    autocomplete
    autofill
    autofocus
    placeholder text
Form elements
    datalist element
    meter element
    output element
    progress element

G

Generalized markup language (GML)
Geolocation
    API
        browser and hosting device
        callback function
        Chrome
        globalTimeout variable
        globalTimeoutCallback function
        navigator.geolocation.clearWatch
        navigator.geolocation.getCurrentPosition
        navigator.geolocation.watchPosition
        object templates
        permission options in Firefox 29
        permission options, internet explorer
        PositionError object
        position object
        Safari Mobile on iOS
        “Share Location”
        simple location query
        timer running
        transparent
        user interaction standpoint
        valid PositionError codes
    application collects and processes
    Bluetooth mapping
    cellular towers
    GPS satellites

- iOS location accuracy
- IP address mapping
- mobile devices
- privacy and security
- Wi-Fi mapping

Gradients, HTML5 canvas element
- Context.createLinearGradient
- Context.createRadialGradient
- linear gradient object
- Gradient.addColorStop
- three-stop

Grouping
- figure and figcaption elements
- main element

H

Header element
- properties
- standards
- usage

HTML
- anchor tags
- browser wars
  - internet explorer
  - libwww
  - Lynx browser
  - Mosaic code
  - mozilla foundation
  - NCSA
  - netscape navigator
- document markup languages
- ENQUIRE
- GML
- HTML5
- hyperlinks
- hypermedia
- hypertext
- IETF
- information sharing
- markup languages
  - descriptive
  - presentation
  - procedural
- markup tags
- oNLine System/NLS
- pilot project
- Project Xanadu
- SGML
  - applications
  - tags

simple client/server network protocol
XHTML
zippered lists
HTML5
canvas
browser support (*see* Browser support)
features
JavaScript APIs
tags
HTML5 API
cross-document dessaging
attributes
host page
parameters
structured clone algorithm
syntax
target page
dataTransfer object
drag and drop
draggable property
drop targets
event.dataTransfer.files and types
server-sent events
EventSource constructor
JSON-formatted text
multiline key
multiple data attributes
WebSockets
API
constructor
communication
full duplex communication
interface
web storage
API definition
interface objects
localStorage and sessionStorage implement
StorageEvent object
web workers
blocking method and origin policy
communication
creation
EventTarget interface
execution context
global JavaScript scope
stand-alone script
threaded JavaScript applications
writing multithreaded applications
HTML5 canvas element (*see* Canvas)
HTML5 elements
audio, video content (*see* Audio, video content, HTML5)

automatic rendering, browsers
deprecated elements
functionality
grouping content, tags
interactive elements
CSS rules, progress
CSS styles, web dialogs
data lists
dialogs
forms
meter
output, form
progress bar
progressive disclosure
timer, progress
nonsemantic tags
div-it/divitis
old and busted markup
obsolete parameters
semantic markup
semantic tags
bi-directional isolation
data
hotness markup
line breaks
mark
Ruby annotations
time
word break
words marking, document
HTML5 Rocks website

I

IETF. *See* Internet engineering task force (IETF)
Images, canvas
drawing
drawing slice of image
scaling
Images, HTML5 canvas element
Context.drawImage
parameters
scaling
slicing and scaling
sources
Input types
application, Chrome
color
dates and times
email
number

- range
- search
- tel
- url

Interactive elements
Internet engineering task force (IETF)

J, K, L

JavaScript
- code formatting
- Promises (*see* Promises)
- verbosity

JavaScript APIs

M

Main element
- properties
- standards
- usage

Mark element
- properties
- standards
- usage

Meter element
- attributes
- standards
- usage

MobiDex
- Add a timer
- Add customization
- Add scoring
- comparing coordinates
  - checking collisions
  - drawGameField method and associated properties
  - MobiDex Class
  - updateRemainingBalls method
- deviceorientation event handler
- Draw Cycle
- finished game
- full listing and coordinate classes
- game initialization
- obstacles and targets
  - simple coordinate class
  - generating random integers
- playing field UI
  - CSS
  - markup
  - rendering
- user story (*see* User story)

Modernizr
moveTo method
Mozilla foundation

N

National center for supercomputing applications (NCSA)
Nav element
    properties
    standards
    usage
NCSA. *See* National center for supercomputing applications (NCSA)
Netscape navigator

O

Objects, as event handlers
Output element
    properties
    standards
    usage

P, Q

Patterns
    Context.createPattern
    creation
    kitten as pattern
Progress element
    properties
    standards
    usage
Promises
    browser support
    chaining
        loadScript function
        multiple shims loading
    constructor
    creation
    definition
    loading script
    returning values
    states

R

Ruby, rp, and rt elements
    properties
    standards
    usage

S

Scalable vector graphics (SVG)
Section elements
  article element
  aside element
  footer element
  header element
  nav element
  properties
  standards
  usage
Selectors
  accessing elements
  attribute
  element abstract class
  element state pseudo-classes
  JavaScript frameworks
  negation pseudo-class
  NodeList objects
  querySelector and querySelectorAll methods
  structural pseudo-classes
Semantics
  bdi element
  data element
  mark element
  ruby, rp, and rt elements
  time element
  wbr element
Shadows
  shadowBlur property
  shadowColor property
  shadowOffsetX property
  shadowOffsetY property
Shadows, canvas
  drop shadows
  parameters
  rendering
Source element
  properties
  standards
  usage
Standard generalized markup language (SGML)
  CERN
  SGMLguid
stopAnimation method

T

Text, 2d drawing context
  fillText method

font property
measureText method
strokeText method
textAlign property
textBaseline property
Text, canvas
Context.fillText
Context.font
Context.measureText
Context.strokeText
Context.textAlign
Context.textBaseline
rendering
Time element
properties
standards
usage
Track element
properties
standards
usage
WebVTT (*see* Web Video Text Tracks Format (WebVTT))
Transformations
arbitrary transformation matrix
Context.rotate
Context.scale
Context.translate
definition
resetting
rotate method
scale method
scale and translate
stacking rotation
text reflection
transform method
translate method
translate transformation
Type definitions
non-nullable
nullable
optional

U

User story
agile software development
definition
features

V

Verbosity
Video element
  global attributes
  information, web
  properties
  standards
  usage

W, X, Y, Z

W3C. *See* World Wide Web Consortium (W3C)
Wbr element
  properties
  standards
  usage
Web Applications standard
  data storage
  draft proposal
  semantic markup tags
  server-pushed events
  state management
Web Graphics Library (WebGL)
Web hypertext application technology working group (WHATWG)
Webshim library
WebSocket, APIs
  close method
  demonstration class
  full duplex communication
  handshake process
    'chat' and 'json' protocols
    GET query
    two-way connection
  headers
  HTTP
  information receiving
    binary large objects
    Blobs and ArrayBuffers
    connection object
    error event connection
    stubbed event handlers
  information transmitting
  network protocol
  sendDataAndClose_
  subprotocol
  url parameter
Web storage, APIs
  custom storage methods
  Evercookies
  HTTP Cookies
  JSON string
  localStorage

privacy concerns
sessionStorage
sessionStorage.getItem
sessionStorage.setItem
Web Video Text Tracks Format (WebVTT)
closed captioning file
WebVTT (*see* Web Video Text Tracks Format (WebVTT))
WHATWG. *See* Web hypertext application technology working group (WHATWG)
World Wide Web Consortium (W3C)
CSS
HTML 4.0
standards process
XML-based solutions