sitepoint

SIMPLY JAVASCRIPT

BY KEVIN YANK
& CAMERON ADAMS

EVERYTHING YOU NEED TO LEARN JAVASCRIPT FROM SCRATCH

SIMPLY JAVASCRIPT

BY **KEVIN YANK**
& CAMERON ADAMS

Simply JavaScript

by Kevin Yank and Cameron Adams

Managing Editor: Simon Mackie
Technical Editor: Kevin Yank
Technical Director: Kevin Yank
Editor: Georgina Laidlaw
Index Editor: Max McMaster
Cover Design: Alex Walker
Printing History:
First Edition: June 2007

Notice of Rights

Notice of Liability

Trademark Notice

Published by SitePoint Pty. Ltd.

424 Smith Street Collingwood
VIC Australia 3066
Web: www.sitepoint.com
Email: business@sitepoint.com

ISBN 978-0-9802858-0-2
Printed and bound in Canada

About Kevin Yank

As Technical Director for SitePoint, Kevin Yank keeps abreast of all that is new and exciting in web technology. Best known for his book, *Build Your Own Database Driven Website Using PHP & MySQL*,[1] now in its third edition, Kevin also writes the *SitePoint Tech Times*,[2] a free, biweekly email newsletter that goes out to over 150,000 subscribers worldwide.

When he isn't speaking at a conference or visiting friends and family in Canada, Kevin lives in Melbourne, Australia, and enjoys performing improvised comedy theater with Impro Melbourne,[3] and flying light aircraft. His personal blog is *Yes, I'm Canadian*.[4]

About Cameron Adams

Cameron Adams melds a background in Computer Science with almost a decade's experience in graphic design, resulting in a unique approach to interface design. He uses these skills to play with the intersection between design and code, always striving to create interesting and innovative sites and applications.

Having worked with large corporations, government departments, nonprofit organizations, and tiny startups, he's starting to get the gist of this Internet thing. In addition to the projects that pay his electricity bills, Cameron muses about web design on his well-respected web-log—*The Man in Blue*[5]—and has written several books on topics ranging from JavaScript to CSS and design.

Sometimes he's in Melbourne, other times he likes to fly around the world to talk about design and programming with other friendly geeks. If you ever see him standing at a bar, buy him a Baileys and say "hi."

About SitePoint

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

---

[1] http://www.sitepoint.com/books/phpmysql1/
[2] http://www.sitepoint.com/newsletter/
[3] http://www.impromelbourne.com.au/
[4] http://yesimcanadian.com/
[5] http://themaninblue.com/

Without you, Lisa, this book would never have been written. I can only hope to return the same amount of love and support that you have given me.

—Cameron

To Jessica,
my partner in crime,
the lemon to my lime.

—Kevin

Table of Contents

Preface

On the surface, JavaScript is a simple programming language that lets you make changes to your web pages on the fly, while they're being displayed in a web browser. How hard could that be to learn, right? It sounds like something you could knock over in an afternoon.

But JavaScript is bigger on the inside than it seems from the outside. If you were a *Dr. Who* fan, you might call it the Tardis of programming languages. If you're *not* a *Dr. Who* fan, roll your eyes with me as the fanboys (and girls) geek out.

Everyone back with me? Put your Daleks away, Jimmy.

As I was saying, JavaScript *sounds* like it should be simple. Nevertheless, throughout its ten year history (so far), the best ways of doing things with JavaScript have seemed to change with the seasons. And advice on how to write good JavaScript can be found everywhere: "Do it this way—it'll run faster!" "Use this code—it'll run on more browsers!" "Stay away from that feature—it causes memory leaks!"

Too many other JavaScript books—some of them from very respected names in the industry—will teach you a handful of simple solutions to simple problems and then call it a day, leaving you with just enough rope with which to hang yourself when you actually try to solve a real-world problem on your own. And when in desperation you go looking on the Web for an example that does what you need it to, you'll likely be unable to make sense of the JavaScript code you find, because the book you bought didn't cover many of the truly useful features of the language, such as object literals, event listeners, or closures.

This book aims to be different. From the very first page, we'll show you the *right* way to use JavaScript. By working through fully fleshed-out examples that are ready to be plugged right into a professionally-designed web site, you'll gain the confidence not only to write JavaScript code of your own, but to understand code that was written by others, and even to spot harmful, old-fashioned code that's more trouble than it's worth!

Throughout this book, we've tried to go the extra mile by giving you more than just the basics. In particular, we've covered some of the new JavaScript-powered devel-

opment techniques—like Ajax—that are changing the face of the Web. We've also included sections that explore the new crop of JavaScript libraries like jQuery, Prototype, Yahoo! UI, and Dojo, making this the only beginner's JavaScript book to cover these powerful time-savers.

… all of which made this book a lot harder to write, but that's why they pay us the big bucks.

Who Should Read this Book?

Whether you've never seen a line of JavaScript code in your life, or you've seen one too many lines that doesn't do what you expect, this book will show you how to make JavaScript work for you.

We assume going in that you've got a good handle on web design with HyperText Markup Language (HTML) and Cascading Style Sheets (CSS). You needn't be an expert in these languages, but as we'll see, JavaScript is just another piece in the puzzle. The better you understand basic web design techniques, the more you can enhance them with JavaScript.

If you need a refresher, we highly recommend *Build Your Own Web Site The Right Way Using HTML & CSS*[1] (Melbourne: SitePoint, 2006).

What's Covered in this Book?

Chapter 1: The Three Layers of the Web

A big part of learning JavaScript is learning when it's the right tool for the job, and when ordinary HTML and CSS can offer a better solution. Before we dive into learning JavaScript, we'll take a little time to review how to build web sites with HTML and CSS, and see just how JavaScript fits into the picture.

Chapter 2: Programming with JavaScript

JavaScript is a programming language. To work with it, then, you must get your head around the way computer programs work—which to some extent means learning to think like a computer. The simple concepts introduced in this

[1] http://www.sitepoint.com/books/html1/

chapter—statements, variables, expressions, loops, functions, and objects—are the building blocks for every JavaScript program you'll ever write.

Chapter 3: Document Access

While certain people enjoy writing JavaScript code for its own sake, you wouldn't want to run into them in a dark alley at night. As a well-adjusted web developer, you'll probably want to use JavaScript to make changes to the contents of your web pages using the Document Object Model (DOM). Lucky for you, we wrote a whole chapter to show you how!

Chapter 4: Events

By far the most *event*ful portion of this book (ha ha ha … I slay me), this chapter shows you how to write JavaScript programs that will respond to the actions of your users as they interact with a web page. As you'll see, this can be done in a number of ways, for which varying degrees of support are provided by current browsers.

Chapter 5: Animation

Okay, okay. We can talk all day about the subtle usability enhancements that JavaScript makes possible, but we know you won't be satisfied until you can make things swoosh around the page. In this chapter, you'll get all the swooshing you can handle.

Chapter 6: Form Enhancements

I know what you're thinking: forms are boring. Nobody leaps out of bed in the morning, cracks their knuckles, and shouts, "Today, I'm going to fill in some *forms*!" Well, once you trick out your forms with the enhancements in this chapter, they just might. Oh, and just to spice up this chapter a bit more, we'll show you how to make an element on your page draggable.

Chapter 7: Errors and Debugging

When things go wrong in other programming languages, your computer will usually throw a steady stream of error messages at you until you fix the problem. With JavaScript, however, your computer just folds its arms and gives you a look that seems to say, "You were expecting, maybe, something to happen?" No, English is not your computer's first language. What did you expect? It was made in Taiwan. In this chapter, we'll show you how to fix scripts that don't behave the way they should.

Chapter 8: Ajax

You might have heard about this thing called Ajax that makes web pages look like desktop applications, and shaky business ventures look like solid investments. We put it into this book for both those reasons.

Chapter 9: Looking Forward

JavaScript doesn't just *have* a future; JavaScript *is* the future! Okay, you might think that's taking it a bit far, but when you read this chapter and see the many amazing things that JavaScript makes possible, you might reconsider.

Appendix A: The Core JavaScript Library

As we progress through the book, we'll write code to solve many common problems. Rather than making you rewrite that code every time you need it, we've collected it all into a JavaScript library that you can reuse in your own projects to save yourself a *ton* of typing. This appendix will provide a summary and breakdown of all the code that's collected in this library, with instructions on how to use it.

The Book's Web Site

Located at http://www.sitepoint.com/books/javascript1/, the web site that supports this book will give you access to the following facilities.

The Code Archive

As you progress through this book, you'll note file names above many of the code listings. These refer to files in the code archive, a downloadable ZIP file that contains all of the finished examples presented in this book. Simply click the **Code Archive** link on the book's web site to download it.

Updates and Errata

No book is error-free, and attentive readers will no doubt spot at least one or two mistakes in this one. The Corrections and Typos page on the book's web site[2] will provide the latest information about known typographical and code errors, and will offer necessary updates for new releases of browsers and related standards.

[2] http://www.sitepoint.com/books/javascript1/errata.php

The SitePoint Forums

If you'd like to communicate with other web developers about this book, you should join SitePoint's online community.[3] The JavaScript forum,[4] in particular, offers an abundance of information above and beyond the solutions in this book, and a lot of fun and experienced JavaScript developers hang out there. It's a good way to learn new tricks, get questions answered in a hurry, and just have a good time.

The SitePoint Newsletters

In addition to books like this one, SitePoint publishes free email newsletters including *The SitePoint Tribune*, *The SitePoint Tech Times*, and *The SitePoint Design View*. Reading them will keep you up to date on the latest news, product releases, trends, tips, and techniques for all aspects of web development. If nothing else, you'll get useful CSS articles and tips, but if you're interested in learning other technologies, you'll find them especially valuable. Sign up to one or more SitePoint newsletters at http://www.sitepoint.com/newsletter/.

Your Feedback

If you can't find an answer through the forums, or if you wish to contact us for any other reason, the best place to write is `books@sitepoint.com`. We have an email support system set up to track your inquiries, and friendly support staff members who can answer your questions. Suggestions for improvements as well as notices of any mistakes you may find are especially welcome.

Acknowledgments

Kevin Yank

I'd like to thank Mark Harbottle and Luke Cuthbertson, SitePoint's Co-founder and General Manager, who sat me down late in 2006 and—for the second time in my career—convinced me that stepping away from SitePoint's day-to-day operations to write a book wouldn't be the worst career move ever. I also owe a beverage to

[3] http://www.sitepoint.com/forums/
[4] http://www.sitepoint.com/launch/jsforum/

Simon Mackie, whose idea it was in the first place. Let's hope someone buys it, guys!

To Jessica, for the many evenings that I stayed at the office to write long past the hour I said I'd be home, and for the boundless support and patience with which she greeted my eventual arrival, I owe something big and chocolaty.

And to the more than 150,000 readers of the *SitePoint Tech Times* newsletter,[5] with whom I shared many of the ideas that made their way into this book, and who provided valuable and challenging feedback in return, my gratitude.

Cameron Adams

The knowledge I've accrued on JavaScript has been drawn from so many sources that it would be impossible to name them all. Anything that I can pass on is only due to the contributions of hundreds—if not thousands—of charitable individuals who use their valuable time to lay out their knowledge for the advantage of others. If you're ever in a position to add to those voices, try your hardest to do so. Still, I'd like to put out an old school shout-out to the *Webmonkey* team, in particular Thau and Taylor, who inspired me in the beginning. I'd also like to thank my coding colleagues, who are always available for a quick question or an extended discussion whenever I'm stuck: Derek Featherstone, Dustin Diaz, Jonathan Snook, Jeremy Keith, Peter-Paul Koch, and Dan Webb.

[5] http://www.sitepoint.com/newsletter/

Conventions Used in this Book

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

Code Samples

Any code will be displayed using a fixed-width font like so:

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

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

example.css

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

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

example.css *(excerpt)*

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

Tips, Notes, and Warnings

Hey, You!

Tips will give you helpful little pointers.

Ahem, Excuse Me ...

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

Make Sure you Always ...

... pay attention to these important points.

Watch Out!

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

Chapter 1

The Three Layers of the Web

Once upon a time, there was … 'A king!' my little readers will say right away. No, children, you are wrong. Once upon a time there was a piece of wood…
—The Adventures of Pinocchio

You can do a lot without JavaScript. Using Hypertext Markup Language (HTML),[1] you can produce complex documents that intricately describe the content of a page—and that content's meaning—to the minutest detail. Using Cascading Style Sheets (CSS), you can present that content in myriad ways, with variations as subtle as a single color, as striking as replacing text with an image.

No matter how you dress it up, though, HTML and CSS can only achieve the static beauty of the department store mannequin—or at best, an animatronic monstrosity that wobbles precariously when something moves nearby. With JavaScript, you can bring that awkward puppet to life, lifting you as its creator from humble shop clerk to web design mastery!

[1] Throughout this book, we'll refer to HTML and XHTML as just HTML. Which *you* choose is up to you, and doesn't have a much to do with JavaScript. In case it matters to you, the HTML code we'll present in this book will be valid XHTML 1.0 Strict.

But whether your new creation has the graceful stride of a runway model, or the shuffling gait of Dr. Frankenstein's monster, depends as much on the quality of its HTML and CSS origins as it does on the JavaScript code that brought it to life.

Before we learn to work miracles, therefore, let's take a little time to review how to build web sites that look good both inside *and* out, and see how JavaScript fits into the picture.

Keep 'em Separated

Not so long ago, professional web designers would gleefully pile HTML, CSS, and JavaScript code into a single file, name it **index.html**,[2] and call it a web page. You can still do this today, but be prepared for your peers to call it something rather less polite.

Somewhere along the way, web designers realized that the code they write when putting together a web page does three fundamental things:

[2] Or **default.htm**, if they had been brainwashed by Microsoft.

- It describes the *content* of the page.
- It specifies the *presentation* of that content.
- It controls the *behavior* of that content.

They also realized that keeping these three types of code separate, as depicted in Figure 1.1, made their jobs easier, and helped them to make web pages that work better under adverse conditions, such as when users have JavaScript disabled in their browsers.

Computer geeks have known about this for years, and have even given this principle a geeky name: the **separation of concerns**.

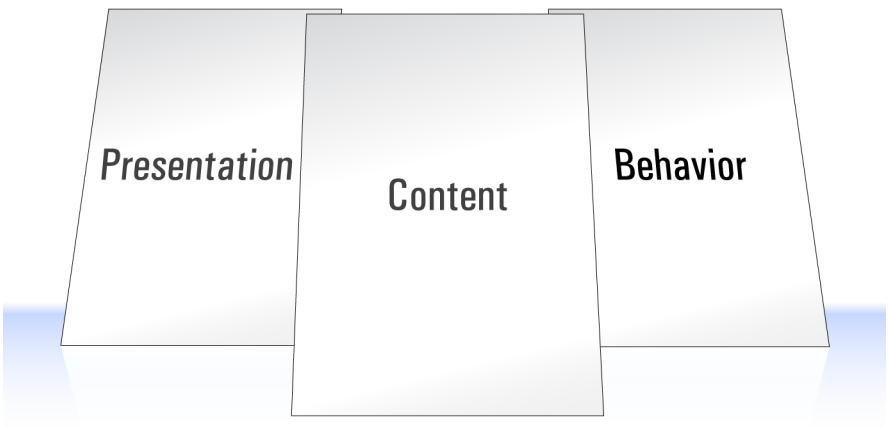


Figure 1.1. Separation of concerns

Now, realizing this is one thing, but actually *doing* it is another—especially if you're not a computer geek. I *am* a computer geek, and I'm tempted to do the wrong thing all the time.

I'll be happily editing the HTML code that describes a web page's content, when suddenly I'll find myself thinking how nice that text would look if it were in a slightly different shade of gray, if it were nudged a little to the left, and if it had that hee-larious photocopy of my face I made at the last SitePoint office party in the background. Prone to distraction as I am, I want to make those changes right away.

Now which is easier: opening up a separate CSS file to modify the page's style sheet, or just typing those style properties into the HTML code I'm already editing?

Like behaving yourself at work functions, keeping the types of code you write separate from one another takes discipline. But once you understand the benefits, you too will be able to summon the willpower it takes to stay on the straight and narrow.

Three Layers

Keeping different kinds of code as separate as possible is a good idea in any kind of programming. It makes it easier to reuse portions of that code in future projects, it reduces the amount of duplicate code you end up writing, and it makes it easier to find and fix problems months and years later.

When it comes to the Web, there's one more reason to keep your code separate: it lets you cater for the many different ways in which people access web pages.

Depending on your audience, the majority of your visitors may use well-appointed desktop browsers with cutting-edge CSS and JavaScript support, but many might be subject to corporate IT policies that force them to use older browsers, or to browse with certain features (like JavaScript) disabled.

Visually impaired users often browse using screen reader or screen magnifier software, and for these users your slick visual design can be more of a hindrance than a help.

Some users won't even *visit* your site, preferring to read content feeds in RSS or similar formats if you offer them. When it comes time to build these feeds, you'll want to be able to send your HTML content to these users without any JavaScript or CSS junk.

The key to accommodating the broadest possible range of visitors to your site is to think of the Web in terms of **three layers**, which conveniently correspond to the three kinds of code I mentioned earlier. These layers are illustrated in Figure 1.2.

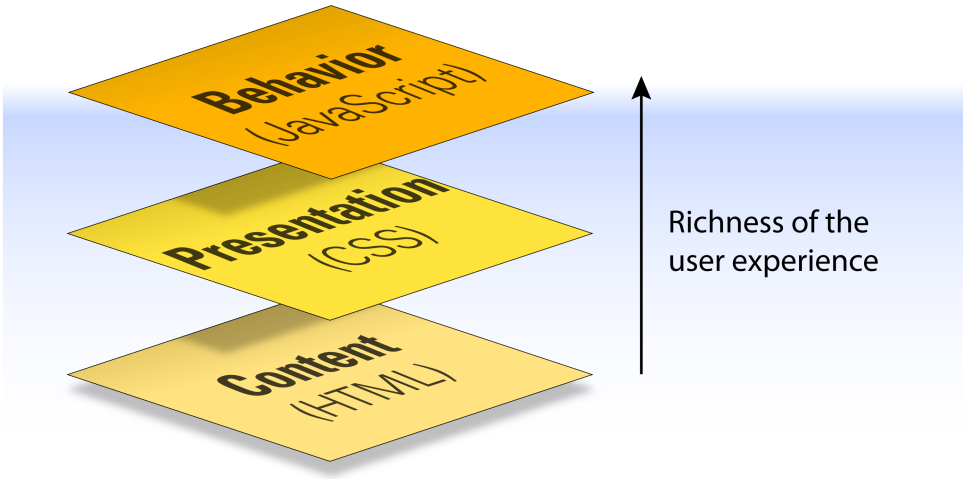


Figure 1.2. The three layers of the Web

When building a site, we work through these layers from the bottom up:

1. We start by producing the **content** in HTML format. This is the base layer, which any visitor using any kind of browser should be able to view.

2. With that done, we can focus on making the site look better, by adding a layer of **presentation** information using CSS. The site will now look good to users able to display CSS styles.

3. Lastly, we can use JavaScript to introduce an added layer of interactivity and dynamic **behavior**, which will make the site easier to use in browsers equipped with JavaScript.

If we keep the HTML, CSS, and JavaScript code separate, we'll find it much easier to make sure that the content layer remains readable in browsing environments where the presentation and/or behavior layers are unable to operate. This "start at the bottom" approach to web design is known in the trade as **progressive enhancement**.

Let's look at each of these layers in isolation to see how we can best maintain this separation of code.

HTML for Content

Everything that's needed to read and understand the content of a web page belongs in the HTML code for that page—nothing more, nothing less. It's that simple. Web designers get into trouble when they forget the K.I.S.S. principle,[3] and cram non-content information into their HTML code, or alternatively move some of the page's content into the CSS or JavaScript code for the page.

A common example of non-content information that's crammed into pages is **presentational HTML**—HTML code that describes how the content should *look* when it's displayed in the browser. This can include old-fashioned HTML tags like `<b>`, `<i>`, `<u>`, `<tt>`, and `<font>`:

```
<p>Whatever you do, <a href="666.html"><font color="red">don't
  click this link</font></a>!</p>
```

It can take the form of inline CSS applied with the `style` attribute:

```
<p>Whatever you do, <a href="666.html" style="color: red;">don't
  click this link</a>!</p>
```

It can also include the secret shame of many well-intentioned web designers—CSS styles applied with presentational class names:

```
<p>Whatever you do, <a href="666.html" class="red">don't click
  this link</a>!</p>
```

Presentational Class Names?

If that last example looks okay to you, you're not alone, but it's definitely bad mojo. If you later decide you want that link to be yellow, you're either stuck updating both the class name and the CSS styles that apply to it, or living with the embarrassment of a class named "red" that is actually styled yellow. *That'll* turn your face yellow—er, red!

[3] Keep It Simple, Stupid.

Rather than embedding presentation information in your HTML code, you should focus on the *reason* for the action—for example, you want a link to be displayed in a different color. Is the link especially important? Consider surrounding it with a tag that describes the emphasis you want to give it:

```
<p>Whatever you do, <em><a href="evil.html">don't click this
  link</a></em>!</p>
```

Is the link a warning? HTML doesn't have a tag to describe a warning, but you could choose a CSS class name that conveys this information:

```
<p>Whatever you do, <a href="evil.html" class="warning">don't
  click this link</a>!</p>
```

You can take this approach too far, of course. Some designers mistake tags like `<h1>` as presentational, and attempt to remove this presentational code from their HTML:

```
<p class="heading">A heading with an identity crisis</p>
```

Really, the presentational information that you should keep out of your document is the font, size, and color in which a heading is to be displayed. The fact that a piece of text *is* a heading is part of the content, and as such should be reflected in the HTML code. So this code is perfectly fine:

```
<h1>A heading at peace with itself</h1>
```

In short, your HTML should do everything it can to convey the meaning, or **semantics** of the content in the page, while steering clear of describing how it should look. Web standards geeks call HTML code that does this **semantic markup**.

Writing semantic markup allows your HTML files to stand on their own as meaningful documents. People who, for whatever reason, cannot read these documents by viewing them in a typical desktop web browser will be better able to make sense of them this way. Visually impaired users, for example, will be able to use assistive software like screen readers to listen to the page as it's read aloud, and the more clearly your HTML code describes the content's meaning, the more sense tools like these will be able to make of it.

Best of all, however, semantic markup lets you apply new styles (presentation) and interactive features (behavior) without having to make many (or, in some cases, any!) changes to your HTML code.

CSS for Presentation

Obviously, if the content of a page should be entirely contained within its HTML code, its style—or presentation—should be fully described in the CSS code that's applied to the page.

With all the work you've done to keep your HTML free of presentational code and rich with semantics, it would be a shame to mess up that file by filling it with snippets of CSS.

As you probably know, CSS styles can be applied to your pages in three ways:

inline styles

```
<a href="evil.html" style="color: red;">
```

Inline styles are tempting for the reasons I explained earlier: you can apply styles to your content as you create it, without having to switch gears and edit a separate style sheet. But as we saw in the previous section, you'll want to avoid inline styles like the plague if you want to keep your HTML code meaningful to those who cannot see the styles.

embedded styles

```
<style type="text/css">
  .warning {
    color: red;
  }
</style>
⋮
<a href="evil.html" class="warning">
```

Embedded styles keep your markup clean, but tie your styles to a single document. In most cases, you'll want to share your styles across multiple pages on your site, so it's best to steer clear of this approach as well.

external styles

```
<link rel="stylesheet" href="styles.css" />
⋮
<a href="evil.html" class="warning">
```

styles.css

```
.warning {
  color: red;
}
```

External styles are really the way to go, because they let you share your styles between multiple documents, they reduce the amount of code browsers need to download, and they also let you modify the look of your site without having to get your hands dirty editing HTML.

But you knew all that, right? This is a JavaScript book, after all, so let's talk about the JavaScript that goes into your pages.

JavaScript for Behavior

As with CSS, you can add JavaScript to your web pages in a number of ways:

- You can embed JavaScript code directly in your HTML content:

```
<a href="evil.html" onclick="JavaScript code here">
```

- You can include JavaScript code at the top of your HTML document in a `<script>` tag:

```
<script type="text/javascript"><!--//--><![CDATA[//><!--
  JavaScript code here
//--><!]]></script>
⋮
<a href="evil.html" class="warning">
```

CDATA?

If you're wondering what all that gobbledygook is following the `<script>` tag and preceding the `</script>` tag, that's what it takes to legitimately embed JavaScript in an XHTML document without confusing web browsers that don't understand XHTML (like Internet Explorer).

If you write your page with HTML instead of XHTML, you can get away with this much simpler syntax:

```
<script type="text/javascript">
  JavaScript code here
</script>
```

- You can put your JavaScript code in a separate file, then link to that file from as many HTML documents as you like:

```
<script type="text/javascript" src="script.js"></script>
⋮
<a href="evil.html" class="warning">
```

script.js *(excerpt)*

```
JavaScript code here
```

Guess which method you should use.

Writing JavaScript that enhances usability without cluttering up the HTML document(s) it is applied to, without locking out users that have JavaScript disabled in their browsers, and without interfering with *other* JavaScript code that might be applied to the same page, is called **unobtrusive scripting**.

Unfortunately, while many professional web developers have clued in to the benefits of keeping their CSS code in separate files, there is still a lot of JavaScript code mixed into HTML out there. By showing you the *right* way to use JavaScript in this book, we hope to help change that.

The Right Way

So, how much does all this stuff really matter? After all, people have been building web sites with HTML, CSS, and JavaScript mixed together for years, and for the majority of people browsing the Web, those sites have worked.

Well, as you come to learn JavaScript, it's arguably more important to get it right than ever before. JavaScript is by far the most powerful of the three languages that you'll use to design web sites, and as such it gives you unprecedented freedom to completely mess things up.

As an example, if you really, really like JavaScript, you could go so far as to put everything—content, presentation, and behavior—into your JavaScript code. I've actually seen this done, and it's not pretty—especially when a browser with JavaScript disabled comes along.

Even more telling is the fact that JavaScript is the only one of these three languages that has the ability to hang the browser, making it unresponsive to the user.[4]

Therefore, through the rest of this book, we'll do our darnedest to show you the right way to use JavaScript, not just because it keeps your code tidy, but because it helps to keep the Web working the way it's meant to—by making content accessible to as many people as possible, no matter which web browser they choose to use.

JavaScript Libraries

As I mentioned, one of the benefits of keeping different kinds of code separate is that it makes it easier to take code that you've written for one site and reuse it on another. Certain JavaScript maniacs (to be referred to from this point on as "people") have taken the time to assemble vast **libraries** of useful, unobtrusive JavaScript code that you can download and use on your own web sites for free.

Throughout this book, we'll build each of the examples from scratch—all of the JavaScript code you need can be found right here in these pages. Since there isn't always time to do this in the real world, however, and because libraries are quickly

[4] We'll show you an example of this in Chapter 7.

becoming an important part of the JavaScript landscape, we'll also look at how the popular JavaScript libraries do things whenever the opportunity presents itself.

Here are the libraries that we'll use in this book:

Prototype	http://www.prototypejs.org/
script.aculo.us	http://script.aculo.us/
Yahoo! User Interface Library (YUI)	http://developer.yahoo.com/yui/
Dojo	http://dojotoolkit.org/
jQuery	http://jquery.com/
MooTools	http://mootools.net/

Not All Libraries are Created Equal

Watch out for sites offering snippets of JavaScript code for you to copy and paste into your web pages to achieve a particular effect. There is a lot of free code out there, but not all of it is good.

In general, the good libraries come in the form of JavaScript (**.js**) files that you can link into your pages unobtrusively, instead of pasting JavaScript directly into your HTML code.

If you don't feel confident to judge whether a particular JavaScript library is good or bad, ask for some advice in the SitePoint Forums,[5] or just stick with the libraries mentioned in this book—they're all very good.

Let's Get Started!

Enough preaching—you picked up this book to learn JavaScript, right? (If you didn't, I'm afraid you're in for a bit of a disappointment.) Clean HTML and CSS are nice and all, but it's time to take the plunge into the third layer of the Web: behavior.

Turn the page, and get ready to start using some cool (and unobtrusive) JavaScript.

[5] http://www.sitepoint.com/forums/

Chapter 2

Programming with JavaScript

Programming is all about speaking the language of computers. If you're a robot, this should be pretty easy for you, but if you're unlucky enough to be a human, it might take a bit of adjustment.

If you want to learn how to program, there are really two things you have to get your head around. First, you have to think about reducing one big problem into small, digestible chunks that are just right for a computer to crunch. Second, you have to know how to translate those chunks into a language that the computer understands.

I find that the second part—the **syntax**—gradually becomes second nature (much like when you learn a *real* second language), and experienced programmers have very little trouble switching between different languages (like JavaScript, PHP, Ruby, or Algol 60). Most of the thought in programming is focused on the first part—thinking about how you can break down a problem so that the computer can solve it.

By the time you've finished this book, you'll understand most of the syntax that JavaScript has to offer, but you'll continue learning new ways to solve programming

problems for as long as you continue to program. We'll tell you how to solve quite a few problems in this book, but there are always different ways to achieve a given task, and there will always be new problems to solve, so don't think that your learning will stop on the last page of this book.

Running a JavaScript Program

Before you even start writing your first JavaScript program, you'll have to know how to run it.

Every JavaScript program designed to run in a browser has to be attached to a document. Most of the time this will be an HTML or XHTML document, but exciting new uses for JavaScript emerge every day, and in the future you might find yourself using JavaScript on XML, SVG, or something else that we haven't even thought of yet. We're just going to worry about HTML in this book, because that's what 99% of people use JavaScript with.

To include some JavaScript on an HTML page, we have to include a `<script>` tag inside the `head` of the document. A script doesn't necessarily have to be JavaScript, so we need to tell the browser what type of script we're including by adding a `type` attribute with a value of `text/javascript`:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
<html lang="en-US">
  <head>
    <title>The Running Man</title>
    <meta http-equiv="Content-Type"
        content="text/html; charset=utf-8">

    <script type="text/javascript">
    </script>

  </head>
</html>
```

You can put as much JavaScript code as you want inside that `<script>` tag—the browser will execute it as soon as it has been downloaded:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
<html lang="en-US">
  <head>
    <title>The Running Man</title>
    <meta http-equiv="Content-Type"
        content="text/html; charset=utf-8">

    <script type="text/javascript">
      alert("Arnie says hi!");
    </script>

  </head>
</html>
```

XHTML and Embedded JavaScript don't Mix

For this one example, we've switched from an XHTML DOCTYPE to an HTML DOCTYPE. As mentioned in Chapter 1, embedding JavaScript in XHTML requires gobbledygook that few mortals can remember:

```
<script type="text/javascript"><!--//--><![CDATA[//><!--
  alert("Arnie says hi!");
//--><!]]></script>
```

For many, this is reason enough to avoid embedded JavaScript.

Even though it's nice and easy to just type some JavaScript straight into your HTML code, it's preferable to include your JavaScript in an external file. This approach provides several advantages:

- It maintains the separation between content and behavior (HTML and JavaScript).
- It makes it easier to maintain your web pages.
- It allows you to easily reuse the same JavaScript programs on different pages of your site.

To reference an external JavaScript file, you need to use the `src` attribute on the `<script>` tag:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en-US">
  <head>
    <title>The Running Man</title>
    <meta http-equiv="Content-Type"
        content="text/html; charset=utf-8" />

    <script type="text/javascript" src="example.js"></script>

  </head>
</html>
```

Any JavaScript that you might have included between your `<script>` and `</script>` tags can now be put into that external file and the browser will download the file and run the code.

The file can be called whatever you want, but the common practice is to include a **.js** extension at the end of it.

If you'd like to try out the little program above, create a new HTML document (or open the closest one to hand) and insert the `<script>` tag inside the `head`. Once you've done that, put this snippet into a file called **example.js** in the same directory or folder:

```
alert("Arnie says hi!");
```

Now, open the HTML file in your browser, and see what happens! As you read through the rest of this chapter, you can replace the contents of **example.js** with each of the simple programs that I'll show you, and try them for yourself!

Absolute URLs Work Too

As with the `src` attribute of an image, you can reference a file anywhere on your server, or anyone else's server:

```
<script type="text/javascript"
    src="http://www.example.com/script.js"></script>
```

It's possible to include as many external scripts on your page as you want:

```
<script type="text/javascript" src="library.js"></script>
<script type="text/javascript" src="more.js"></script>
<script type="text/javascript" src="example.js"></script>
```

This capability is what makes JavaScript libraries, where you include a standard library file on your page alongside other code that uses the contents of that library, possible.

Every time you load a page with JavaScript on it, the browser will interpret all of the included JavaScript code and figure out what to do with it. If you've loaded a page into your browser, and then you make some changes to that page's JavaScript (either on the page itself or in an external file), you'll need to refresh the page before those changes will be picked up by the browser.

Statements: Bite-sized Chunks for your Browser

So now you know how to tell the browser that it needs to run some JavaScript, but you don't know any JavaScript for it to run. We'd better fix that!

Earlier, we were talking about reducing a problem into steps that a computer can understand. Each small step you take in a program is called a **statement**, and it tells the browser to perform an action. By building up a series of these actions, we create a **program**. Statements are to programs as sentences are to books.

In JavaScript each statement has to be separated by a new line or a semicolon. So, two statements could be written like this:

```
Statement one
Statement 2.0
```

Or they could be written like this:

```
Statement one;Statement 2.0;
```

It is generally considered best practice, however, to do both—separate statements by a semicolon *and* a new line:

```
Statement one;
Statement 2.0;
```

This way, each of your statements will be easy to read, and you'll have removed the potential for any ambiguity that might occur if two statements accidentally run together.

There's a whole bunch of different tasks you can achieve inside each statement; the first one that we'll look at shortly is creating variables.

Comments: Bite-sized Chunks Just for You

If you follow the advice in this book and keep your JavaScript code simple and well structured, you should be able to get the gist of how it works just by looking at it. Every once in a while, however, you'll find yourself crafting a particularly tricky segment of code, or some esoteric browser compatibility issue will force you to insert a statement that might seem like nonsense if you had to come back and work on the program later. In situations like these, you may want to insert a comment.

A **comment** is a note in your code that browsers will ignore completely. Unlike the rest of the code you write, comments are there to be read by *you* (or other programmers who might later need to work on your code). In general, they explain the surrounding code, making it easier to update the program in future.

JavaScript supports two types of comments. The first is a single-line comment, which begins with two slashes (`//`) and runs to the end of the line:

```
Statement one; // I'm especially proud of this one
Statement 2.0;
```

As soon as the browser sees two slashes, it closes its eyes and sings a little song to itself until it reaches the end of the line, after which it continues to read the program as usual.

If you need to write a more sizable comment, you can use a multi-line comment, starting with `/*` and ending with `*/`:

```
/* This is my first JavaScript program. Please forgive any
   mistakes you might find here.
   If you have any suggestions, write to nOOb@example.com. */
Statement one; // I'm especially proud of this one
Statement 2.0;
```

You'll notice a distinct lack of comments in the code presented in this book. The main reason for this is that all of the code is explained in the surrounding text, so why not save a few trees? In real-world programs, you should always include a comment if you suspect that you might not understand a piece of code when you return to work on it later.

Variables: Storing Data for your Program

It's possible to write a program that defines the value of every single piece of data it uses, but that's like driving a ski lift—you don't really get to choose where you're going. If you want your program to be able to take user input, and adapt to different pages and situations, you have to have some way of working with values that you don't know in advance.

As with most programming concepts, it's very useful at this point to think of your computer as a BGC (Big, Giant Calculator). You know where you are with a calculator, so it makes programming a bit easier to understand.

Now, we could write a program for a calculator that said:

```
4 + 2
```

But every time we run that program, we're going to get exactly the same answer. There's no way that we can substitute the values in the equation for something else—values from another calculation, data from a file, or even user input.

If we want the program to be a bit more flexible, we need to abstract some of its components. Take a look at the equation above and ask yourself, "What does it really do?"

It adds two numbers.

If we're getting those numbers when we *run* the program, we don't know what they'll be when we *write* the program, so we need some way of referring to them without using actual numbers. How about we give them names? Say … "x" and "y."

Using those names, we could rewrite the program as:

```
x + y
```

Then, when we get our data values from some faraway place, we just need to make sure it's called `x` and `y`. Once we've done that, we've got **variables**.

Variables allow us to give a piece of data a name, then reference that data by its name further along in our program. This way, we can reuse a piece of data without having to remember what its actual value was; all we have to do is remember a variable name.

In JavaScript, we create a variable by using the keyword `var` and specifying the name we want to use:

```
var chameleon;
```

This is called **declaring** a variable.

Having been declared, `chameleon` is ready to have some data assigned to it. We can do this using the **assignment operator** (=), placing the variable name on the left and the data on the right:

```
var chameleon;
chameleon = "blue";
```

This whole process can be shortened by declaring and assigning the variable in one go:

```
var chameleon = "blue";
```

In practice, this is what most JavaScript programmers do—declare a variable whenever that variable is first assigned some data.

If you've never referenced a particular variable name before, you can actually assign that variable without declaring it using `var`:

```
chameleon = "blue";
```

The JavaScript interpreter will detect that this variable hasn't been declared before, and will automatically declare it when you try to assign a value to it. At first glance, this statement seems to do exactly the same thing as using the `var` keyword; however, the variable that it declares is actually quite different, as we'll see later in this chapter when we discuss functions and scoping. For now, take it from me—it's always safest to use `var`.

The `var` keyword has to be used only when you first declare a variable. If you want to change the value of the variable later, you do so without `var`:

```
var chameleon = "blue";
⋮
chameleon = "red";
```

You can use the value of a variable just by calling its name. Any occurrence of the variable name will automatically be replaced with its value when the program is run:

```
var chameleon = "blue";
alert(chameleon);
```

The second statement in this program tells your browser to display an alert box with the supplied value, which in this case will be the value of the variable `chameleon`, as shown in Figure 2.1.

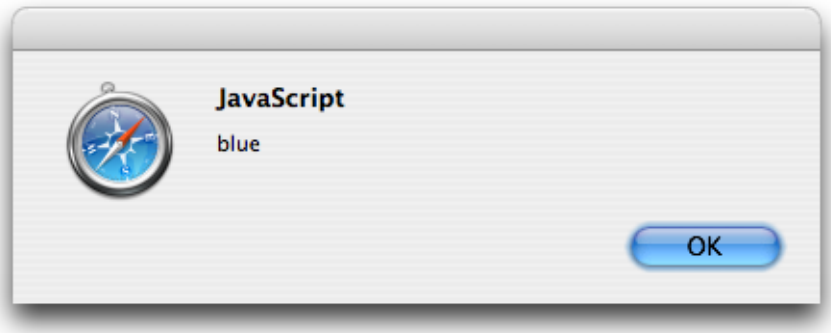


Figure 2.1. JavaScript replacing the variable name with its value

Your variable names can comprise almost any combination of letters and numbers, though no spaces are allowed. Most punctuation and symbols have special meaning inside JavaScript, so the dollar sign (`$`) and the underscore (`_`) are the only non-alphanumeric characters allowed in variable names. Variable names are also case-sensitive, so the names `chameleon`, `Chameleon`, and `CHAMELEON` refer to unique variables that could exist simultaneously.

Given those rules, these are all acceptable variable declarations:

```
var chameleon = "blue";
var Chameleon = "red";
var CHAMELEON = "green";
var yellow_chameleon = "yellow";
var orangeChameleon = "orange";
var chameleon$ = "greedy";
```

It's standard practice to create variable names in lowercase letters, unless you're concatenating more than one word. And as I mentioned, variable names can't have spaces in them, so if you want a variable name to include more than one word, you can separate each word with an underscore (`multi_word_variable`) or capitalize the first letter of each word except for the first (`multiWordVariable`)—an approach called **camel casing**, because the name has humps like a camel (if you squint your eyes and tilt your head slightly … kind of).

The approach you use to name variables really comes down to personal preference, and which name style you find more readable. I use camel casing because some long-forgotten lecturer beat it into me with a big plank.

Variable Types: Different Types for Different Data

A lot of programming languages feature **strictly typed** variables. With these, you have to tell the program what type of data the variable is going to hold when it's declared, and you can't change a variable's type once it has been created.

JavaScript, however, is **loosely typed**—the language doesn't care *what* your variables hold. A variable could start off holding a number, then change to holding a character, a word, or anything else you want it to hold.

Even though you don't have to declare the data type up front, it's still vital to know what types of data a variable can store, so that you can use and manipulate them properly inside your own programs. In JavaScript, you can work with numbers, strings, Booleans, arrays and objects. We'll take a look at the first four of these types now, but you'll have to wait till the end of the chapter to read about objects, because they're a bit trickier.

Numbers

Eventually, everything inside a computer is broken down into numbers (see the Big Giant Calculator theory we explored earlier). Numbers in JavaScript come in two flavors: whole numbers and decimals. The technical term for a whole number is an **integer** or **int**. A decimal is called a **floating point number**, or **float**. These terms are used in most programming languages, including JavaScript.

To create a variable with numerical data, you just assign a number to a variable name:

```
var whole = 3;
var decimal = 3.14159265;
```

Floating point numbers can have as many decimal places as you want:

```
var shortDecimal = 3.1;
var longDecimal = 3.14159265358979323846264338327950288419716939937;
```

And both floats and integers can have negative values if you place a minus sign (-) in front of them:

```
var negativeInt = -3;
var negativeFloat = -3.14159265;
```

Mathematical Operations

Numbers can be combined with all of the mathematical operations you'd expect: addition (`+`), subtraction (`-`), multiplication (`*`), and division (`/`). They're written in fairly natural notation:

```
var addition = 4 + 6;
var subtraction = 6 – 4;
var multiplication = 5 * 9;
var division = 100 / 10;
var longEquation = 4 + 6 + 5 * 9 – 100 / 10;
```

The symbols that invoke these operations in JavaScript—`+`, `-`, `*`, and `/`—are called **operators**, and as we'll see through the rest of this chapter, JavaScript has a lot of them!

In a compound equation like the one assigned to `longEquation`, each of the operations is subject to standard mathematical precedence (that is, multiplication and division operations are calculated first, from left to right, after which the addition and subtraction operations are calculated from left to right).

If you want to override the standard precedence of these operations, you can use brackets, just like you learned in school. Any operations that occur inside brackets will be calculated before any multiplication or division is done:

```
var unbracketed = 4 + 6 * 5;
var bracketed = (4 + 6) * 5;
```

Here, the value of `unbracketed` will be `34`, because `6 * 5` is calculated first. The value of `bracketed` will be `50`, because `(4 + 6)` is calculated first.

You can freely combine integers and floats in your calculations, but the result will always be a float:

```
var whole = 3;
var decimal = 3.14159265;
var decimal2 = decimal – whole;
var decimal3 = whole * decimal;
```

`decimal2` now equals `0.14159265` and `decimal3` equals `9.42477795`.

If you divide two integers and the result is not a whole number, it will automatically become a float:

```
var decimal = 5 / 4;
```

The value of `decimal` will be `1.25`.

Calculations can also involve any combination of numbers or numerical variables:

```
var dozen = 12;
var halfDozen = dozen / 2;
var fullDozen = halfDozen + halfDozen;
```

A handy feature of JavaScript is the fact that you can refer to the current value of a variable in describing a new value to be assigned to it. This capability lets you do things like increase a variable's value by one:

```
var age = 26;
age = age + 1;
```

In the second of these statements, the `age` reference on the right uses the value of `age` *before* the calculation; the result of the calculation is then assigned to `age`, which ends up being `27`. This means you can keep calculating new values for the same variable without having to create temporary variables to store the results of those calculations.

The program above can actually be shortened using the handy `+=` operator, which tells your program to add and assign in one fell swoop:

```
var age = 26;
age += 1;
```

Now, `age` will again equal `27`.

It turns out that adding `1` to a variable is something that happens quite frequently in programming (you'll see why when we get to loops later in this chapter), and there's an even shorter shortcut for adding `1` to a variable:

```
var age = 26;
age++;
```

By adding the special `++` operator to the end of `age`, we tell the program to increment the value of `age` by `1` and assign the result of this operation as the new value. After those calculations, `age` again equals `27`.

Before or After?

As an alternative to placing the increment operator at the end of a variable name, you can also place it at the beginning:

```
var age = 26;
++age;
```

This achieves exactly the same end result, with one subtle difference in the processing: the value of `age` is incremented *before* the variable's value is read. This has no effect in the code above, because we're not using the variable's value there, but consider this code:

```
var age = 26;
var ageCopy = age++;
```

Here, `ageCopy` will equal `26`. Now consider this:

```
var age = 26;
var ageCopy = ++age;
```

In this code, `ageCopy` will equal `27`.

Due to the possible confusion arising from this situation, the tasks of incrementing a variable and reading its value are not often completed in a single step. It's safer to increment and assign variables separately.

As well as these special incrementing operators, JavaScript also has the corresponding decrementing operators, `-=` and `--`:

```
var age = 26;
age -= 8;
```

Now `age` will be `18`, but let's imagine we just wanted to decrease it by one:

```
var age = 26;
age--;
```

`age` will now be `25`.

You can also perform quick assignment multiplication and division using `*=` and `/=`, but these operators are far less common.

Strings

A string is a series of characters of any length, from zero to infinity (or as many as you can type in your lifetime; ready … set … go!). Those characters could be letters, numbers, symbols, punctuation marks, or spaces—basically anything you can find on your keyboard.

To specify a string, we surround a series of characters with quote marks. These can either be single or double straight quote marks,[1] just as long as the opening quote mark matches the closing quote mark:

```
var single = 'Just single quotes';
var double = "Just double quotes";
var crazyNumbers = "18 crazy numb3r5";
var crazyPunctuation = '~cr@zy_punctu&t!on';
```

The quote marks don't appear in the value of the string, they just mark its boundaries. You can prove this to yourself by putting the following code into a test JavaScript file:

[1] Some text editors will let you insert curly quotes around a string, "like this." JavaScript will not recognize strings surrounded by curly quotes; it only recognizes straight quotes, "like this."

```
var single = 'Just single quotes';
alert(single);
```

When you load the HTML page that this file's attached to, you'll see the alert shown in Figure 2.2.

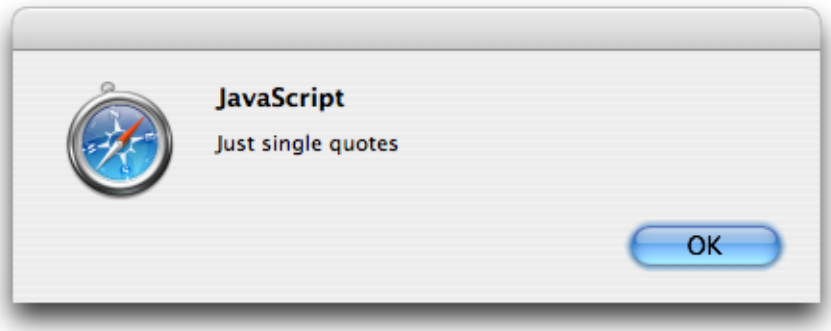


Figure 2.2. The string's value displaying without the quotes used to create the string

It's okay to include a single quote inside a double-quoted string, or a double quote inside a single-quoted string, but if you want to include a quote mark inside a string that's quoted with the same mark, you must precede the internal quote marks with a backslash (\). This is called **escaping** the quote marks:

```
var singleEscape = 'He said \'RUN\' ever so softly.';
var doubleEscape = "She said \"hide\" in a loud voice.";
```

Don't worry—those backslashes disappear when the string is actually used. Let's put this code into a test JavaScript file:

```
var doubleEscape = "She said \"hide\" in a loud voice.";
alert(doubleEscape);
```

When you load the HTML page the file's attached to, you'll see the alert box shown in Figure 2.3.

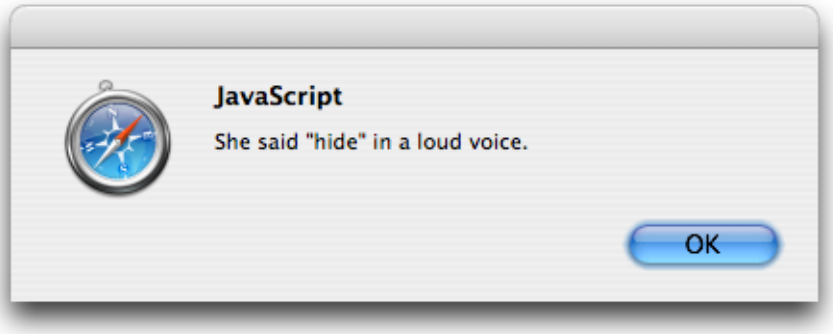


Figure 2.3. The string's value displaying without the backslashes used to escape quote marks in the string

It doesn't matter whether you use single or double quotes for your strings—it's just a matter of personal preference. I tend to use double quotes, but if I'm creating a string with a lot of double quotes in it (such as HTML code), I'll switch to using single quotes around that string, just so I don't have to escape all the double quotes it contains.

String Operations

We can't perform as many operations on strings as we can on numbers, but a couple of very useful operators are available to us.

If you'd like to add two strings together, or **concatenate** them, you use the same + operator that you use for numbers:

```
var complete = "com" + "plete";
```

The value of `complete` will now be `"complete"`.

Again, you can use a combination of strings and string variables with the + operator:

```
var name = "Slim Shady";
var sentence = "My name is " + name;
```

The value of `sentence` will be `"My name is Slim Shady"`.

You can use the += operator with strings, but not the ++ operator—it doesn't make sense to increment strings. So the previous set of statements could be rewritten as:

```
var name = "Slim Shady";
var sentence = "My name is ";
sentence += name;
```

There's one last trick to concatenating strings: you can concatenate numbers and strings, but the result will always end up being a string. If you try to add a number to a string, JavaScript will automatically convert the number into a string, then concatenate the two resulting strings:

```
var sentence = "You are " + 1337
```

`sentence` now contains `"You are 1337"`. Use this trick when you want to output sentences for your h4x0r friends.

Booleans

Boolean values are fairly simple, really—they can be either `true` or `false`. It's probably easiest to think of a Boolean value as a switch that can either be on or off. They're used mainly when we're making decisions, as we'll see in a few pages time.

In order to assign a Boolean value to a variable, you simply specify which state you want it to be in. `true` and `false` are keywords in JavaScript, so you don't need to put any quote marks around them:

```
var lying = true;
var truthful = false;
```

If you were to surround the keywords in quote marks, they'd just be normal strings, not Boolean values.

Arrays

Numbers, strings and Booleans are good ways to store individual pieces of data, but what happens when you have a group of data values that you want to work with, like a list of names or a series of numbers? You could create a whole bunch of variables, but they still wouldn't be grouped together, and you'd have a hard time keeping track of them all.

Arrays solve this problem by providing you with an ordered structure for storing a group of values. You can think of an array as being like a rack in which each slot is able to hold a distinct value.

In order to create an array, we use the special array markers, which are the opening and closing square brackets:

```
var rack = [];
```

The variable `rack` is now an array, but there's nothing stored in it.

Each "slot" in an array is actually called an **element**, and in order to put some data into an element you have to correctly reference which element you want to put it in. This reference is called an **index**, which is a number that represents an element's position in an array. The first element in an array has an index of 0, which can be a little confusing at first, but it's just a programming quirk you have to get used to. The second element has an index of 1, the third: 2, and so on.

To reference a particular element, we use the variable name, followed by an opening square bracket, then the index and a closing square bracket, like this:

```
var rack = [];
rack[0] = "First";
rack[1] = "Second";
```

With that data in the array, you could imagine it looking like Figure 2.4.

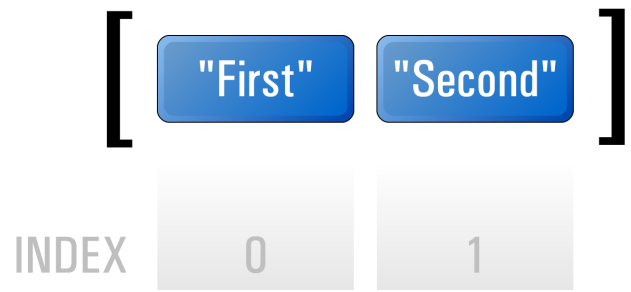


Figure 2.4. An array storing data sequentially, with an index for each element, starting at 0

When we want to retrieve a particular element, we use the array-index notation just like a normal variable name. So, if we had an array like the one above, we could create an alert box displaying the value of the second element like this:

```
alert(rack[1]);
```

The resulting alert is shown in Figure 2.5.

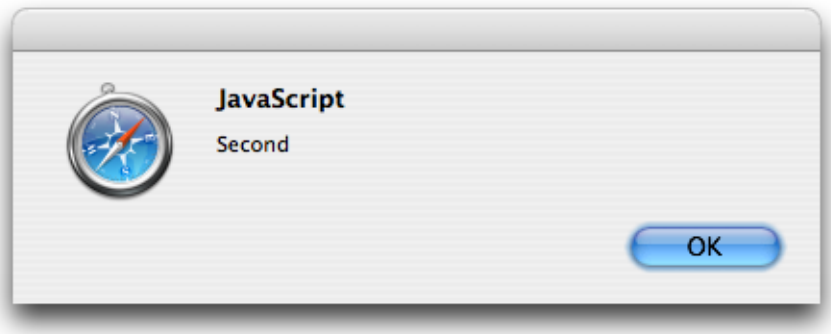


Figure 2.5. An alert box displaying a value retrieved from an array

It's possible to populate an array when it's declared. We simply insert values, separated with commas, between the square brackets:

```
var rack = ["First", "Second", "Third", "Fourth"];
```

That statement says that we should create an array—`rack`—that has four elements with the values specified here. The first value will have an index of 0, the second value an index of 1, and so on. The array that's created will look like Figure 2.6.

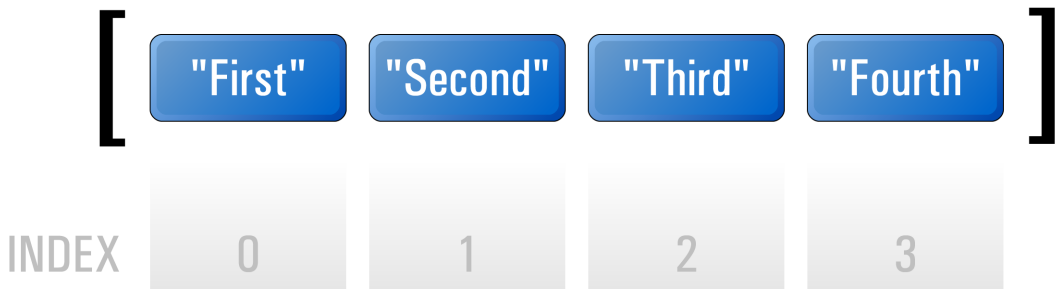


Figure 2.6. The resulting array

Arrays can contain any data type—not just strings—so you could have an array of numbers:

```
var numberArray = [1, 2, 3, 5, 8, 13, 21, 34];
```

You might have an array of strings:

```
var stringArray = ["Veni", "Vidi", "Vici"];
```

A mixed array, containing multiple data types, would look like this:

```
var mixedArray = [235, "Parramatta", "Road"];
```

Here's an array of arrays:

```
var subArray1 = ["Paris", "Lyon", "Nice"];
var subArray2 = ["Amsterdam", "Eindhoven", "Utrecht"];
var subArray3 = ["Madrid", "Barcelona", "Seville"];

var superArray = [subArray1, subArray2, subArray3];
```

That last example is what we call a **multi-dimensional array**—it's a two-dimensional array, to be precise—and it's useful if you want to create a group of groups. In order to retrieve a value from one of the sub-arrays, you have to reference two indices, like so:

```
var city = superArray[0][2];
```

If we translate that statement, starting from the right side, it says:

`[2]` Get the third element …

`[0]` of the first array …

`superArray` in `superArray` ...

`var city =` and save that value in a new variable, `city`.

It's possible to have arrays of arrays of arrays, and arrays of arrays of arrays of arrays, but as you can probably tell from these descriptions, such arrangements quickly become unmanageable, so two-dimensional arrays are normally as far as you ever need to go.

The last thing to understand about arrays is the fact that a very useful property is attached to them: `length`. Sometimes, you'll be dealing with an unknown array—an array you've obtained from somewhere else—and you won't know how many elements it contains. In order to avoid referencing an element that doesn't exist, you can check the array's `length` to see how many items it actually contains. We perform this check by adding `.length` to the end of the array name:

```
var shortArray = ["First", "Second", "Third"];
var total = shortArray.length;
```

The value of `total` will now be `3` because there are three items in the array `shortArray`.

It's important to note that you can't use `array.length` to get the index of the last item in the array. Because the first item's index is `0`, the last item's index is actually `array.length – 1`:

```
var lastItem = shortArray[shortArray.length – 1];
```

This situation might seem a bit annoying, until you realize that this makes it easy to add an element to the end of the array:

```
shortArray[shortArray.length] = "Fourth";
```

Associative Arrays

Normal arrays are great for holding big buckets of data, but they can sometimes make it difficult to find the exact piece of data you're looking for.

Associative arrays provide a way around this problem—they let you specify key-value pairs. In most respects an associative array is just like an ordinary array, except that instead of the indices being numbers, they're strings, which can be a lot easier to remember and reference:

```
var postcodes = [];
postcodes["Armadale"] = 3143;
postcodes["North Melbourne"] = 3051;
postcodes["Camperdown"] = 2050;
postcodes["Annandale"] = 2038;
```

Now that we've created our associative array, it's not hard to get the postcode for Annandale. All we have to do is specify the right key, and the value will appear:

```
alert(postcodes["Annandale"]);
```

The resulting alert is shown in Figure 2.7.

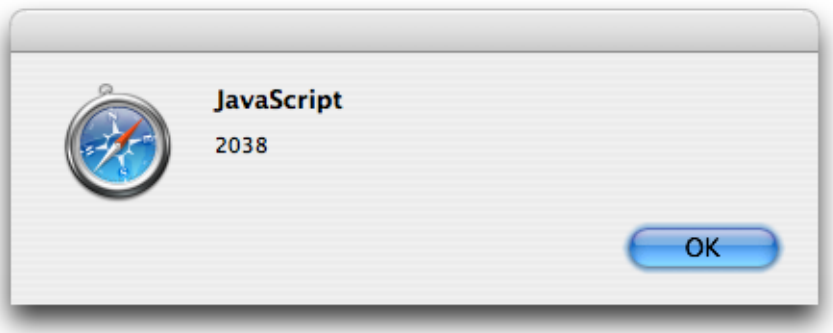


Figure 2.7. Finding a postcode using an associative array

Although the keys for an associative array have to be strings, the values can be of any data type, including other arrays or associative arrays.

Conditions and Loops: Controlling Program Flow

So far, we've seen statements that allow you to set and retrieve variables inside your program. For a program to be really useful, however, it has to be able to make decisions based on the values of those variables.

The way we make those decisions is through the use of special structures called conditions and loops, which help to control which parts of your program will run under particular conditions, and how many times those parts will be run.

Conditions: Making Decisions

If you think of your program as being like a road map, and the browser as a car navigating those roads, you'll realize that the browser has to be able to take different paths depending on where the user wants to go. Although a program might seem like a linear path—one statement following another—**conditional statements** act like intersections, allowing you to change directions on the basis of a given condition.

`if` Statements

The most common conditional statement is an `if` statement. An `if` statement checks a condition, and if that condition is met, allows the program to execute some code. If the condition isn't met, the code is skipped.

The flow of a program through an `if` statement can be visualized as in Figure 2.8.

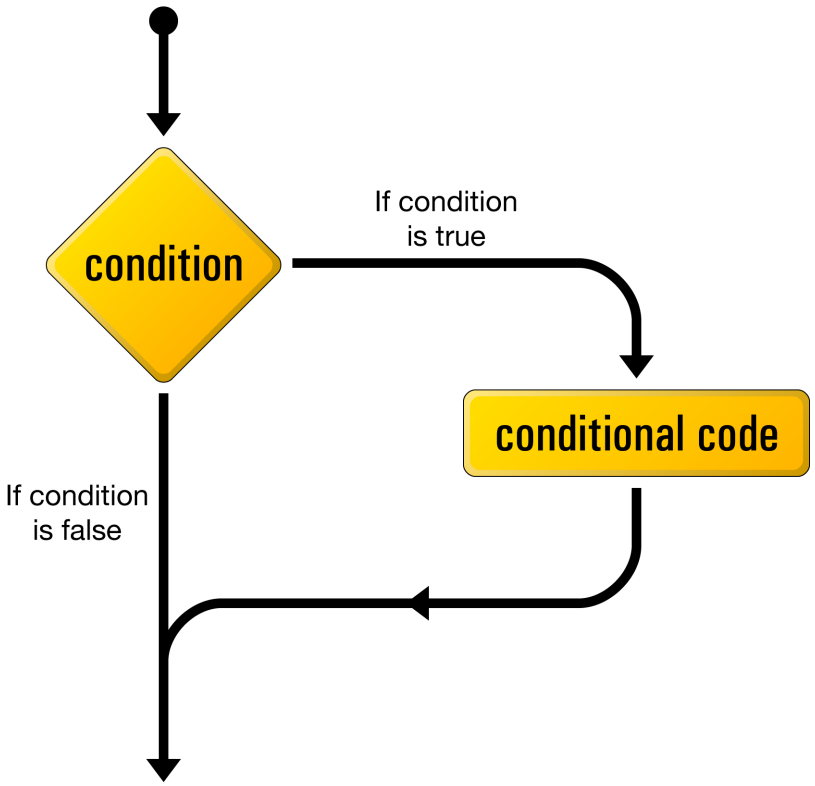


Figure 2.8. The logical flow of an `if` statement

Written as code, `if` statements take this form:

```
if (condition)
{
  conditional code;
}
```

Instead of a semicolon, an `if` statement ends with the conditional code between curly braces (`{…}`).[2] It's considered best practice to put each of these braces on its own line, to make it as easy as possible to spot where blocks of code begin and end.

Indenting Code

It's standard practice to indent the code that's inside curly braces.

On each indented line, a standard number of spaces or tab characters should appear before the first character of the statement. This helps to improve the readability of your code and makes it easier to follow the flow of your programs.

We use two spaces as the standard indentation in this book, but you can use four spaces, one tab—whatever looks best to you. Just be consistent. Every time you nest curly braces (for instance, in another `if` statement inside a block of conditional code), you should increase the indentation for the nested lines by one standard indent.

The condition has to be contained within round brackets (also called parentheses) and will be evaluated as a Boolean, with `true` meaning the code between the curly braces will be executed and `false` indicating it will be skipped. However, the condition doesn't have to be an explicit Boolean value—it can be any **expression** that evaluates to a value that's able to be used as a Boolean.

Expressions

An expression is a combination of values, variable references, operators, and function calls that, when evaluated, produce another value. Wherever a JavaScript value (like a number or a string) is expected, you can use an expression instead.

[2] If the conditional code consists of just one statement, you can choose to omit the curly braces. I find it clearer to always include the braces, which is what we'll do in this book.

Here's a simple expression:

```
4 + 6
```

When evaluated, it produces a value (10). We can write a statement that uses this expression like so:

```
var effect = 4 + 6;
```

We now have in our program a variable called `effect`, with a value of 10.

With conditional statements, the most useful types of expressions are those that use **comparison operators** to test a condition and return a Boolean value indicating its outcome.

You might remember comparison operators such as **greater than** (>) and **less than** (<) from some of your old mathematics classes, but there are also **equality** (==) and **inequality** (!=) operators, and various combinations of these. Basically, each comparison operator compares what's on the left of the operator with what's on the right, then evaluates to `true` or `false`. You can then use that result in a conditional statement like this:

```
var age = 27;

if (age > 20)
{
  alert("Drink to get drunk");
}
```

The greater than and less than operators are really only useful with numbers, because it feels a bit too Zen to ask "is one string greater than another?"

However, the equality operator (==) is useful with both strings and numbers:

```
var age = 27;

if (age == 50){
  alert("Half century");
}
```

```
var name = "Maximus";

if (name == "Maximus")
{
  alert("Good afternoon, General.");
}
```

In the first condition, `age` is `27` and we're testing whether it is equal to `50`; obviously, this condition will evaluate to `false`, so the conditional code will not be run.

In the second condition, `name` is `"Maximus"` and we're testing whether it is equal to `"Maximus"`. This condition will evaluate to `true` and the conditional code will be executed.

== versus =

Be careful to use two equals signs rather than one when you intend to check for equality. If you use only one, you'll be assigning the value on the right to the variable on the left, rather than comparing them, so you'll end up losing your original value rather than checking it!

We can reverse the equality test by using the inequality operator (`!=`):

```
var name = "Decimus";

if (name != "Maximus")
{
  alert("You are not allowed in.");
}
```

Now, because `name` is `"Decimus"` and we're testing whether it *isn't* equal to `"Maximus"` that condition will evaluate to `true` and the conditional code will be run.

Table 2.1 lists the most commonly used comparison operators, and the results they'll return with different values:

Table 2.1. Commonly Used Comparison Operators

Operator	Example	Result
>	A > B	true if A is greater than B
>=	A >= B	true if A is greater than or equal to B
<	A < B	true if A is less than B
<=	A <= B	true if A is less than or equal to B
==	A == B	true if A equals B
!=	A != B	true if A does not equal B
!	!A	true if A's Boolean value is false

Multiple Conditions

Instead of using just one test as a condition, you can create a whole chain of them using the logical operators AND (&&) and OR (||).[3]

Both of these operators may be used to combine conditional tests. The AND operator specifies that *both* tests must evaluate to `true` in order for the whole expression to evaluate to `true`. The OR operator specifies that only one of the tests has to evaluate to `true` in order for the whole expression to evaluate to `true`.

Take a look at this conditional statement:

```
var age = 27;

if (age > 17 && age < 21)
{
  alert("Old enough to vote, too young to drink");
}
```

Here, `age` is greater than `17` but it's not less than `21`, so, since one of the tests evaluated to `false`, the entire condition evaluates to `false`. This is a good way to check if a number falls within a specific range.

On the other hand, the OR operator is good for checking whether a variable matches one of a few values:

[3] That's two vertical bars, not lowercase Ls or number 1s.

```
var sport = "Skydiving";

if (sport == "Bungee jumping" || sport == "Cliff diving" ||
    sport == "Skydiving")
{
  alert("You're extreme!");
}
```

Although the first two tests in this expression evaluate to `false`, `sport` matches the last test in the OR expression, so the whole condition will evaluate to `true`.

`if-else` Statements

An `if` statement allows you to execute some code when a condition is met, but doesn't offer any alternative code for cases when the condition *isn't* met. That's the purpose of the `else` statement.

In an `if-else` statement, you begin just as you would for an `if` statement, but immediately after the closing brace of the `if`, you include an `else`, which specifies code to be executed when the condition of the `if` statement fails:

```
if (condition)
{
  conditional code;
}
else
{
  alternative conditional code;
}
```

The flow of this construct can be visualized as shown in Figure 2.9.

To provide some alternative code, all you have to do is append an `else` statement to the end of the `if`:

```
var name = "Marcus";

if (name == "Maximus")
{
  alert("Good afternoon, General.");
}
```

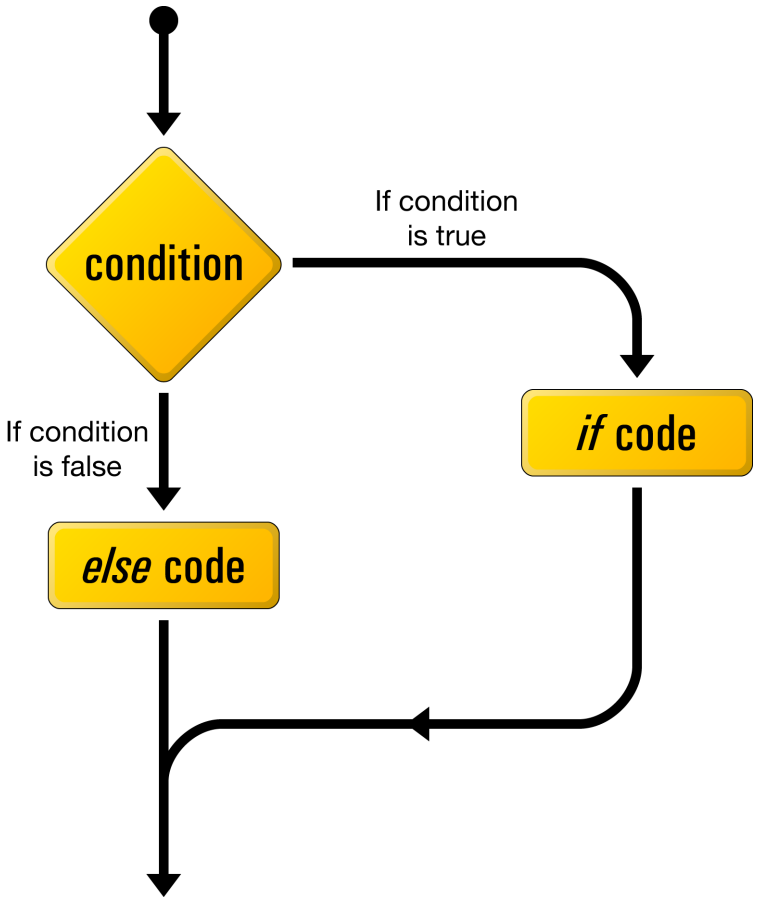


Figure 2.9. The logical flow of an `if-else` statement

```
else
{
  alert("You are not allowed in.");
}
```

This approach saves you from creating a separate `if` statement with a negative formulation of the original condition.

`else-if` Statements

Technically speaking, `else-if` isn't a separate type of statement from `if-else`, but you should be aware of it, because it can be quite useful.

If you want to provide some alternative code for cases in which an `if` statement fails, but you want to further assess the data in order to decide what course of action to take, an `else-if` statement is what you need. Instead of just typing `else`, type `else if`, followed by the extra condition you want to test:

```
var name = "Marcus";

if (name == "Maximus")
{
  alert("Good afternoon, General.");
}
else if (name == "Marcus")
{
  alert("Good afternoon, Emperor.");
}
else
{
  alert("You are not allowed in.");
}
```

You can chain together as many `else-if` statements as you want, and at the end, you can include a normal `else` statement for use when everything fails (though it's not necessary).

Loops: Minimizing Repetition

Computers are meant to make life easier, right? Well, where are those darn robot servants, huh?

Luckily, computers have a few capabilities that will save you thinking and typing time when you're programming. The most effective of these are **loops**, which automate repetitive tasks like modifying each element in an array.

There are a couple of different loop statements but they essentially do the same thing: repeat a set of actions for as long as a specified condition is `true`.

`while` Loops

`while` is the simplest of the loops. All it needs is a condition, and some conditional code:

```
while (condition)
{
  conditional code;
}
```

When the program first encounters the `while` loop, it checks the condition. If the condition evaluates to `true`, the conditional code will be executed. When the program reaches the end of the conditional code, it goes back up to the condition, checks it, and if it evaluates to `true`, the conditional code will be executed … and so on, as Figure 2.10 shows.

A `while` loop only finishes when its condition evaluates to `false`. This means it's important to have something inside the conditional code that will affect the condition, eventually making it evaluate to `false`. Otherwise, your program will never escape the `while` loop, and will repeat the conditional code forever, causing the browser to become unresponsive.[4]

Loops are extremely handy when they're used in conjunction with arrays, because they allow you to step sequentially through the array and perform the same operation on each element.

To step through an array with a `while` loop, you need an incrementing counter that starts at `0` and increases by one each time the loop executes. This incrementer will keep track of the index of the element that we're currently working with. When we reach the end of the array, we need to make it stop—that's where we use the array's `length` property.

In this example, we'll multiply each element of the `numbers` array by two:

```
var numbers = [1, 2, 3, 4, 5];
var incrementer = 0;
while (incrementer < numbers.length)
{
  numbers[incrementer] *= 2;
  incrementer++;
}
```

[4] In Firefox, the browser will eventually display a message to the user complaining that your script is taking a long time to execute. Oh, the shame!

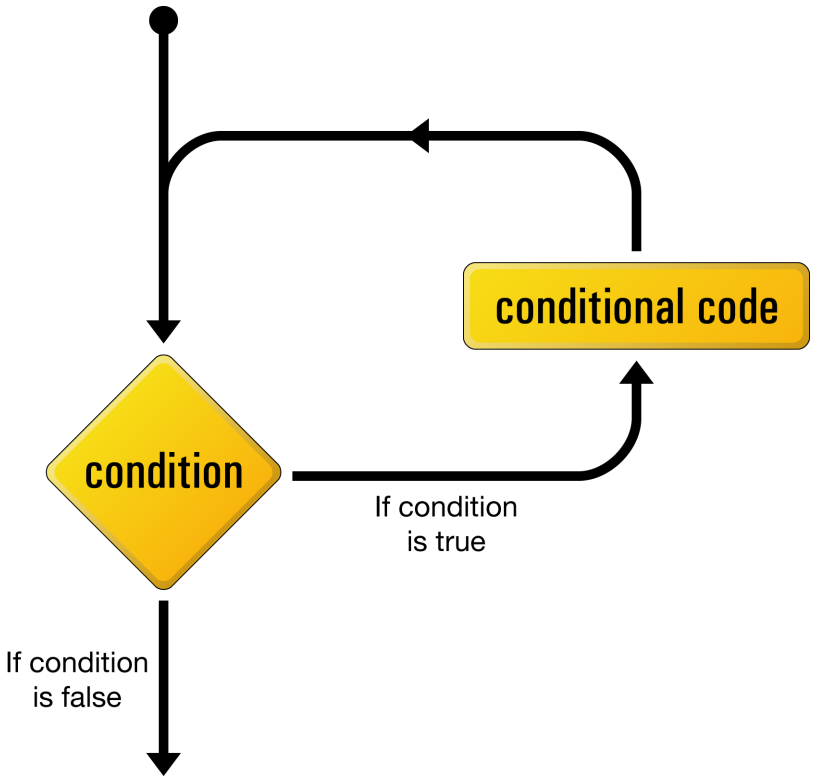


Figure 2.10. The logical flow of a `while` loop

The conditional code inside that `while` loop uses `incrementer` as the index for the array. Starting at `0`, this variable will reference the first element, but because we increase it by one for each execution of the loop, it will step through all of the elements in turn. Once `incrementer` has the same value as `numbers.length`, the condition will fail and the program will exit the `while` loop, having doubled all the elements in the array.

`i` is for `incrementer`

The variable name `incrementer` is frequently shortened to `i`, which is a commonly used name for a variable that increments inside a loop. This variable is often called a **counter variable**, because it counts how many times the loop has been executed.

do-while Loops

A do-while loop behaves almost identically to a while loop, with one key difference: the conditional code is placed before the condition, so the conditional code is always executed at least once, even if the condition is immediately false.

The conditional code is placed inside the curly braces of the do; the while statement contains the condition right after that:

```
do
{
  conditional code;
}
while (condition);
```

The flow of the program can be described as in Figure 2.11.

do-while loops aren't used very much. In fact, I don't think I've used one in ten years of programming.[5] Your friends and family will be impressed if you know about them, though.

for Loops

for loops are my favorite kind of loops—they're so succinct!

They're a lot like while loops, but they offer a couple of handy shortcuts for statements that we commonly use with loops. Consider this while loop:

```
var numbers = [1, 2, 3, 4, 5];
var i = 0;
while (i < numbers.length)
{
  numbers[i] *= 2;
  i++;
}
```

With a for loop, you can reduce the code above to:

[5] The co-author wishes it noted that he uses them all the time … possibly just because he likes to show off.

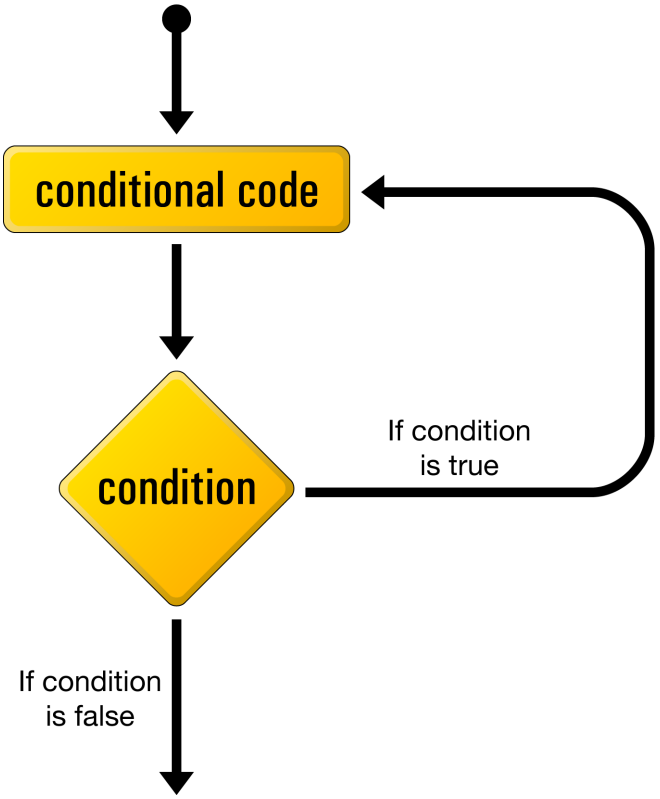


Figure 2.11. The logical flow of a `do-while` loop

```
var numbers = [1, 2, 3, 4, 5];

for (var i = 0; i < numbers.length; i++)
{
  numbers[i] *= 2;
}
```

A `for` loop shortens two aspects of the `while` loop: the declaration of a counter variable, and the incrementing of that variable.

If you look inside the round brackets immediately after the `for` keyword, you'll see three different statements separated by semicolons. The first statement is the declaration. It allows us to declare a counter variable—in this case `i`—and set its initial value.

The second statement is the condition that controls the loop. Just like the condition in a `while` loop, this condition must evaluate to `true` in order for the conditional code to be executed. It's evaluated as soon as the program reaches the `for` loop (but after the counter has been declared), so if it evaluates to `false` immediately, the conditional code will never be executed.

The third statement is an action that will be executed every time the program reaches the end of the conditional code. It is normally used to increment (or decrement) the counter, but you could theoretically put anything in there.

A `for` loop can be thought to exhibit a flow similar to that shown in Figure 2.12.

Functions: Writing Code for Later

So far, all the JavaScript code we've seen (and you've perhaps tried out) executes as soon as the page loads in your browser. It runs from top to bottom and then stops, never to run again (at least, until the page is reloaded).

Quite often, we'll want to execute different parts of our program at different times, or re-run the same code quite a few times. In order to do this, you have to put your code into **functions**.

Functions are like little packages of JavaScript code waiting to be called into action. You've seen one function already in this chapter—the `alert` function we used to pop up an alert box in the browser. `alert` is a function that's native to all browsers—that means it comes built-in with the browser's JavaScript interpreter—but it's possible to create your own functions, which you can call whenever you want.

A function can essentially be seen as a wrapper for a block of code. All you need to do is name that block, and you'll be able to call it from other areas of your program, whenever you like.

You can define your own functions using the `function` keyword. This tells the program that you're defining a new function, and that the code contained between the curly braces that follow should be executed whenever that function is called:

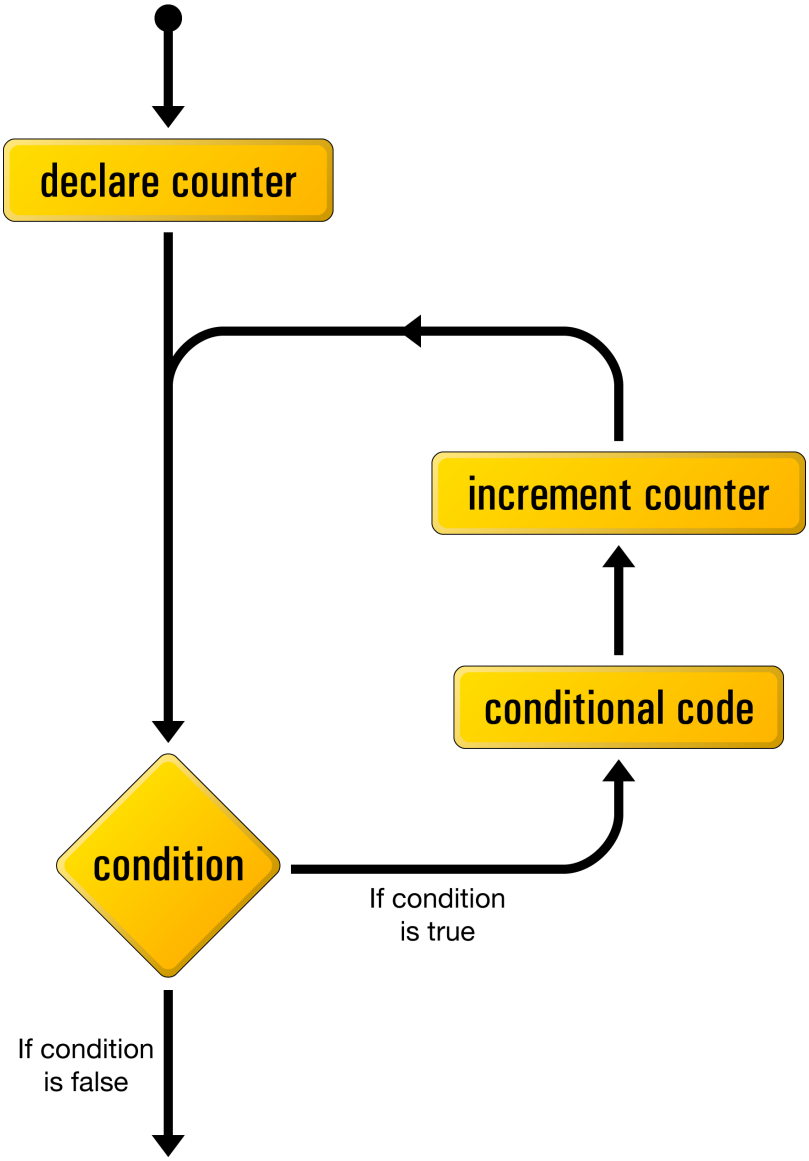


Figure 2.12. The logical flow of a `for` loop

```
function warning()
{
  alert("This is your final warning");
}
```

The name that follows the `function` keyword is the name that you want to give your function (function names have the same restrictions as variable names). This is the name you'll call whenever you want your program to run the code inside the function. The name must be followed by round brackets—they're empty in this instance, but as you'll see in the next section, this will not always be the case.

In the example above, we created a new function called `warning`, so whenever we make a call to this function, the statements inside the function will be executed, causing an alert box to appear, displaying the text, "This is your final warning."

As in the function declaration above, round brackets must appear immediately after the function name in a function call:

```
warning();
```

These brackets serve two purposes: they tell the program that you want to execute the function, and they contain the data—also known as **arguments**—that you want to pass to the function.[6] Not every function has to have arguments passed to it, but you always have to use the brackets in a function call.

Arguments: Passing Data to a Function

If you look at the ways we used the `alert` function on previous pages, you'll notice that we always inserted a string between the brackets of the function call:

```
alert("Insert and play");
```

The string `"Insert and play"` is actually an argument that we're passing to the `alert` function; the `alert` is designed to take that argument and display it in the browser's alert box.

Functions can be designed to take as many arguments as you want, and those arguments don't have to be strings—they can be any sort of data that you can create in JavaScript.

[6] Some people like to call these "parameters." Some people also like to eat sheep's brains.

When you define your function, you can provide names for the arguments that are to be passed to it. These are included in the round brackets immediately after the function name, with a comma separating arguments in cases where there's more than one:

```
function sandwich(bread, meat)
{
  alert(bread + meat + bread);
}
```

Once an argument name has been defined in the function declaration, that argument becomes a variable that's available every time the function is run, allowing you to use the data passed to the function *inside* the function itself.

As you can see in the `sandwich` function above, two arguments are defined: `bread` and `meat`. These two arguments are used in a call to `alert` and produce a little nonsensical message to the user.

Let's call the function `sandwich` with the arguments `"Rye"` and `"Pastrami"`:

```
sandwich("Rye", "Pastrami");
```

When the code for `sandwich` is executed, those arguments become available as the variables `bread` and `meat`, respectively. So, as Figure 2.13 indicates, the user would end up with a pastrami on rye.

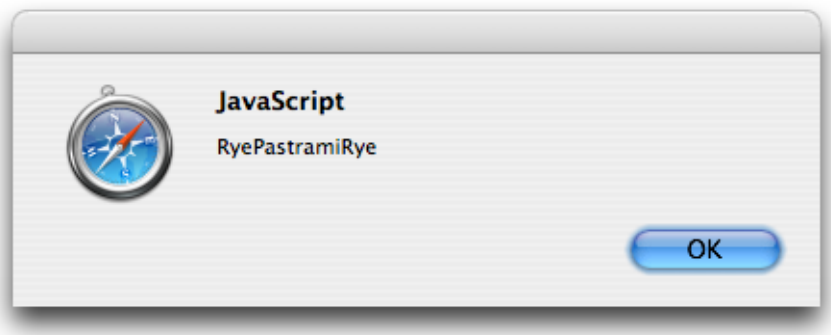


Figure 2.13. Using a function argument as a variable

The `arguments` Array

In addition to being available in their assigned argument names, the values that are passed to a function are also made available inside an automatically generated array variable named `arguments`.

Even if you don't declare any argument names in your function declaration, you can actually pass one or more arguments when you call the function. These arguments will still be available in the `arguments` array. This can be useful for writing functions that will accept any number of arguments.

Imagine we called a function with these arguments:

```
debate("affirmative", "negative");
```

We could access those arguments via the `arguments` array inside the function, like this:

```
function debate()
{
  var affirmative = arguments[0];
  var negative = arguments[1];
}
```

Return Statements: Outputting Data from a Function

Thus far, the outcome of most of our functions has been to display an alert box to the user with a message in it. But most of the time, you'll want your functions to be silent, simply passing data to other parts of your program.

A function may **return** data to the statement that called it. The neat thing about that is that you can assign a function call as the value of a variable, and that variable's value will become whatever was returned by the function.

To get a function to return a value, we use the `return` keyword, followed by the value we want it to return:

```
function sandwich(bread, meat)
{
  var assembled = bread + meat + bread;

  return assembled;
}
```

Then, the function's all ready to be used in an expression:

```
var lunch = sandwich("Rye", "Pastrami");
```

The `lunch` variable now contains the string `"RyePastramiRye"`.

If you want to get *really* tricky, you'll be pleased to hear that the return value can even be an expression:

```
function sandwich(bread, meat)
{
  return bread + meat + bread;
}
```

The expression will be evaluated and the result will be returned, producing the same effect as the previous version of the code.

A `return` statement is always the final act of a function; nothing else is processed after a function has returned. Consider this code:

```
function prematureReturner()
{
  return "Too quick";

  alert("Was it good for you?");
}
```

The `alert` function wouldn't be called, because the `return` statement would always "cut off" execution of the function. This ability to "cut off" execution of a function with a `return` statement can be handy when used in conjunction with a conditional statement, where you only want the rest of the function to be executed if a certain condition is met.

Scope: Keeping your Variables Separate

Right back at the start of this chapter I mentioned that you should avoid using variables without first declaring them using the `var` keyword. This will help you prevent variable clashes in your functions.

Most of the variables we saw in this chapter weren't declared inside a function, and therefore reside in what's known as global scope. Variables declared in **global scope** may be accessed from any other JavaScript code running in the current web page. This mightn't sound too bad, and it often won't be a problem … until you start using common variable names inside your functions.

Take a look at this program:

```
function countWiis()
{
  stock = 5;
  sales = 3;

  return stock - sales;
}

stock = 0;
wiis = countWiis();
```

What will be the value of `stock` after this code has run?

You'd probably expect it still to be `0`, which is what we set it to be before calling `countWiis`. However, `countWiis` *also* uses a variable called `stock`. But because the function doesn't use `var` to declare this variable, JavaScript will go looking *outside* the function—in the global scope—to see whether or not that variable already exists. Indeed it does, so JavaScript will assign the value `5` to that global variable.

What we really intended was for `countWiis` to use its own *separate* `stock` variable. To achieve this, we need to declare that variable with **local scope**. A variable with local scope exists only within the confines of the function in which it was created. It also takes precedence over variables with global scope—if a local variable and a global variable both have the same name, a function will always use the local variable, leaving the global variable untouched.

How do you declare a local variable? Put `var` in front of it.

Let's reformulate our code with all our variables correctly declared:

```
function countWiis()
{
  var stock = 5;
  var sales = 3;

  return stock - sales;
}

var stock = 0;
var wiis = countWiis();
```

The `stock` variable declared *outside* the `countWiis` function will remain untouched by the `stock` variable declared *inside* `countWiis`—our function can live in peace and harmony with the rest of the universe!

The lesson here is that unless you intend a variable to be shared throughout your program, always declare it with `var`.[7]

Objects

Now that we've looked at variables and functions, we can finally take a look at **objects**.

Objects are really just amorphous programming blobs. They're an amalgam of all the other data types, existing mainly to make life easier for programmers. Still, their vagueness of character doesn't mean they're not useful.

Objects exist as a way of organizing variables and functions into logical groups. If your program deals with bunnies and robots, it'll make sense to have all the functions and variables that relate to robots in one area, and all the functions and variables

[7] Strictly speaking, variables created outside of functions will always be in the global scope, whether they are declared with `var` or created simply by assigning a value to an undeclared variable name. Nevertheless, declaring all your variables with `var` is a good habit to get into, and is considered best practice.

that relate to bunnies in another area. Objects do this by grouping together sets of **properties** and **methods**.

Properties are *variables* that are only accessible via their object, and methods are *functions* that are only accessible via their object. By requiring all access to properties and methods to go through the objects that contain them, JavaScript objects make it much easier to manage your programs.

We've actually played with objects already—when you create a new array, you're creating a new instance of the built-in `Array` object. The `length` of an array is actually a property of that object, and arrays also have methods like `push` and `splice`, which we'll use later in this book.

An array is a native object, because it's built in to the JavaScript language, but it's easy to create your own objects using the `Object` constructor:

```
var Robot = new Object();
```

Naming Conventions

Variable names start with a lowercase letter, while object names start with an uppercase letter. That's just the way it is. After decades of finely honed programming practice, this convention helps everyone distinguish between the two.

Once you've instantiated your new object, you're then free to add properties and methods to it, to modify the values of existing properties, and to call the object's methods. The properties and methods of an object are both accessed using the dot (`.`) syntax:

```
Robot.metal = "Titanium";
Robot.killAllHumans = function()
{
  alert("Exterminate!");
};

Robot.killAllHumans();
```

The first line of this code adds to our empty `Robot` object a `metal` property, assigning it a value of `"Titanium"`. Note that we don't need to use the `var` keyword when

we're declaring properties, since properties are always in **object scope**—they must be accessed via the object that contains them.

The statement that begins on the second line adds a `killAllHumans` method to our `Robot` object. Note that this is a little different from the syntax that we used previously to declare a standalone function; here, our method declaration takes the form of an assignment statement (note the assignment operator, `=`, and the semicolon at the end of the code block).

Alternative Syntax for Standalone Functions

As it turns out, you can also use this syntax to declare standalone functions if you want to. Never let it be said that JavaScript doesn't give you options! Before, we used this function declaration:

```
function sandwich(bread, meat)
{
  alert(bread + meat + bread);
}
```

JavaScript lets you write this in the form of a variable assignment, if you prefer:

```
var sandwich = function(bread, meat)
{
  alert(bread + meat + bread);
};
```

As you might expect, there is a very subtle difference between the effects of these two code styles: a function declared with the former syntax can be used by any code in the file, even if it comes before the function declaration. A function declared with the latter syntax can only be used by code that executes after the assignment statement that declares the function. If your code is well organized, however, this difference won't matter.

Finally, the last line of our program calls the `Robot` object's `killAllHumans` method.

As with a lot of JavaScript, we can shortcut this whole sequence using the object literal syntax:

```
var Robot =
{
  metal: "Titanium",
  killAllHumans: function()
  {
    alert("Exterminate!");
  }
};
```

Rather than first creating an empty object and then populating it with properties and methods using a series of assignment statements, **object literal syntax** lets you create the object and its contents with a single statement.

In object literal syntax, we represent a new object with curly braces; inside those braces, we list the properties and methods of the object, separated by commas. For each property and method, we assign a value using a colon (`:`) instead of the assignment operator.

Object literal syntax can be a little difficult to read once you've been using the standard assignment syntax for a while, but it *is* slightly more succinct.

We're going to use this object literal syntax throughout this book to create neatly self-contained packages of functionality that you can easily transport from page to page.

Unobtrusive Scripting in the Real World

After reading Chapter 1, you no doubt have it fairly clear in your head that HTML is for content and JavaScript is for behavior, and never the twain shall meet. However, it's not quite that simple in the real world.

If you have a close look at the way JavaScript is downloaded alongside the HTML page that links to it, you should notice that sometimes—in fact *most* of the time—the JavaScript will download before all of the HTML has downloaded. This presents us with a slight problem.

Browsers execute JavaScript files as soon as the *JavaScript file* is downloaded—not the HTML file. So chances are that the JavaScript will be executed before all of the HTML has been downloaded. If your JavaScript executes and is trying to enhance

the HTML content with behavior before it's ready, you're probably going to start seeing JavaScript errors about HTML elements not being where they're supposed to be.

One way around this problem is to wait until all of the HTML is ready before you run any JavaScript that modifies or uses the HTML. Luckily, JavaScript has a way of detecting when the web page is ready to do this. Unluckily, the code involved is rather complicated.

To get you up to speed quickly, I've created a special library object, `Core`. This object includes a method called `start` that monitors the status of the page, and lets your JavaScript objects know when it's safe to start playing around with the HTML. It does this by calling your object's `init` method. All you have to do is let the function know which objects require this notification, and make sure each of those objects has an `init` method that will start working with the web page when it's called.

So, if you had a `Robot` object that wanted to find all the robots on your page, you'd write the following code:

```
var Robot =
{
  init: function()
  {
    Your HTML modifying code;
  }
};

Core.start(Robot);
```

By registering `Robot` with `Core.start` on the final line, you can rest assured that `Robot.init` will be run only when it's safe to do so.

`Core.start` uses some JavaScript voodoo that we'll learn about in later chapters, but if you want to know all the details now, flick to Appendix A.

Summary

If you've never programmed before, stepping into JavaScript can be a little daunting, so don't think you have to understand it straight away. Take the time to read through

this chapter's explanations again, and maybe try out some of the examples—I find I learn best by practical experience and experimentation.

Once you've got a firm understanding of the concepts behind programming and the basics of JavaScript, continue on to the next chapter, where we'll learn how to work with the contents of web pages and create some real-world programs.

Chapter 3

Document Access

Without a document, JavaScript would have no way to make its presence felt. It's HTML that creates the tangible interface through which JavaScript can reach its users.

This relationship makes it vital that JavaScript be able to access, create, and manipulate every part of the document. To this end, the W3C created the Document Object Model—a system through which scripts can influence the document. This system not only allows JavaScript to make changes to the structure of the document, but enables it to access a document's styles and change the way it looks.

If you want to take control of your interfaces, you'll first have to master the DOM.

The Document Object Model: Mapping your HTML

When an HTML document is downloaded to your browser, that browser has to do the job of turning what is essentially one long string of characters into a web page. To do this, the browser decides which parts are paragraphs, which parts are headings,

which parts are text, and so on. In order to save poor JavaScript programmers from having to do the exact same work, the browser stores its interpretation of the HTML code as a structure of JavaScript objects, called the **Document Object Model**, or **DOM**.

Within this model, each element in the HTML document becomes an object, as do all the attributes and text. JavaScript can access each of these objects independently, using built-in functions that make it easy to find and change what we want on the fly.

As a result of the way in which HTML is written—as a hierarchy of nested elements marked with start and end tags—the DOM creates a different object for each element, but links each element object to its enclosing (or parent) element. This creates an explicit parent-child relationship between elements, and lends the visualization of the DOM most readily to a tree structure.

Take, for example, this HTML:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en-US">
  <head>
    <title>DOMinating JavaScript</title>
    <meta http-equiv="Content-Type"
        content="text/html; charset=utf-8" />
  </head>
  <body>
    <h1>
      DOMinating JavaScript
    </h1>
    <p>
      If you need some help with your JavaScript, you might like
      to read articles from <a href="http://www.danwebb.net/"
          rel="external">Dan Webb</a>,
      <a href="http://www.quirksmode.org/" rel="external">PPK</a>
      and <a href="http://adactio.com/" rel="external">Jeremy
      Keith</a>.
    </p>
  </body>
</html>
```

These elements, as mapped out in the DOM, can most easily be thought of as shown in Figure 3.1.

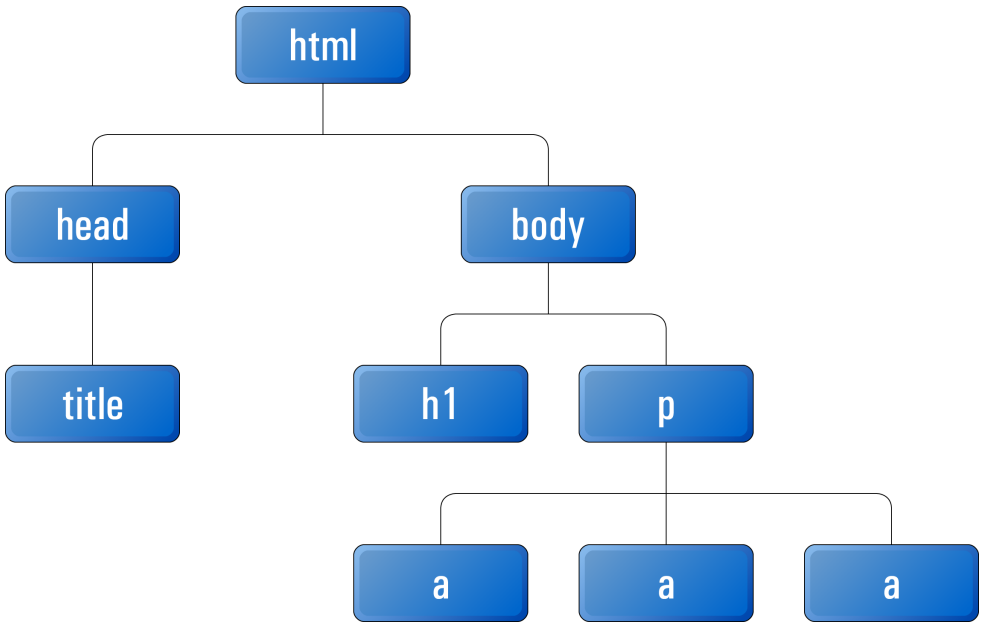


Figure 3.1. Each element on an HTML page linking to its parent in the DOM

To create the DOM for a document, each element in the HTML is represented by what's known as a **node**. A node's position in the DOM tree is determined by its parent and child nodes.

An element node is distinguished by its element name (`head`, `body`, `h1`, etc.), but this doesn't have to be unique. Unless you supply some identifying characteristic—like an `id` attribute—one paragraph node will appear much the same as another.

Technically, there's a special node that's always contained in a document, no matter what that document's content is. It always sits right at the top of the tree and it's called the **document node**. With that in mind, Figure 3.2 would be a more accurate representation of the DOM.

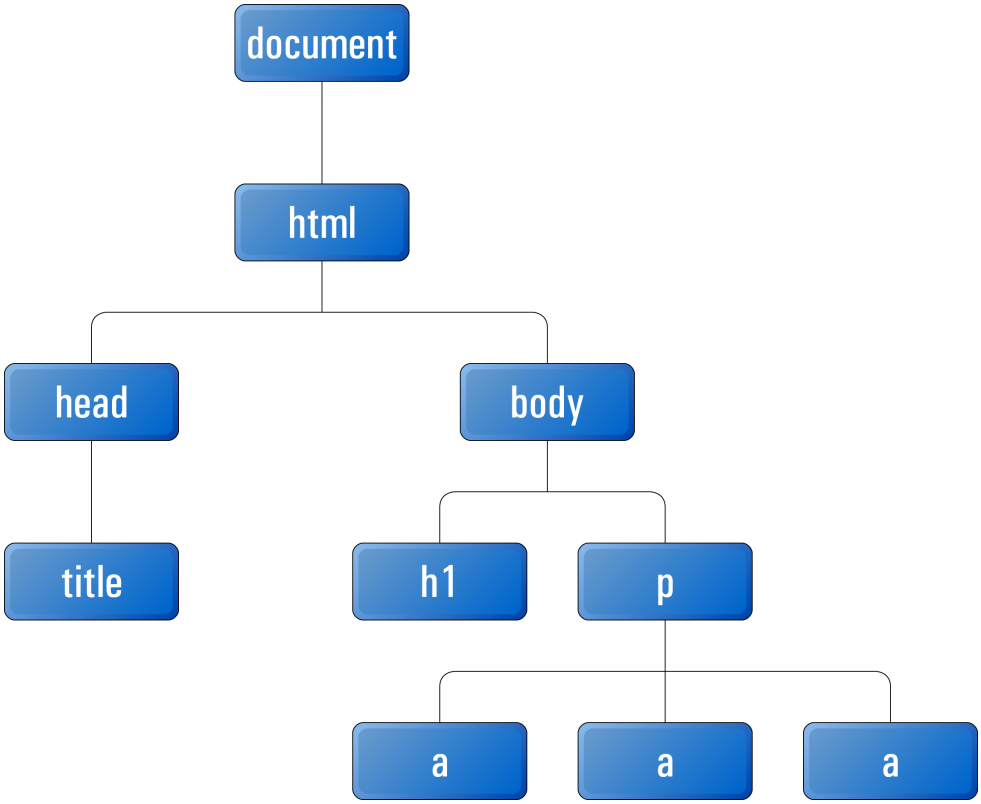


Figure 3.2. The DOM tree, including the document node

Element nodes (that is, nodes that represent HTML elements) are one type of node, and they define most of the structure of the DOM, but the actual *content* of a document is contained in two other types of nodes: text nodes and attribute nodes.

Text Nodes

In HTML code, anything that's not contained between angled brackets will be interpreted as a **text node** in the DOM. Structurally, text nodes are treated almost exactly like element nodes: they sit in the same tree structure and can be reached just like element nodes; however, they cannot have children.

If we reconsider the HTML example we saw earlier, and include the text nodes in our visualization of the DOM, it becomes a lot bigger, as Figure 3.3 illustrates.

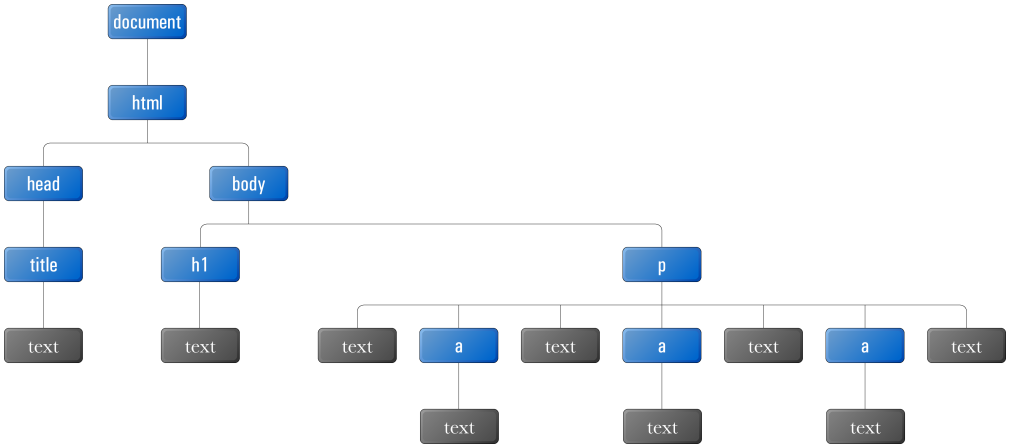


Figure 3.3. The complete DOM tree, including text nodes

Although those text nodes all look fairly similar, each node has its own value, which stores the actual text that the node represents. So the value of the text node inside the `title` element in this example would be "DOMinating JavaScript."

Whitespace May Produce Text Nodes

As well as visible characters, text nodes contain invisible characters such as new lines and tabs. If you indent your code to make it more readable (as we do in this book), each of the lines and tabs that you use to separate any tags or text will be included in a text node.

This means you may end up with text nodes in between adjacent elements, or with extra white space at the beginning or end of a text node. Browsers handle these **whitespace nodes** differently, and this variability in DOM parsing is the reason why you have to be very careful when relying upon the number or order of nodes in the DOM.

Attribute Nodes

With tags and text covered by element and text nodes, the only pieces of information that remain to be accounted for in the DOM are attributes. At first glance, attributes would appear to be part of an element—and they are, in a way—but they still occupy their own type of nodes, handily called **attribute nodes**.

Either of the two anchor elements in the example DOM we saw earlier could be visualized as shown in Figure 3.4 with the element's attribute nodes.

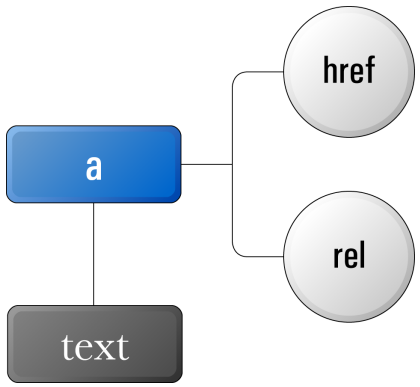


Figure 3.4. The `href` and `rel` attributes represented as attribute nodes in the DOM

Attribute nodes are always attached to an element node, but they don't fit into the structure of the DOM like element and text nodes do—they're not counted as children of the element they're attached to. Because of this, we use different functions to work with attribute nodes—we'll discuss those functions later in the chapter.

As you can see from the diagrams presented here, the DOM quickly becomes complex—even with a simple document—so you'll need some powerful ways to identify and manipulate the parts you want. That's what we'll be looking at next.

Accessing the Nodes you Want

Now that we know how the DOM is structured, we've got a good idea of the sorts of things we'll want to access. Each node—be it an element, text, or attribute node—contains information that we can use to identify it, but it's a delicate matter to sort through all of the nodes in a document to find those we want.

In many ways, manipulating an element via the DOM is a lot like applying element styles via CSS. Both tasks take this general pattern:

1. Specify the element or group of elements that you want to affect.
2. Specify the effect you want to have on them.

Although the ways in which we manipulate elements vary greatly between the two technologies, the processes we use to find the elements we want to work on are strikingly similar.

Finding an Element by ID

The most direct path to an element is via its `id` attribute. `id` is an optional HTML attribute that can be added to any element on the page, but each ID you use has to be unique within that document:

```
<p id="uniqueElement">
  ⋮
</p>
```

If you set out to find an element by ID, you'll need to make one big assumption: that the element you want has an ID. Sometimes, this assumption will mean that you need to massage your HTML code ahead of time, to make sure that the required element has an ID; at other times, that ID will naturally appear in the HTML (as part of the document's semantic structure). But once an element does have an ID, it becomes particularly easy for JavaScript to find.

If you wanted to reference a particular element by ID in CSS, you'd use an ID selector beginning with `#`:

```
#uniqueElement (1)
{
  color: blue; (2)
}
```

Roughly translated, that CSS says:

(1) Find the element with the ID `uniqueElement`.

(2) Make its color blue.

CSS is quite a succinct language. JavaScript is not. So, to reference an element by ID in JavaScript, we use the `getElementById` method, which is available only from the `document` node. It takes a string as an argument, then finds the element that has that string as its ID. I like to think of `getElementById` as a sniper that can pick out

one element at a time—highly targeted. For instance, imagine that our document included this HTML:

```
<h1>
  Sniper (1993)
</h1>
<p>
  In this cinema masterpiece,
  <a id="berenger" href="/name/nm0000297/">Tom Berenger</a> plays
  a US soldier working in the Panamanian jungle.
</p>
```

We can obtain a reference to the HTML element with the ID `berenger`, irrespective of what type of element it is:

```
var target = document.getElementById("berenger");
```

The variable `target` will now reference the DOM node for the anchor element around Tom Berenger's name. But let's suppose that the ID was moved onto another element:

```
<h1 id="berenger">
  Sniper (1993)
</h1>
<p>
  In this cinema masterpiece,
  <a href="/name/nm0000297/">Tom Berenger</a> plays a US soldier
  working in the Panamanian jungle.
</p>
```

Now, if we execute the same JavaScript code, our `target` would reference the `h1` element.

Once you have a reference to an element node, you can use lots of native methods and properties on it to gain information about the element, or modify its contents. You'll explore a lot of these methods and properties as you progress through this book.

If you'd like to try to get some information about the element we just found, you can access one or more of the element node's native properties. One such property

is `nodeName`, which tells you the exact tag name of the node you're referencing. To display the tag name of the element captured by `getElementById`, you could run this code:

```
var target = document.getElementById("berenger");
alert(target.nodeName);
```

An alert dialog will pop up displaying the tag name, as shown in Figure 3.5.

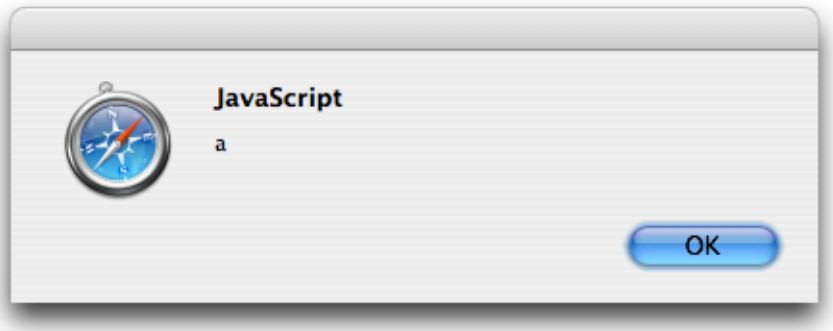


Figure 3.5. Displaying an element's tag name using the `nodeName` property

If an element with the particular ID you're looking for doesn't exist, `getElementById` won't return a reference to a node—instead, it will return the value `null`. `null` is a special value that usually indicates some type of error. Essentially, it indicates the absence of an object when one might normally be expected.

If you're not sure that your document will contain an element with the particular ID you're looking for, it's safest to check that `getElementById` actually returns a node object, because performing most operations on a `null` value will cause your program to report an error and stop running. You can perform this check easily using a conditional statement that verifies that the reference returned from `getElementById` isn't `null`:

```
var target = document.getElementById("berenger");

if (target != null)
{
  alert(target.nodeName);
}
```

Finding Elements by Tag Name

Using IDs to locate elements is excellent if you want to modify one element at a time, but if you want to find a group of elements, `getElementsByTagName` is the method for you.

Its equivalent in CSS would be the element type selector:

```
li
{
  color: blue;
}
```

Unlike `getElementById`, `getElementsByTagName` can be executed as a method of any element node, but it's most commonly called on the `document` node.

Take a look at this document:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en-US">
  <head>
    <title>Tag Name Locator</title>
    <meta http-equiv="Content-Type"
        content="text/html; charset=utf-8" />
  </head>
  <body>
    <p>
      There are 3 different types of element in this body:
    </p>
    <ul>
      <li>
        paragraph
      </li>
      <li>
        unordered list
      </li>
      <li>
        list item
      </li>
```

```
    </ul>
  </body>
</html>
```

We can retrieve all these list item elements using one line of JavaScript:

```
var listItems = document.getElementsByTagName("li");
```

By executing that code, you're telling your program to search through all of the descendants of the `document` node, get all the nodes with a tag name of `"li"`, and assign that group to the `listItems` variable.

`listItems` ends up containing a collection of nodes called a **node list**. A node list is a JavaScript object that contains a list of node objects in source order. In the example we just saw, all the nodes in the node list have a tag name of `"li"`.

Node lists behave a lot like arrays, which we saw in Chapter 2, although they lack some of the useful methods that arrays provide. In general, however, you can treat them the same way. Since `getElementsByTagName` always returns a node list in source order, we know that the second node in the list will actually be the second node in the HTML source, so to reference it you would use the index `1` (remember, the first index in an array is `0`):

```
var listItems = document.getElementsByTagName("li");
var secondItem = listItems[1];
```

`secondItem` would now be a reference to the list item containing the text "unordered list."

Node lists also have a `length` property, so you can retrieve the number of nodes in a collection by referencing its `length`:

```
var listItems = document.getElementsByTagName("li");
var numItems = listItems.length;
```

Given that the document contained three list items, `numItems` will be `3`.

The fact that a node list is referenced similarly to an array means that it's easy to use a loop to perform the same task on each of the nodes in the list. If we wanted to check that `getElementsByTagName` only returned elements with the same tag name, we could output the tag name of each of the nodes using a `for` loop:

```
var listItems = document.getElementsByTagName("li");

for (var i = 0; i < listItems.length; i++)
{
  alert(listItems[i].nodeName);
}
```

Unlike `getElementById`, `getElementsByTagName` will return a node list even if no elements matched the supplied tag name. The `length` of this node list will be `0`. This means it's safe to use statements that check the `length` of the node list, as in the loop above, but it's *not* safe to directly reference an index in the list without first checking the `length` to make sure that the index will be valid. Looping through the node list using its `length` property as part of the loop condition is usually the best way to do this.

Restricting Tag Name Selection

At the start of this section, I mentioned that `getElementsByTagName` can be executed from any element node, not just the `document` node. Calling this method from an element node allows you to restrict the area of the DOM from which you want to select nodes.

For instance, imagine that your document included multiple unordered lists, like this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en-US">
  <head>
    <title>Tag Name Locator</title>
    <meta http-equiv="Content-Type"
        content="text/html; charset=utf-8" />
  </head>
  <body>
    <p>
```

```
      There are 3 different types of element in this body:
    </p>
    <ul>
      <li>
        paragraph
      </li>
      <li>
        unordered list
      </li>
      <li>
        list item
      </li>
    </ul>
    <p>
      There are 2 children of html:
    </p>
    <ul>
      <li>
        head
      </li>
      <li>
        body
      </li>
    </ul>
  </body>
</html>
```

Now, you might want to get the list items from the second list only—not the first. If you were to call `document.getElementsByTagName("li")`, you'd end up with a collection that contained all five list items in the document, which, obviously, is not what you want. But if you get a reference to the second list and use *that* reference to call the method, it's possible to get the list items from that list alone:

```
var lists = document.getElementsByTagName("ul");
var secondList = lists[1];
var secondListItems = secondList.getElementsByTagName("li");
```

`secondListItems` now contains just the two list items from the second list.

Here, we've used two `getElementsByTagName` calls to get the elements we wanted, but there is an alternative. We could use a `getElementById` call to get the required reference to the second list (if the second list had an ID) before we called

`getElementsByTagName`, to get the list items it contains. Combining multiple DOM method calls is something you should get a feel for fairly quickly. The best approach will often depend upon the structure of the HTML you're dealing with.

Finding Elements by Class Name

It's quite often very handy to find elements based on a class rather than a tag name. Although we're stuck with the same 91 HTML elements wherever we go, we can readily customize our classes to create easily referenced groups of elements that suit our purposes.

Compared to searching by tag name, using a class as a selector can be a more granular way to find elements (as it lets you get a subset of a particular tag name group) as well as a broader way to find elements (as it lets you select a group of elements that have a range of tag names).

Unfortunately, no built-in DOM function lets you get elements by class, so I think it's time we created our first real function! Once that's done, we can add the function to our custom JavaScript library and call it whenever we want to get all elements with a particular class.

Starting your First Function

When you're writing a function or a program, your first step should be to define clearly in plain English what you want it to do. If you're tackling a relatively simple problem, you might be able to translate that description straight into JavaScript, but usually you'll need to break the task down into simple steps.

The full description of what we want to do here could be something like, "find all elements with a particular class in the document."

That sounds deceptively simple; let's break it down into more logical steps:

1. Look at each element in the document.
2. For each element, perform a check that compares its class against the one we're looking for.
3. If the classes match, add the element to our group of elements.

A couple of things should jump out at you immediately from those steps. Firstly, whenever you see the phrase "for each," chances are that you're going to need a loop. Secondly, whenever there's a condition such as "if it matches," you're going to need a conditional statement. Lastly, when we talk about a "group," that usually means an array or node list.

With those predictions in mind, let's turn these three steps into code.

Looking at All the Elements

First of all, we'll need to get all the elements in the document. We do this using `getElementsByTagName`, but we're not going to look for a particular tag; instead, we're going to pass this method the special value `"*"`, which tells it to return all elements.

Unfortunately, Internet Explorer 5.x doesn't understand that special value, so we have to write some additional code in order to support that browser. In Internet Explorer 5.x, Microsoft created a special object that contains all the elements in the document, and called it `document.all`. `document.all` is basically a node list containing all the elements, so it's synonymous with calling `document.getElementsByTagName("*")`.

Most other browsers don't have the `document.all` object, but those that do implement it just like Internet Explorer, so our code can simply test to see whether `document.all` exists. If it does, we use the Internet Explorer 5.x way of getting all the elements. If it doesn't, we use the normal approach:

```
var elementArray = [];

if (typeof document.all != "undefined")
{
  elementArray = document.all;
}
else
{
  elementArray = document.getElementsByTagName("*");
}
```

The conditional statement above uses the `typeof` operator to check for the existence of `document.all`. `typeof` checks the data type of the value that follows it, and pro-

duces a string that describes the value's type (for instance, `"number"`, `"string"`, `"object"`, etc.). Even if the value is `null`, it will still return a type (`"object"`), but if you supply `typeof` with a variable or property name that hasn't been assigned any value whatsoever, it will return the string `"undefined"`. This technique, called **object detection**, is the safest way of testing whether an object—such as `document.all`—exists. If `typeof` returns `"undefined"`, we know that the browser doesn't implement that feature.

Whichever part of the conditional statement the browser decides to execute, we end up correctly assigning to `elementArray` a node list of every element in the document.

Checking the Class of Each Element

Now that we have a collection of elements to look at, we can check the class of each:

```
var pattern = new RegExp("(^| )" + theClass + "( |$)");

for (var i = 0; i < elementArray.length; i++)
{
  if (pattern.test(elementArray[i].className))
  {
    ⋮
  }
}
```

The value that we assign to the variable `pattern` on the first line will probably look rather alien to you. In fact, this is a **regular expression**, which we'll explore more fully in Chapter 6. For now, what you need to know is that regular expressions help us search strings for a particular pattern. In this case, our regular expression uses the variable `theClass` as the class we want to match against; `theClass` will be passed into our function as an argument.

Once we've set up our regular expression with that class name, we use a `for` loop to step through each of the elements in `elementArray`.

Every time we move through the `for` loop, we use the `pattern` regular expression, testing the current element's `class` attribute against it. We do this by passing the element's `className` property—a string value—to the regular expression's `test`

method. Every element node has a `className` property, which corresponds directly to that element's `class` attribute in the HTML.

When `pattern.test` is run, it checks the string argument that's passed to it against the regular expression. If the string matches the regular expression (that is, it contains the specified class name), it will return `true`; if the string doesn't match the regular expression, it will return `false`. In this way, we can use a regular expression test as the condition for an `if` statement. In this example, we use the conditional statement to tell us if the current element has a class that matches the one we're looking for.

But why can't we just perform a direct string comparison on the class, like this?

```
if (elementArray[i].className == theClass) // this won't work
```

The thing about dealing with an element's `className` property is that it can actually contain multiple classes, separated by spaces, like this:

```
<div class="article summary clicked">
```

For this reason, simply checking whether the `class` attribute's value equals the class that we're interested in is not always sufficient. When checking to see whether `class` contains a particular class, we need to use a more advanced method of searching within the attribute value, which is why we used a regular expression.

Adding Matching Elements to our Group of Elements

Once we've decided that an element matches the criteria we've set, we need to add it to our group of elements. But where's our group? Earlier, I said that a node list is a lot like an array. We can't actually create our own node lists—the closest thing we can create is an array.

Outside the `for` loop, we create the array that's going to hold the group of elements, then add each matched element to the array as we find it:

```
var matchedArray = [];
var pattern = new RegExp("(^| )" + theClass + "( |$)");

for (var i = 0; i < elementArray.length; i++)
```

```
{
  if (pattern.test(elementArray[i].className))
  {
    matchedArray[matchedArray.length] = elementArray[i];
  }
}
```

Within the `if` statement we wrote in the previous step, we add any newly matched elements to the end of `matchedArray`, using its current `length` as the index of the new element (remember that the `length` of an array will always be one more than the index of the last element).

Once the `for` loop has finished executing, all of the elements in the document that have the required class will be referenced inside `matchedArray`. We're almost done!

Putting it All Together

The guts of our function are now pretty much written. All we have to do it paste them together and put them inside a function:

core.js *(excerpt)*

```
Core.getElementsByClass = function(theClass)
{
  var elementArray = [];

  if (document.all)
  {
    elementArray = document.all;
  }
  else
  {
    elementArray = document.getElementsByTagName("*");
  }

  var matchedArray = [];
  var pattern = new RegExp("(^| )" + theClass + "( |$)");

  for (var i = 0; i < elementArray.length; i++)
  {
    if (pattern.test(elementArray[i].className))
    {
```

```
      matchedArray[matchedArray.length] = elementArray[i];
    }
  }

  return matchedArray;
};
```

We've called our new function `Core.getElementsByClass`, and our function definition contains one argument—`theClass`—which is the class we use to construct our regular expression. As well as placing the code inside a function block, we include a `return` statement that passes `matchedArray` back to the statement that called `Core.getElementsByClass`.

Now that it's part of our `Core` library, we can use this function to find a group of elements by class from anywhere in our JavaScript code:

```
var elementArray = Core.getElementsByClass("dataTable");
```

Navigating the DOM Tree

The methods for finding DOM elements that I've described so far have been fairly targeted—we're jumping straight to a particular node in the tree without worrying about the connections in between.

This works fine when there's some distinguishing feature about the element in question that allows us to identify it: an ID, a tag name, or a class. But what if you want to get an element on the basis of its relationship with the nodes that surround it? For instance, if we have a list item node and want to retrieve its parent `ul`, how do we do that? For that matter, how do we get the next item in the list?

For each node in the tree, the DOM specifies a number of properties, and it's these properties that allow us to move around the tree one step at a time. Where `document.getElementById` and its ilk are like direct map references ("go to S37° 47.75', E144° 59.01'"), these DOM properties are like giving directions: "turn left onto the Bayshore Freeway and a right onto Amphitheater Parkway." Some people call this process **walking the DOM**.

Finding a Parent

Every element node—except for the document node—has a parent. Consequently, each element node has a property called `parentNode`. When we use this property, we receive a reference to the target element's parent.

Consider this HTML:

```
<p>
  <a id="oliver" href="/oliver/">Oliver Twist</a>
</p>
```

Once we have a reference to the anchor element, we can get a reference to its parent paragraph using `parentNode` like so:

```
var oliver = document.getElementById("oliver");
var paragraph = oliver.parentNode;
```

Finding Children

The parent-child relationship isn't just one way. You can find all of the children of an element using the `childNodes` property.

An element can only have one parent, but it can have many children, so `childNodes` is actually a node list that contains all of the element's children, in source order.

Take, for instance, a list like this:

```
<ul id="baldwins">
  <li>
    Alec
  </li>
  <li>
    Daniel
  </li>
  <li>
    William
  </li>
  <li>
```

```
    Stephen
  </li>
</ul>
```

The unordered list node will have four child nodes,[1] each of which matches a list item. To get the third list item (the one containing "William") in the list above, we'd get the third element in the `childNodes` list:

```
var baldwins = document.getElementById("baldwins");
var william = baldwins.childNodes[2];
```

Two shortcut properties are available to help us get the first child or last child of an element: the `firstChild` and `lastChild` properties, respectively.

To get the "Alec" list item, we could just use:

```
var alec = baldwins.firstChild;
```

And to get the "Stephen" list item, we can use:

```
var stephen = baldwins.lastChild;
```

I don't think `firstChild` is all that much easier than typing `childNodes[0]`, but `lastChild` is definitely shorter than `childNodes[childNodes.length – 1]`, so it's a shortcut that I use regularly.

Finding Siblings

As well as moving up and down the DOM tree, we can move from side to side by getting the next or previous node on the same level. The properties we use to do so are `nextSibling` and `previousSibling`.

If we continued on from the example we saw a moment ago, we could get to the "Stephen" list item from "William" using `nextSibling`:

```
var stephen = william.nextSibling;
```

[1] As noted at the start of this chapter, the number of nodes may vary depending on whether the browser in question counts the whitespace between each of the list items.

We could get to the "Daniel" list item using `previousSibling`:

```
var daniel = william.previousSibling;
```

If we're at the last node on a level, and try to get the `nextSibling`, the property will be `null`. Similarly, if we're at the first node on a level and try to get `previousSibling`, that property will also be `null`. You should check to make sure you have a valid node reference whenever you use either of these properties.

Figure 3.6 provides a clear visualization of where each of these DOM-walking properties will get you to from a given node in the DOM tree.

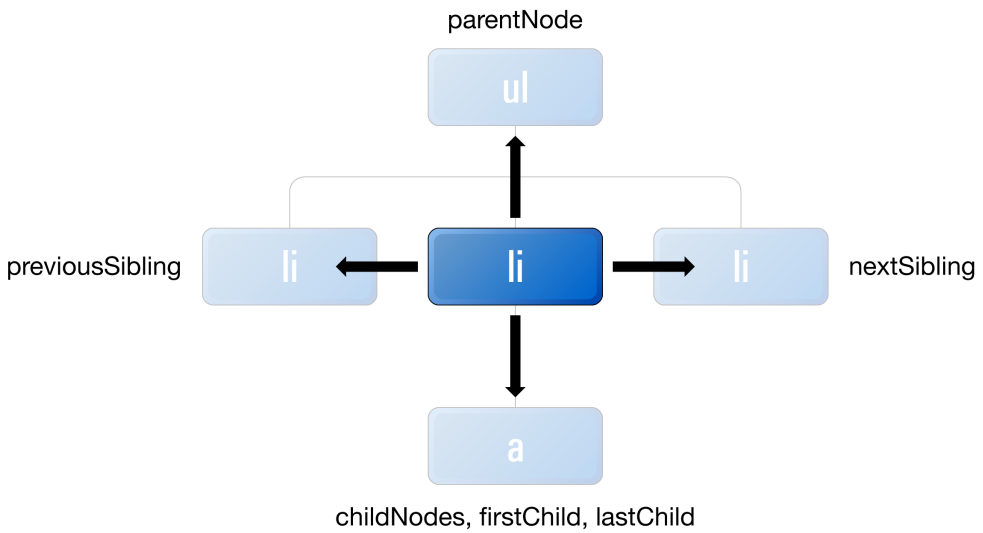


Figure 3.6. Moving around the DOM tree using the element node's DOM properties

Interacting with Attributes

As I mentioned when we discussed the structure of the DOM, attributes are localized to the elements they're associated with—they don't have much relevance in the larger scheme of things. Therefore, we don't have DOM functions that will let you find a particular attribute node, or all attributes with a certain value.

Attributes are more focused on reading and modifying the data related to an element. As such, the DOM only offers two methods related to attributes, and both of them can only be used once you have an element reference.

Getting an Attribute

With a reference to an element already in hand, you can get the value of one of its attributes by calling the method `getAttribute` with the attribute name as an argument.

Let's get the `href` attribute value for this link:

```
<a id="koko" href="http://www.koko.org/">Let's all hug Koko</a>
```

We need to create a reference to the anchor element, then use `getAttribute` to retrieve the value:

```
var koko = document.getElementById("koko");
var kokoHref = koko.getAttribute("href");
```

The value of `kokoHref` will now be `"http://www.koko.org/"`.

This approach works for any of the attributes that have been set for an element:

```
var koko = document.getElementById("koko");
var kokoId = koko.getAttribute("id");
```

The value of `kokoId` will now be `"koko"`.

At least, that's how it's *supposed* to work, according to the W3C. But in reality, `getAttribute` is beset by problems in quite a few of the major browsers.[2] Firefox returns `null` for unset values when it's supposed to return a string, as does Opera 9. Internet Explorer returns a string for most unset attributes, but returns `null` for non-string attributes like `onclick`. When it does return a value, Internet Explorer subtly alters a number of the attribute values it returns, making them different from those returned by other browsers. For example, it converts `href` attribute values to absolute URLs.

[2] `getAttribute` is a bit of a mess across all browsers, but most noticeably in Internet Explorer. For a complete rundown of what's going on, visit http://tobielangel.com/2007/1/11/attribute-nightmare-in-ie.

With all of these problems currently in play, at the moment it's safer to use the old-style method of getting attributes, which we can do by accessing each attribute as a dot property of an element.

In using this approach to get the `href` on our anchor, we'd rewrite the code as follows:

```
var koko = document.getElementById("koko");
var kokoHref = koko.href;
```

In most cases, fetching an attribute value is just a matter of appending the attribute name to the end of the element, but in a couple of cases the attribute name is a reserved word in JavaScript. This is why we use *`element`*`.className` for the `class` attribute, and why, if you ever need to get the `for` attribute, you'll need to use *`element`*`.htmlFor`.

Setting an Attribute

As well as being readable, all HTML attributes are writable via the DOM.

To write an attribute value, we use the `setAttribute` method on an element, specifying both the attribute name we want to set and the value we want to set it to:

```
var koko = document.getElementById("koko");
koko.setAttribute("href", "/koko/");
```

When we run those lines of code, the `href` for Koko's link will change from `http://www.koko.org/` to `/koko/`.

Thankfully, there are no issues with `setAttribute` across browsers, so we can safely use it anywhere.

`setAttribute` can be used not only to change preexisting attributes, but also to add new attributes. So if we wanted to add a title that described the link in more detail, we could use `setAttribute` to specify the value of the new `title` attribute, which would be added to the anchor element:

```
var koko = document.getElementById("koko");
koko.setAttribute("title", "Web site of the Gorilla Foundation");
```

If you were to take the browser's internal representation of the document following this DOM change and convert it to HTML, here's what you'd get:

```
<a id="koko" href="http://www.koko.org/"
    title="Web site of the Gorilla Foundation">Let's all hug
  Koko</a>
```

Changing Styles

Almost every aspect of your web page is accessible via the DOM, including the way it looks.

Each element node has a property called `style`. `style` is a deceptively expansive object that lets you change every aspect of an element's appearance, from the color of its text, to its line height, to the type of border that's drawn around it. For every CSS property that's applicable to an element, `style` has an equivalent property that allows us to change that property's value.

To change the text color of an element, we'd use `style.color`:

```
var scarlet = document.getElementById("scarlet");
scarlet.style.color = "#FF0000";
```

To change its background color, we'd use `style.backgroundColor`:

```
var indigo = document.getElementById("indigo");
indigo.style.backgroundColor = "#000066";
```

We don't have enough space here to list every property you could change, but there's a good rule of thumb: if you wish to access a particular CSS property, simply append it as a property of the `style` object. Any properties that include hyphens (like `text-indent`) should be converted to camel case (`textIndent`). If you leave the hyphen in there, JavaScript will try to subtract one word from the other, which makes about as much sense as that sentence!

Any changes to the `style` object will take immediate effect on the display of the page. Using `style`, it's possible to change a page like Figure 3.7 into a page like Figure 3.8 using just three lines of code.

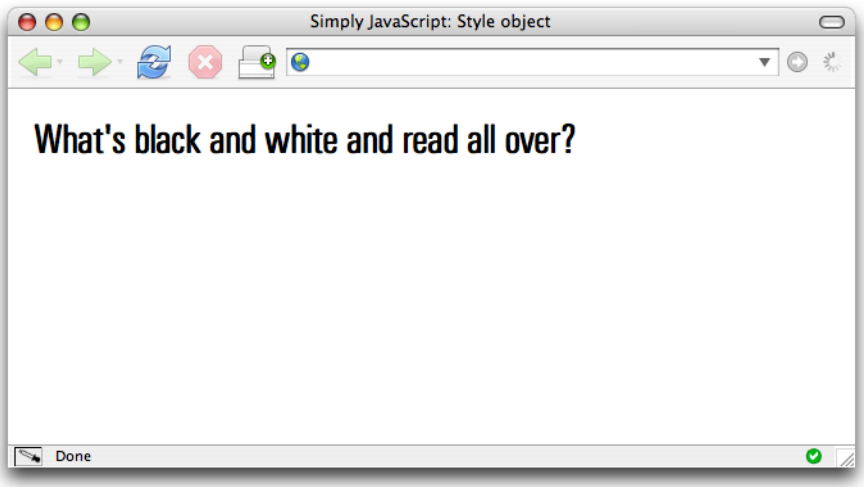


Figure 3.7. A standard page

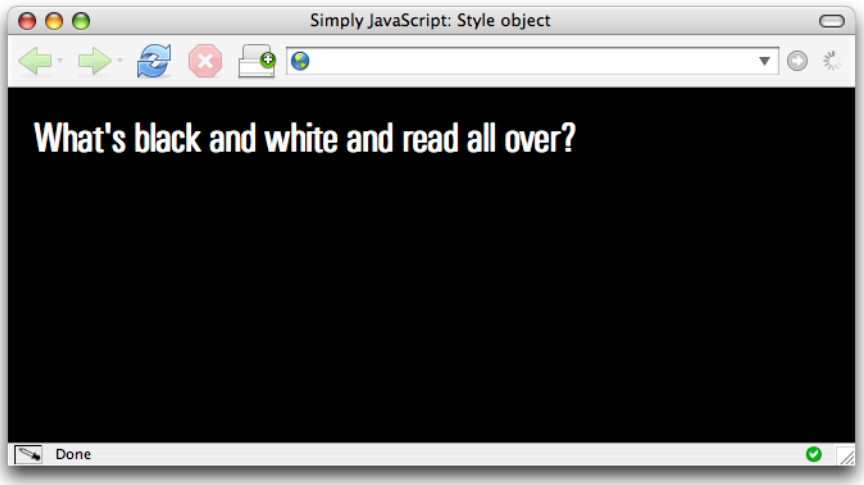


Figure 3.8. The same page, altered using `style`

Here's the code that makes all the difference:

style_object.js *(excerpt)*

```
var body = document.getElementsByTagName("body")[0];
body.style.backgroundColor = "#000000";
body.style.color = "#FFFFFF";
```

The `color` CSS property is inherited by child elements, so changing `style.color` on the `body` element will also affect every element inside the `body` to which a specific color is not assigned.

The `style` object directly accesses the HTML `style` attribute, so the JavaScript code we just saw is literally equivalent to this HTML:

```
<body style="background-color: #000000; color: #FFFFFF;">
```

As it *is* the inline style of an element, if you make a change to an element's `style` property, and that change conflicts with any of the rules in your CSS files, the `style` property will take precedence (except, of course, for properties marked `!important`).

Changing Styles with Class

In the world of CSS, it's considered bad practice to use inline styles to style an element's appearance. Equally, in JavaScript it's considered bad practice to use the `style` property as a means of styling an element's appearance.

As we discussed in Chapter 1, you want to keep the layers separated, so HTML shouldn't include style information, and JavaScript shouldn't include style information.

The best way to change an element's appearance with JavaScript is to change its class. This approach has several advantages:

- We don't mix behavior with style.
- We don't have to hunt through a JavaScript file to change styles.
- Style changes can be made by those who make the styles, not the JavaScript programmers.
- It's more succinct to write styles in CSS.

Most of the time, changes to an element's appearance can be defined as distinct changes to its state, as described in its `class`. It's these state changes that you should be controlling through JavaScript, not specific properties of its appearance.

The only situation in which it's okay to use the `style` property arises when you need to calculate a CSS value on the fly. This often occurs when you're moving objects around the screen (for instance, to follow the cursor), or when you animate

a particular property, such as in the "yellow fade" technique (which changes an element's `background-color` by increments).

Comparing Classes

When we're checking to see whether `className` contains a particular class, we need to use a special search, like the one we used to write `Core.getElementsByClass` earlier in this chapter. In fact, we can use that same regular expression to create a function that will tell us whether or not an element has a particular class attached to it:

core.js *(excerpt)*

```
Core.hasClass = function(target, theClass)
{
  var pattern = new RegExp("(^| )" + theClass + "( |$)");

  if (pattern.test(target.className))
  {
    return true;
  }

  return false;
};
```

`Core.hasClass` takes two arguments: an element and a class. The class is used inside the regular expression and compared with the `className` of the element. If the `pattern.test` method returns `true`, it means that the element *does* have the specified class, and we can return `true` from the function. If `pattern.test` returns `false`, `Core.hasClass` returns `false` by default.

Now, we can very easily use this function inside a conditional statement to execute some code when an element has (or doesn't have) a matching class:

```
var scarlet = document.getElementById("scarlet");

if (Core.hasClass(scarlet, "clicked"))
{
  ⋮
}
```

Adding a Class

When we're *adding* a class, we have to take the same amount of care as we did when comparing it. The main thing we have to be careful about here is to not overwrite an element's existing classes. Also, to make it easy to remove a class, we shouldn't add a class to an element that already has that class. To make sure we don't, we'll use `Core.hasClass` inside `Core.addClass`:

core.js *(excerpt)*

```
Core.addClass = function(target, theClass)
{
  if (!Core.hasClass(target, theClass))
  {
    if (target.className == "")
    {
      target.className = theClass;
    }
    else
    {
      target.className += " " + theClass;
    }
  }
};
```

The first conditional statement inside `Core.addClass` uses `Core.hasClass` to check whether or not the `target` element already has the class we're trying to add. If it does, there's no need to add the class again.

If the `target` *doesn't* have the class, we have to check whether that element has *any* classes at all. If it has none (that is, the `className` is an empty string), it's safe to assign `theClass` directly to `target.className`. But if the element has some preexisting classes, we have to follow the syntax for multiple classes, whereby each class is separated by a space. Thus, we add a space to the end of `className`, followed by `theClass`. Then we're done.

Now that `Core.addClass` performs all these checks for us, it's easy to use it whenever we want to add a new class to an element:

class.js *(excerpt)*

```
var body = document.getElementsByTagName("body")[0];
Core.addClass(body, "unreadable");
```

Then, we specify some CSS rules for that class in our CSS file:

class.css

```
.unreadable
{
  background-image: url(polka_dots.gif);
  background-repeat: 15px 15px;
  color: #FFFFFF;
}
```

The visuals for our page will swap from those shown in Figure 3.9 to those depicted in Figure 3.10.

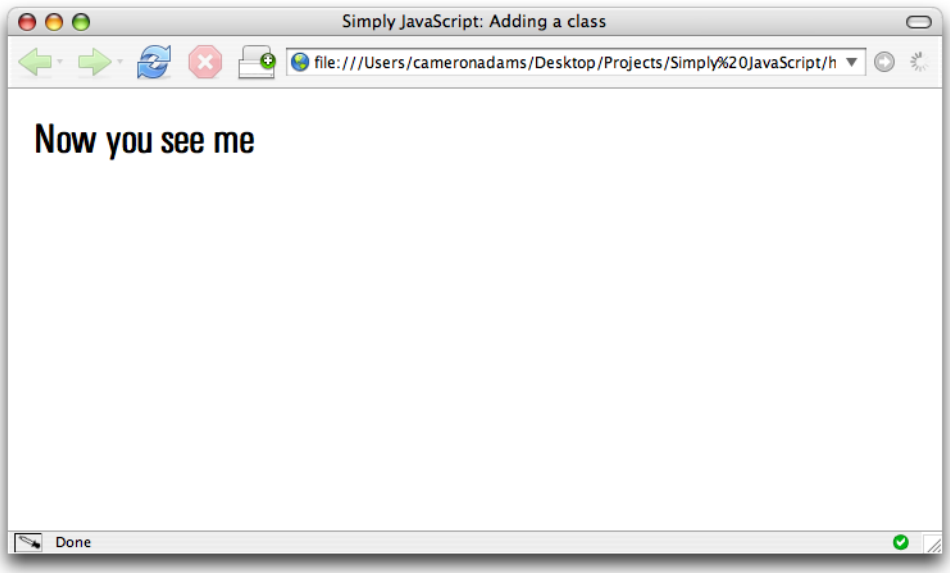


Figure 3.9. The page before we start work

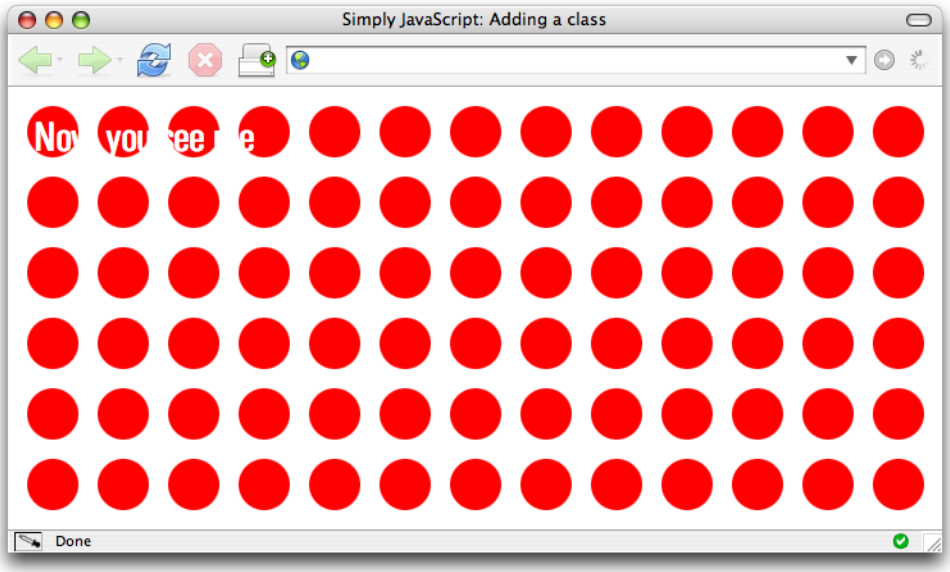


Figure 3.10. The display after we change the class of the body element

Removing a Class

When we want to remove a class from an element, we use that same regular expression (it's a pretty handy one, huh?), but with a slightly different twist:

core.js *(excerpt)*

```
Core.removeClass = function(target, theClass)
{
  var pattern = new RegExp("(^| )" + theClass + "( |$)");

  target.className = target.className.replace(pattern, "$1");
  target.className = target.className.replace(/ $/, "");
};
```

In `Core.removeClass`, instead of using the regular expression to check whether or not the target element has the class, we assume that it does have the class, and instead use the regular expression to replace the class with an empty string, effectively removing it from `className`.

To do this, we use a built-in string method called `replace`. This method takes a regular expression and a replacement string, then replaces the occurrences that match the regular expression with the replacement string. In this case, we're using an empty string as the replacement, so any matches will be erased. If the class exists inside `className`, it will disappear.

The second call to `replace` just tidies up `className`, removing any extraneous spaces that might be hanging around after the class was removed (some browsers will choke if any spaces are present at the start of `className`). Since we assign both these operations back to `className`, the `target` element's class will be updated with the changes straight away, and we can return from the function without fuss.

Example: Making Stripy Tables

Earlier in this chapter, we made our first real *function*, `Core.getElementsByClass`, but now I think you're ready to make your first real *program*, and a useful one it is too!

In my days as an HTML jockey, there was one task I dreaded more than any other, and that was making stripy tables. On static pages, you had to hand code tables so that every odd row had a special class like `alt`, but I just knew that as soon as I finished classing 45 different rows my manager was going to come along and tell me he wanted to add one more row right at the top. Every odd row would become even and every even row would become odd. Then I'd have to remove 45 classes and add them to 45 other rows. Argh!

Of course, that was before I knew about JavaScript. With JavaScript and the magic of the `for` loop, you can include one JavaScript file in your page, sit back, and change tables to your heart's delight. Obviously we're going to be using JavaScript to add a class to every second row in this example. But it might help to break down the desired outcome into a series of simple steps again.

In order to achieve stripy tables, we'll want to:

1. Find all tables with a class of `dataTable` in the document.
2. For each table, get the table rows.
3. For every second row, add the class `alt`.

By now, glancing at that list should cause a few key ideas to spring to mind. On the programming structure side of the equation, you should be thinking about loops, and plenty of them. But on the DOM side you should be thinking about `getElementsByTagName`, `className`, and maybe even our own custom function, `Core.getElementsByClass`. If you found yourself muttering any of those names under your breath while you read through the steps in that list, give yourself a pat on the back.

Finding All Tables with Class `dataTable`

This first step's pretty simple, since we did most of the related work mid-chapter. We don't want to apply striping to every table in the document (just in case someone's been naughty and used one for layout), so we'll apply it only to the tables marked with a class of `dataTable`. To do this, all we have to do is dust off `Core.getElementsByClass`—it will be able to go and find all the `dataTable` elements:

stripy_tables.js *(excerpt)*

```
var tables = Core.getElementsByClass("dataTable");
```

Done. You can't beat your own custom library!

Remember to Load your Library

Remember to add a `<script>` tag to your HTML document to load the Core library of functions (**core.js**) before the `<script>` tag that runs your program, as shown in the code below. Otherwise, your program won't be able to find `Core.getElementsByClass`, and your browser will report a JavaScript error.

stripy_tables.html *(excerpt)*

```
<script type="text/javascript" src="core.js"></script>
<script type="text/javascript" src="stripy_tables.js">
</script>
```

Getting the Table Rows for Each Table

There's that phrase "for each" again. Inside the variable `tables` we have the collection of tables waiting to be striped—we just need to iterate through each of them using a `for` loop.

Every time we move through the `for` loop, we'll want to get the rows for that particular table. This sounds okay, but it's not that simple. Let's look at the markup for a nicely semantic and accessible table:

stripy_tables.html *(excerpt)*

```
<table class="dataTable">
  <thead>
    <tr>
      <th scope="col">
        Web Luminary
      </th>
      <th scope="col">
        Height
      </th>
      <th scope="col">
        Hobbies
      </th>
      <th scope="col">
        Digs microformats?
      </th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>
        John Allsopp
      </td>
      <td class="number">
        6'1"
      </td>
      <td>
        Surf lifesaving, skateboarding, b-boying
      </td>
      <td class="yesno">
        <img src="tick.gif" alt="Yes" />
      </td>
```

```
    </tr>
    ⋮
  </tbody>
</table>
```

There's one row in there that we don't want to be susceptible to striping—the row inside the `thead`.

To avoid affecting this row through our row striping shenanigans, we need to get only the rows that are inside a `tbody`. This means we must add a step to our code—we need to get all of the `tbody` elements in the table (HTML allows more than one to exist), then get all the rows inside each `tbody`. This process will actually require *two* `for` loops—one to step through each of the `table` elements in the document, and another *inside* that to step through each of the `tbody` elements—but that's fine; it just means more work for the computer. Since the variable name `i` is used for the counter in the outer `for` loop, we'll name the counter variable in our inner `for` loop `j`:

stripy_tables.js *(excerpt)*

```
for (var i = 0; i < tables.length; i++)
{
  var tbodys = tables[i].getElementsByTagName("tbody");

  for (var j = 0; j < tbodys.length; j++)
  {
    var rows = tbodys[j].getElementsByTagName("tr");
    ⋮
  }
}
```

The results for both uses of `getElementsByTagName` in the code above will be limited to the current table, because we're using it as a method of a particular element, not the entire document. The variable `rows` now contains a collection of all the `tr` elements that exist inside a `tbody` element of the current table.

Adding the Class `alt` to Every Second Row

"For every" is equivalent to "for each" here, so we know that we're going to use yet another `for` loop. It will be a slightly different `for` loop though, because we only want to modify every second row.

To do this, we'll start the counter on the *second* index of the collection and increment it by two, not one:

stripy_tables.js *(excerpt)*

```
for (var i = 0; i < tables.length; i++)
{
  var tbodys = tables[i].getElementsByTagName("tbody");

  for (var j = 0; j < tbodys.length; j++)
  {
    var rows = tbodys[j].getElementsByTagName("tr");

    for (var k = 1; k < rows.length; k += 2)
    {
      Core.addClass(rows[k], "alt");
    }
  }
}
```

We're already using the variables `i` and `j` as the counters for the outer `for` loops, and we don't want to overwrite their values, so we create a new counter variable called `k`. `k` starts at 1 (the second index), and for every execution of this inner loop we increase its value by 2.

The conditional code for this inner loop is just one line that uses our pre-rolled `Core.addClass` function to add the class `alt` to the current row. Once the inner `for` loop finishes, every second row will be marked with this class, and once the outer `for` loops finish, every data table will be stripy.

Putting it All Together

The main code for our function is now complete; we just have to wrap it inside a self-contained object:

stripy_tables.js *(excerpt)*

```
var StripyTables =
{
  init: function()
  {
    var tables = Core.getElementsByClass("dataTable");

    for (var i = 0; i < tables.length; i++)
    {
      var tbodys = tables[i].getElementsByTagName("tbody");

      for (var j = 0; j < tbodys.length; j++)
      {
        var rows = tbodys[j].getElementsByTagName("tr");

        for (var k = 1; k < rows.length; k += 2)
        {
          Core.addClass(rows[k], "alt");
        }
      }
    }
  }
};
```

Kick-start it when the page loads, using `Core.start`:

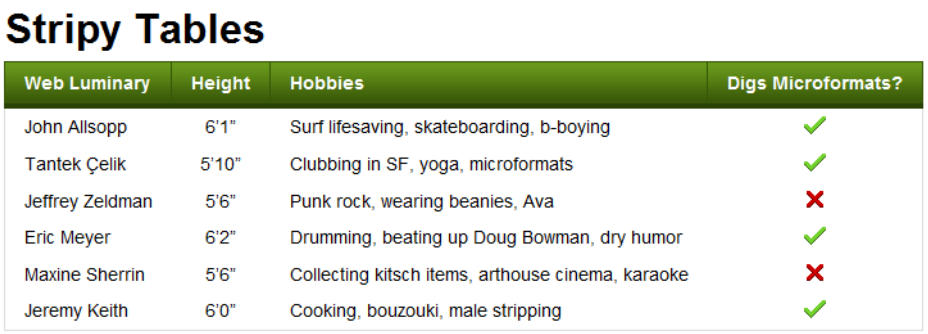
stripy_tables.js *(excerpt)*

```
Core.start(StripyTables);
```

Now, whenever you include this script file (and the Core library) on your page, `StripyTables` will go into action to automatically stripe all your tables:

stripy_tables.html *(excerpt)*

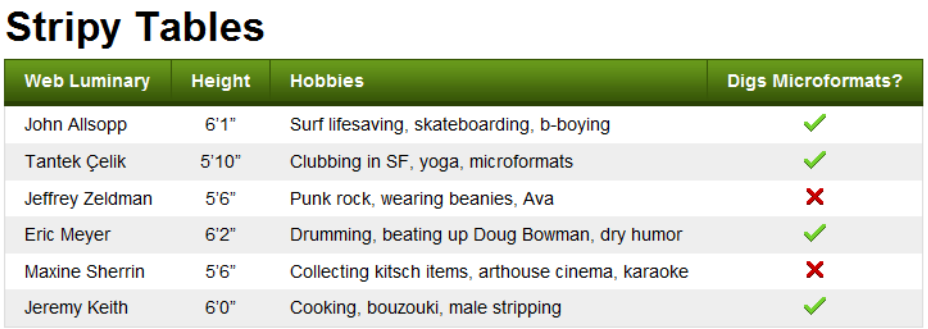
```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en-US">
  <head>
    <title>Stripy Tables</title>
    <meta http-equiv="Content-Type"
```



Stripy Tables

Web Luminary	Height	Hobbies	Digs Microformats?
John Allsopp	6'1"	Surf lifesaving, skateboarding, b-boying	✓
Tantek Çelik	5'10"	Clubbing in SF, yoga, microformats	✓
Jeffrey Zeldman	5'6"	Punk rock, wearing beanies, Ava	✗
Eric Meyer	6'2"	Drumming, beating up Doug Bowman, dry humor	✓
Maxine Sherrin	5'6"	Collecting kitsch items, arthouse cinema, karaoke	✗
Jeremy Keith	6'0"	Cooking, bouzouki, male stripping	✓

Figure 3.11. Hard-to-scan table content without stripes



Stripy Tables

Web Luminary	Height	Hobbies	Digs Microformats?
John Allsopp	6'1"	Surf lifesaving, skateboarding, b-boying	✓
Tantek Çelik	5'10"	Clubbing in SF, yoga, microformats	✓
Jeffrey Zeldman	5'6"	Punk rock, wearing beanies, Ava	✗
Eric Meyer	6'2"	Drumming, beating up Doug Bowman, dry humor	✓
Maxine Sherrin	5'6"	Collecting kitsch items, arthouse cinema, karaoke	✗
Jeremy Keith	6'0"	Cooking, bouzouki, male stripping	✓

Figure 3.12. Using a script to produce stripy tables and improve the usability of the document

```
        content="text/html; charset=utf-8" />
    <link rel="stylesheet" type="text/css"
        href="stripy_tables.css" />
    <script type="text/javascript" src="core.js"></script>
    <script type="text/javascript"
        src="stripy_tables.js"></script>
```

You can style the `alt` class however you want with a simple CSS rule:

stripy_tables.css *(excerpt)*

```
tr.alt
{
  background-color: #EEEEEE;
}
```

You can turn a plain, hard-to-follow table like the one in Figure 3.11 into something that's much more usable—like that pictured in Figure 3.12—with very little effort.

This type of script is a great example of progressive enhancement. Users who browse with JavaScript disabled will still be able to access the table perfectly well; however, the script provides a nice improvement for those who can run it.

Exploring Libraries

Most of the available JavaScript libraries have little helper functions that can help you expand the functionality of the DOM. These range from neat little shortcuts to entirely different ways of finding and manipulating elements.

Prototype

Prototype was one of the first libraries to swap the painful-to-type `document.getElementById` for the ultra-compact `$`.

The `$` function in Prototype not only acts as a direct substitute for `document.getElementById`, it also expands upon it. You can get a reference to a single element by ID, so this normal code:

```
var money = document.getElementById("money");
```

would become:

```
var money = $("money");
```

But you don't have stop at getting just one element; you can specify a whole list of element IDs that you want, and `$` will return them all as part of an array. So this normal code:

```
var elementArray = [];
elementArray[0] = document.getElementById("kroner");
elementArray[1] = document.getElementById("dollar");
elementArray[2] = document.getElementById("yen");
```

becomes considerably shorter:

```
var elementArray = $("kroner", "dollar", "yen");
```

Earlier in this chapter we created our own library function to get elements by class. Prototype has a similar function, which is slightly more powerful. It creates an extension to the document node, called `getElementsByClassName`. Like our function `Core.getElementsByClass`, this method allows us to retrieve an array of elements that have a particular class:

```
var tables = document.getElementsByClassName("dataTable");
```

It also takes an optional second argument, which allows us to specify a parent element under which to search. Only elements that are descendants of the specified element, and have a particular class, will be included in the array:

```
var tables =
    document.getElementsByClassName("dataTable", $("content"));
```

The variable `tables` will now be an array containing elements that are descendants of the element with ID `content`, and that have a class of `dataTable`.

Prototype also replicates all of the class functions that we created for our own library. These functions take exactly the same arguments that ours did, but the functions themselves are methods of Prototype's `Element` object. So Prototype offers `Element.hasClassName`, `Element.addClassName`, and `Element.removeClassName`:

```
var body = document.getElementsByTagName("body")[0];
Element.addClassName(body, "unreadable");

if (Element.hasClassName(body, "unreadable"))
{
  Element.removeClassName(body, "unreadable");
}
```

jQuery

jQuery was one of the first libraries to support an entirely different way of finding elements with JavaScript: it allows us to find groups of elements using CSS selectors.

The main function in jQuery is also called `$`, but because it uses CSS selectors, this function is much more powerful than Prototype's version, and manages to roll a number of Prototype's functions into one.[3]

If you wanted to use jQuery to get an element by ID, you'd type the following:

```
var money = $("#money");
```

`#` indicates an ID selector in CSS, so `$("#money")` is the equivalent of typing `document.getElementById("money")`.

To get a group of elements by tag name, you'd pass `$` a CSS element type selector:

```
var paragraphs = $("p");
```

And to get a group of elements by class, you'd use a class selector:

```
var tables = $(".dataTable");
```

And, as with CSS, you can combine all these simple selector types in, say, a descendant selector:

```
var tables = $("#content table.dataTable");
```

`tables` is now an array of table elements that are descendants of the element with ID `content`, and that have a class of `dataTable`.

The CSS rule parsing in jQuery is really quite spectacular, and it supports the majority of selectors from CSS1, CSS2, and CSS3, as well as XPath.[4] This makes it possible for us to use selectors like this:

```
var complex = $("form > fieldset:only-child input[@type=radio]");
```

[3] In fact, based on the popularity of this feature in jQuery, Prototype went on to include similar functionality in a function named `$$`.

[4] XPath is a zany language for selecting nodes from XML documents (including XHTML documents). While XPath is extremely powerful, the process of learning it is likely to give you a facial tick.

Once you break it down, that query finds all radio button `input` elements inside `fieldsets` that are direct children of form elements, but only where the `fieldset` is the only child of the form. Phew!

Dojo

Dojo follows the previous two libraries closely in how they deal with the DOM.

It has its own shortcut to `document.getElementById`, but it doesn't expand upon the DOM's native functionality:

```
var money = Dojo.byId("money");
```

It also has its own `getElementsByClass` function inside the `html` module:

```
var tables = dojo.html.getElementsByClass("dataTable");
```

This function allows you to get elements by class under a particular parent:

```
var tables = Dojo.html.getElementsByClass("dataTable",
    dojo.byId("content"));
```

For completeness, it has the usual class handling functions, which take the same form as our own `Core` functions:

```
var body = document.getElementsByTagName("body")[0];
Dojo.html.addClass(body, "unreadable");

if (Dojo.html.hasClass(body, "unreadable"))
{
  Dojo.html.removeClass(body, "unreadable");
}
```

Summary

An understanding of the DOM is central to using JavaScript, which is why the use of JavaScript on the Web is sometimes referred to as "DOM scripting."

As you delve further into this book, and we begin to look at more complex interfaces, our manipulation of the DOM will also become more complex, so your familiarity with the basics presented in this chapter is vital.

In the next chapter, we take a look at events, which allow your JavaScript programs to respond to users' interactions with your web pages. Dynamic interfaces, here we come!

Chapter 4

Events

When we started out in Chapter 2, every script that we looked at would be loaded by the browser and executed right away. Since then, we've learned how to wait until the HTML document has finished loading before unleashing the awesome power of JavaScript. In every case, however, the script will work its magic (for instance, making the rows of your tables stripy), then fizzle out, leaving behind an enhanced—but still very static—page.

You don't need to suffer with scripts that peak too quickly! With the simple techniques we'll explore in this chapter, you'll learn to take control, write scripts that last longer, and become a superstar … well, in your favorite JavaScript chat room, anyway.

Don't get me wrong—scripts that enhance web pages the instant they're loaded (let's call them "quickies") have their place, but there are limits to how much they can improve the user experience of a site. JavaScript *really* gets interesting when you start to write scripts that are triggered by **events** that occur during the user's interaction with the page, like clicking a hyperlink, scrolling the browser's viewport, or typing a value into a form field.

An Eventful History

Thanks to the wide adoption of the Document Object Model (DOM) standard, accessing HTML elements in your JavaScript code works very similarly in every browser. If only the same could be said for every aspect of JavaScript! As it happens, running JavaScript code in response to an event stands out as one of the few remaining features that are implemented in wildly varying ways in current browsers.

The news isn't all bad. There is a certain amount of common ground. For as long as they've supported JavaScript, browsers have had a simple model for dealing with events—using event handlers—and all current browsers provide compatible support for these handlers, despite the fact that a complete standard was never written for them.[1] As we'll see, these techniques come with limitations that you'll want to avoid when you can, but they offer a good fallback option.

DOM Level 0

The first version of the W3C DOM specification was called Document Object Model Level 1. Since event handlers (along with a number of other nonstandard JavaScript features) predate this specification, developers like to call them Document Object Model Level 0.

Stepping into the 21st century, the World Wide Web Consortium (W3C) has developed the DOM Level 2 Events standard,[2] which provides a more powerful means of dealing with events, called event listeners. Almost all browsers now support this standard, the notable exception being Internet Explorer up to and including IE 7.0. Internet Explorer has its own way of doing things, and though its approach is almost as powerful, it's also sufficiently different to force us to write extra code to cater for this popular browser.

It's interesting to note that Microsoft participated in the development of the DOM Level 2 Events specification within the W3C, but when it came time to release IE 5.5, Microsoft chose not to support the specification in that browser. In the two

[1] The HTML 4 specification briefly discusses them under the heading Intrinsic Events [http://www.w3.org/TR/html4/interact/scripts.html#h-18.2.3].
[2] http://www.w3.org/TR/DOM-Level-2-Events/

major releases of Internet Explorer since then (IE 6.0 and 7.0), there has been no sign of Microsoft adding support for this standard.

Thankfully, we don't have to wait for Microsoft. The benefits of using event listeners, be they the W3C standard version or Internet Explorer's peculiar alternative, are so great that legions of dedicated geeks have investigated the incompatibilities and come up with reasonable solutions. With a little work, we can build these solutions into our `Core` library so that we can use event listeners freely, without encountering browser compatibility issues.

Event Handlers

The simplest way to run JavaScript code in response to an event is to use an **event handler**. Event handlers have been around for as long as browsers have supported JavaScript, and predate the DOM standard. An event handler is a JavaScript function that's "plugged into" a node in the DOM so that it's called automatically when a particular event occurs in relation to that element. Figure 4.1 illustrates this concept.

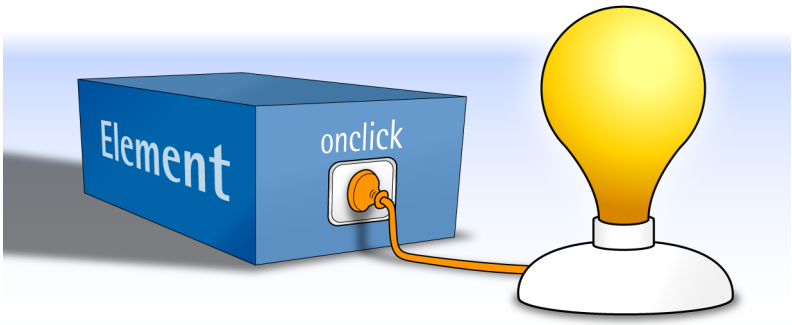


Figure 4.1. Plugging in a single event handler function to respond to a particular event

Let's start with an obvious example—the user clicking on a link like this:

linkhandler.html *(excerpt)*

```
<p>The first captain of the USS Enterprise NCC-1701 was
  <a id="wikipedia" href="http://en.wikipedia.org/…">Christopher
  Pike</a>.</p>
```

When a user clicks on a link like this one, the browser generates a `click` event. By default, the browser will respond to that `click` event by navigating to the URL specified by the link. But before this happens, we can plug in our own event handler to respond to the event.

Let's say you want to display an alert to notify users that they're leaving your site. An event handler is just a JavaScript function, so we can write a function to present the alert. As usual, we don't want to interfere with other scripts, so we'll wrap this function in an object with a unique name:

linkhandler.js *(excerpt)*

```
var wikipediaLink =
{
  clickHandler: function()
  {
    alert("Don't believe everything you read on Wikipedia!");
  }
};
```

Setting up a function as an event handler is easy. All you need is a reference to the DOM element for which you want to handle events. Then, you set the element's `on`*`event`* property, where *event* is the type of event you want to handle:

```
element.onevent = eventHandler;
```

To handle `click` events for the `wikipedia` link above with our `clickHandler` function (which is a method of our `wikipediaLink` object), we write this code:

linkhandler.js *(excerpt)*

```
var link = document.getElementById("wikipedia");
link.onclick = wikipediaLink.clickHandler;
```

But there's a catch: we can't assign an event handler to our element until the element has loaded. Thankfully, we already know how to write code that's executed only after the entire document is loaded:

linkhandler.js

```
var wikipediaLink =
{
  init: function()
  {
    var link = document.getElementById("wikipedia");
    link.onclick = wikipediaLink.clickHandler;
  },

  clickHandler: function()
  {
    alert("Don't believe everything you read on Wikipedia!");
  }
};

Core.start(wikipediaLink);
```

The code for this example is deceptively simple. As Figure 4.2 reveals, our code is actually executed in three stages:

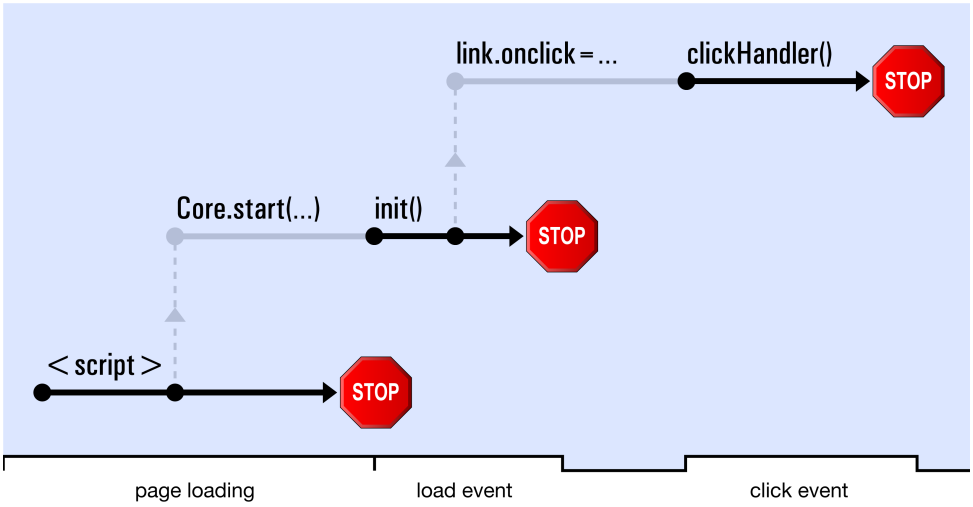


Figure 4.2. The three stages of script execution

1. The browser encounters the `<script>` tag in the HTML document's header and loads the JavaScript file. Our code declares the `wikipediaLink` object, then

calls `Core.start` to request that the object's `init` method be called when the whole document has loaded.

2. The page finishes loading, and the `wikipediaLink` object's `init` method is called. This method finds the `wikipedia` link and sets up the `clickHandler` method as its `click` event handler.

3. The user clicks the link, which generates a `click` event. The browser calls `clickHandler`, the link's `click` event handler, which displays the alert shown in Figure 4.3.

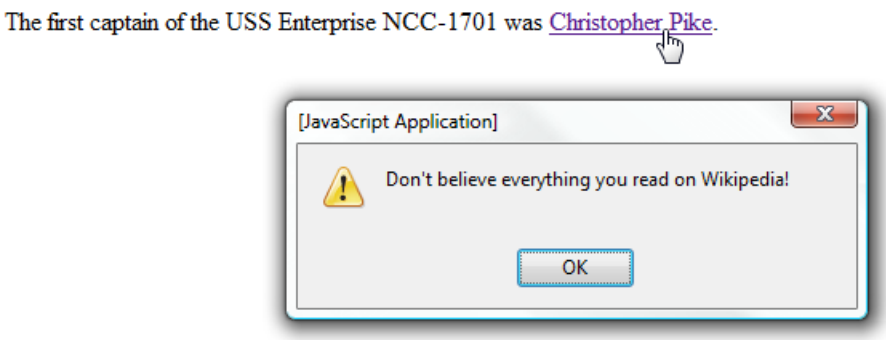


Figure 4.3. The event handler in action

Once the user clicks the **OK** button to dismiss the alert, the browser follows the link as normal.

Event Handlers as HTML Attributes

If you go looking, you'll find that a lot of sites set up JavaScript event handlers using HTML attributes, like this:

```
<a href="…" onclick="JavaScript code here">…</a>
```

As I mentioned in Chapter 1, this is the JavaScript equivalent of assigning CSS properties to your elements using the HTML `style` attribute. It's messy, it violates the principle of keeping code for dynamic behavior separate from your document content, and it's *so* 1998.

Default Actions

As we've just seen, event handlers let you respond to user actions by running any JavaScript code you like. But often, the browser still gets the last word. Take the example we just saw: no matter how creative and amazing the code in our event handler, when it's done, the browser will take over and follow the link as normal. I don't know about you, but I call that being a slave to the Man, and I won't take it.

Browsers take all sorts of actions like this:

- They follow links that users click.
- They submit forms when users click a **Submit** button, or hit **Enter**.
- They move keyboard focus around the page when the user hits **Tab**.

These are called **default actions**—things the browser *normally* does in response to events. In most cases you'll want the browser to do these things, but sometimes you'll want to prevent them from occurring.

The easiest way to stop the browser from performing a default action in response to an event is to create for that event an event handler that returns `false`. For example, we can modify the link `click` event handler we created above to ask the user for confirmation before the link is followed:

clickprompt.js *(excerpt)*

```
clickHandler: function()
{
  if (!confirm("Are you sure you want to leave this site?"))
  {
    return false;
  }
}
```

The `confirm` function used in this code is built into the browser, just like `alert`. And it displays a message to the user just like `alert` does, except that it offers the user two buttons to click: **OK** and **Cancel**. If the user clicks **OK**, the function returns `true`. If the user clicks **Cancel**, the function returns `false`. We then use the `!` operator introduced in Table 2.1 to reverse that value so that the body of the `if` statement is executed when the user clicks **Cancel**.

As shown in Figure 4.4, this new code prompts the user with the message "Are you sure you want to leave this site?" and causes our `clickHandler` method to return `false` if the user clicks **Cancel**. This, in turn, prevents the browser from performing the default action for the `click` event, so the browser does not follow the link.

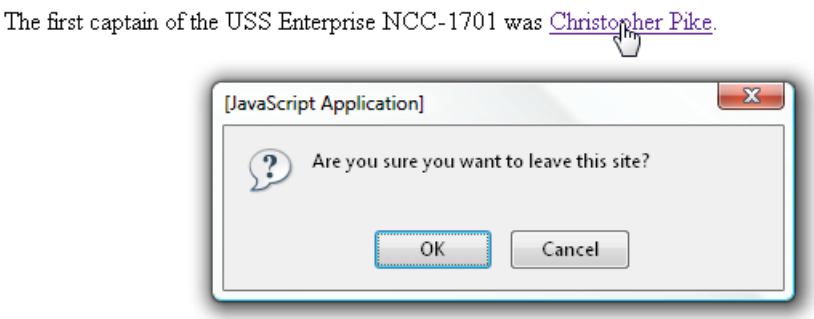


Figure 4.4. The user choosing whether or not to take the default action

Cutting Down on Code

If you're feeling especially zen, you might have spotted the fact that `confirm` returns `false` when we want `clickHandler` to return `false`. Since these values match, you can simplify the code of `clickHandler` if you want to:

```
clickHandler: function()
{
  return confirm(
      "Are you sure you want to leave this site?");
}
```

This version of the code simply returns whatever `confirm` returns, which turns out to be exactly what we want.

The `this` Keyword

So far, we've created one event handler that handles one particular event occurring on one particular HTML element—pretty pathetic, if you ask me. The real fun is in writing an event handler that can handle events for *many* HTML elements!

Now, while assigning an event handler to many different elements is relatively straightforward, making it do something *sensible* for each element can be tricky, and that's where the `this` keyword comes into play.

A popular request from web development clients is for links to external sites to open in a new browser window or tab. However you feel about this in principle, the way a link opens is considered a part of the *behavior* of the page, and thus should be controlled by your JavaScript code (if at all).[3]

To open a URL in a new window or tab, simply use JavaScript's built-in `open` function:[4]

```
open(URL);
```

Writing a `click` event handler that opens a particular URL in a new window is therefore trivial:

```
clickHandler: function()
{
  open("http://en.wikipedia.org/wiki/Christopher_Pike");
  return false;
}
```

But how do we do this for *every* external link in the page? We definitely don't want to write a separate `click` handler for every external link.

The solution is to write a single handler that can retrieve the URL of the link that has just been clicked, using the `this` keyword:

[3] In past versions of the HTML standard, you could set the `target` attribute of a link to control how it would open. This attribute was deprecated in HTML 4 in favor of JavaScript alternatives.

[4] You'll see this referred to as `window.open` in many other books and online tutorials, because technically, all built-in functions are actually methods of the global `window` object. But you don't see them talking about `window.alert`, do you?

newwindow.js *(excerpt)*

```
clickHandler: function()
{
  open(this.href);
  return false;
}
```

`this` is a special JavaScript keyword that behaves like a variable, except that you can't assign it a value—its value is the object upon which the currently-executing function was invoked as a method. In other words, if you call *`object.method`*`()`, then within the code of *`method`*, the `this` keyword will refer to *`object`*. When the code currently being executed is not within a function, or when the function was not called as a method of an object, then `this` points to the global object that contains all global variables and functions.

Since the browser calls an event handler as a method of the element on which the event occurred, you can use `this` within an event handler to get a reference to that element. In the above code, we use `this` to get a reference to the link that the user has clicked, then use its `href` property to obtain the URL to which the link points.

By using `this` to retrieve from the element itself the information we need in order to respond to an event, we have created an event handler that can be assigned to all of the external links on the page. We just need to identify them with a class in our HTML code, and use that in our script's `init` method:

newwindow.js

```
var externalLinks =
{
  init: function()
  {
    var extLinks = Core.getElementsByClass("external");

    for (var i = 0; i < extLinks.length; i++)
    {
      extLinks[i].onclick = externalLinks.clickHandler;
    }
  },
```

```
  clickHandler: function()
  {
    open(this.href);
    return false;
  }
};

Core.start(externalLinks);
```

The Problem with Event Handlers

Many JavaScript developers like event handlers because they're simple to use, and they work in all browsers. Unfortunately, they come with one big, honking limitation: *you can only assign one event handler to a given event on a given HTML element.*

In simple terms, you can't easily make more than one thing happen when an event occurs. Consider this code:

```
element.onclick = script1.clickHandler;
element.onclick = script2.clickHandler;
```

Only the `clickHandler` in *`script2`* will be executed when a click occurs on *`element`*, because assigning the second event handler replaces the first.

You might wonder if we really *need* to assign more than one event handler. After all, how often are you going to want more than one script to respond to the `click` event of a link? And as long as we were just talking about `click` events, you'd be right to wonder.

But there are all sorts of events that you can respond to, and for some of them it would be extremely useful to have multiple handlers. As we'll see in Chapter 6, for example, a form's `submit` event often requires multiple scripts to check that the various form fields have been filled out correctly.

The commonly used workaround to this problem is to assign as the event handler a function that calls multiple event handling functions:

```
element.onclick = function()
{
  script1.clickHandler();
  script2.clickHandler();
}
```

But all sorts of things are wrong with this approach:

- `this` will no longer point to the *`element`* within the `clickHandler` methods.
- If either `clickHandler` method returns `false`, it will not cancel the default action for the event.
- Instead of assigning event handlers neatly inside a script's `init` method, you have to perform these assignments in a separate script, since you have to reference both *`script1`* and *`script2`*.

There are solutions to all of these problems, of course, but they involve complicated and twisty code[5] that you really shouldn't have to deal with to accomplish something as basic as responding to events.

In addition to the simple event handlers we've just looked at, most browsers today have built-in support for a more advanced way of handling events: event listeners, which do not suffer from the one-handler-only restriction.

Event Listeners

The good news is that **event listeners** are just like event handlers, except that you can assign as many event listeners as you like to a particular event on a particular element, and there is a W3C specification that explains how they should work.[6]

The bad news is that Internet Explorer has its own completely different, and somewhat buggy version of event listeners that you also need to support if you want your scripts to work in that browser. Oh, and sometimes Safari likes to do things slightly differently, too.

[5] http://dean.edwards.name/weblog/2005/10/add-event/
[6] http://www.w3.org/TR/DOM-Level-2-Events/

Like an event handler, an event listener is just a JavaScript function that is "plugged into" a DOM node. Where you could only plug in one event handler at a time, however, you can plug multiple listeners in, as Figure 4.5 illustrates.

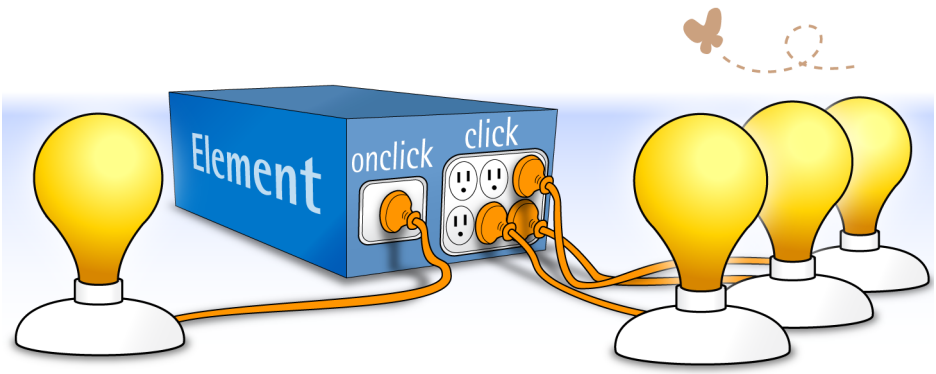


Figure 4.5. Plugging in only one handler, but many listeners

The code that sets up an event listener is quite different from that used to set up an event handler, but it's still fairly easy:

```
element.addEventListener("event", eventListener, false);
```

In browsers that support W3C-standard event listeners, the `addEventListener` method is available on every object that supports events. This method takes three arguments: the name of the event to which you want to assign the listener (e.g. `"click"`), the listener function itself, and a Boolean value that you'll usually want to set to `false` (more on this last argument in the section called "Event Propagation").

To set up an event listener in Internet Explorer, however, you need to use a method called `attachEvent`. This method works a lot like `addEventListener`, but it takes slightly different arguments:

```
element.attachEvent("onevent", eventListener);
```

Spot the differences? The first argument—the name of the event you're interested in—must be prefixed with `on` (for example, `"onclick"`), and there is no mysterious third argument.

Any script that uses event listeners will need to use `addEventListener` for all browsers that support it, and `attachEvent` for Internet Explorer browsers that don't. Ensuring that your script uses the right method is a simple matter of using an `if-else` statement that checks if the `addEventListener` or `attachEvent` methods exist in the current browser:

```
if (typeof element.addEventListener != "undefined")
{
  element.addEventListener("event", eventListener, false);
}
else if (typeof element.attachEvent != "undefined")
{
  element.attachEvent("onevent", eventListener);
}
```

This is another example of the object detection technique that we first saw at work in Chapter 3.

Let's employ this technique to display an alert in response to the `click` event of a particular link, as we did using event handlers earlier in this chapter:

linklistener.js *(excerpt)*

```
var wikipediaLink =
{
  init: function()
  {
    var link = document.getElementById("wikipedia");

    if (typeof link.addEventListener != "undefined")
    {
      link.addEventListener(
          "click", wikipediaLink.clickListener, false);
    }
    else if (typeof link.attachEvent != "undefined")
    {
      link.attachEvent("onclick", wikipediaLink.clickListener);
    }
  },

  clickListener: function()
  {
```

```
    alert("Don't believe everything you read on Wikipedia!");
  }
};

Core.start(wikipediaLink);
```

It's not as simple as setting up an event handler, of course, but this code isn't too complex, and it allows for another script to add its own `click` event listener to the link without dislodging the one we've set up here.

Although you'll usually just add event listeners to your DOM nodes and forget about them, you can "unplug" an event listener from a DOM node if you need to. In the W3C's standard event listener model, we use the `removeEventListener` method to achieve this, whereas in Internet Explorer, we use `detachEvent`. In either case, we pass the method the same arguments we passed when adding the listener:

```
if (typeof element.removeEventListener != "undefined")
{
  element.removeEventListener("event", eventListener, false);
}
else if (typeof element.detachEvent != "undefined")
{
  element.detachEvent("onevent", eventListener);
}
```

Default Actions

You'll remember that you can simply return `false` from an event handler in order to prevent the browser from carrying out the default action for an event, such as following a clicked hyperlink. Event listeners let you do this too, but in a slightly different way.

In the W3C standard event listener model, the browser will always pass an **event object** to the event listener function. The event object's properties contain information about the event (for instance, the position of the cursor when the event occurred), while its methods let us control how the event is processed by the browser.

In order to prevent the browser from performing the default action for an event, we simply call the event object's `preventDefault` method:

```
clickListener: function(event)
{
  if (!confirm("Are you sure you want to leave this site?"))
  {
    event.preventDefault();
  }
}
```

If multiple listeners are associated with an event, any one of those listeners calling `preventDefault` is enough to stop the default action from occurring.

Internet Explorer's event model is, of course, similar but different. In Internet Explorer, the event object isn't passed to the event listener as an argument; it's available as a global variable named `event`. Also, the event object doesn't have a `preventDefault` method; instead, it has a property named `returnValue` that we can set to `false` in order to prevent the default action from taking place:

```
clickListener: function()
{
  if (!confirm("Are you sure you want to leave this site?"))
  {
    event.returnValue = false;
  }
}
```

Again, using the technique of object detection to figure out which event model the current browser supports, we can write an event listener that's able to cancel the default action in either event model:

clickpromptlistener.js *(excerpt)*

```
clickListener: function(event)
{
  if (typeof event == "undefined")
  {
    event = window.event;
  }

  if (!confirm("Are you sure you want to leave this site?"))
  {
    if (typeof event.preventDefault != "undefined")
```

```
    {
      event.preventDefault();
    }
    else
    {
      event.returnValue = false;
    }
  }
}
```

At the start of this listener, we check if we've actually been passed an event object as an argument according to the W3C event model. If we haven't, we set our `event` variable to `window.event`, which is Internet Explorer's global event object. We refer to it as `window.event` instead of just `event` because our function already has its own variable named `event`.

Then, when it comes time to cancel the default action, we check to see whether or not the event object has a `preventDefault` method. If it does, we call it. If it doesn't, we set the object's `returnValue` property to `false` instead. Either way, the default action is prevented.

Preventing Default Actions in Safari 2.0.3 and Earlier

Although it did an admirable job of supporting the rest of the DOM 2 Events standard, prior to version 2.0.4 the Safari browser could not stop a default action from occurring in an event listener. The `preventDefault` method was there; it just didn't do anything.

As I write this, a lot of Mac users are still using Safari 1.2, which is affected by this issue. If you need to support Safari version 2.0.3 or earlier, the only way to cancel a default action is to use an old-style event handler. If you're lucky enough to be working on a script that will *always* cancel the default event, you can use an event listener in combination with an event handler that simply returns `false`:

```
element.onevent = function()
{
  return false;
}
```

Event Propagation

Obviously, if you stick a `click` event listener on a hyperlink, then click on that link, the listener will be executed. But if, instead, you assign the `click` listener to the paragraph containing the link, or even the document node at the top of the DOM tree, clicking the link will still trigger the listener. That's because events don't just affect the target element that generated the event—they travel through the tree structure of the DOM. This is known as **event propagation**, and I should warn you: it's not sexy or exciting.

The W3C event model is very specific about how event propagation works. As illustrated in Figure 4.6, an event propagates in three phases:

1. In the **capture phase**, the event travels down through the DOM tree, visiting each of the target element's ancestors on its way to the target element. For example, if the user clicked a hyperlink, that `click` event would pass through the document node, the `html` element, the `body` element, and the paragraph containing the link.

   At each stop along the way, the browser checks for **capturing event listeners** for that type of event, and runs them.

   What's that? You don't know what a capturing event listener is? Remember when I mentioned the third argument of the `addEventListener` method and I told you that you'd usually want to set it to `false`? Well, if you set it to `true`, you'll create a capturing event listener.

   You'll also recall that Internet Explorer's `attachEvent` method doesn't support a third argument. That's because Internet Explorer's event model doesn't have a capture phase. Consequently, most developers avoid using capturing event listeners.

2. In the **target phase**, the browser looks for event listeners that have been assigned to the target of the event, and runs them. The target is the DOM node on which the event is focused. For example, if the user clicks a hyperlink, the target node is the hyperlink.[7]

[7] This is the case except in Safari, where the target is actually the text node *inside* the hyperlink. The W3C events specification is ambiguous about which behavior is correct, but in practice it doesn't make

3. In the **bubbling phase**, the event travels back up the DOM tree, again visiting the element's ancestors one by one until it reaches the document node. At each stop along the way, the browser checks for event listeners that are *not* capturing event listeners, and runs them.

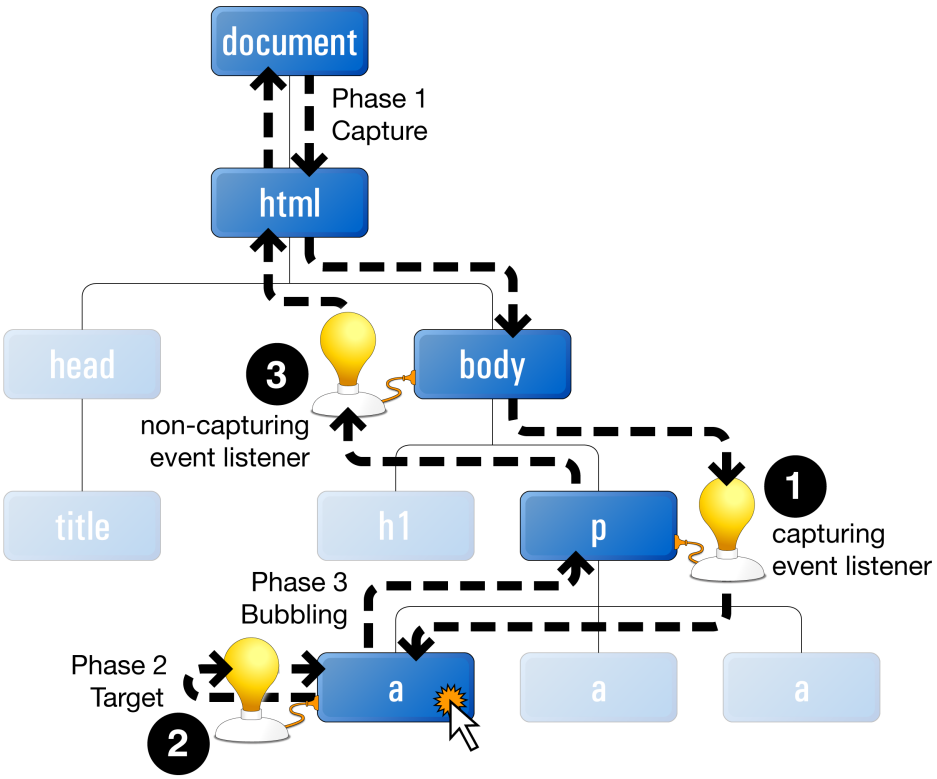


Figure 4.6. Standard event propagation

Not All Events Bubble

All events go through the capture and target phases, but certain events skip the bubbling phase. Specifically, `focus` and `blur` events, which occur when keyboard focus is given to and removed from an element, respectively, do not bubble. In most cases, this isn't a detail you need to lose much sleep over, but as we'll see

a big difference, since an event listener that's assigned to the hyperlink itself will still be triggered in the bubbling phase.

in the section called "Example: Accordion" later in this chapter, it'll make your life more difficult every once in a while.

So why am I boring you with all these details on event propagation? After all, you can assign an event listener to an element on which you expect an event to occur, and your listener will run when that event occurs. Does it have to be any more complicated than that? In most cases, no—it doesn't. But sometimes you want to get a little more creative, and creativity inevitably breeds complexity. Stay with me, here.

Let's say you were an especially helpful sort of person—I'm talking about the kind of helpful that most people find annoying. You might want to display a "helpful" message if a user were to accidentally click on a part of your page that wasn't a hyperlink:

strayclickcatcher.js *(excerpt)*

```
strayClickListener: function(event)
{
  alert("Did you mean to click a link? " +
      "It's that blue, underlined text.");
},
```

In order to catch clicks anywhere in the document, you can just assign this as a `click` listener for the document node. The listener will be triggered in the bubbling phase of every `click` event, no matter where in the document the target element is located.

But how do you keep the message from appearing when the user *does* click a link? What you need to do is prevent *those* `click` events from bubbling up to trigger your event listener. To do this, you need to set up another event listener that stops the propagation of those events.

To stop the propagation of an event in the W3C event model, you call the `stopPropagation` method of the event object that's passed to your event listener. In Internet Explorer's event model there's no such method; instead, you need to set the `cancelBubble` property of the event object to `true`. Here's what the resulting listener looks like:

strayclickcatcher.js *(excerpt)*

```
linkClickListener: function(event)
{
  if (typeof event == "undefined")
  {
    event = window.event;
  }

  if (typeof event.stopPropagation != "undefined")
  {
    event.stopPropagation();
  }
  else
  {
    event.cancelBubble = true;
  }
}
```

Just assign this second listener function to every link in your document, and it will stop the propagation of `click` events when they reach a link. This prevents those clicks from bubbling up to the document element to trigger the first event listener.

Figure 4.7 shows what happens when you click on a part of the document that doesn't link to anything. If you click on a link, however, the browser will follow it without complaint.

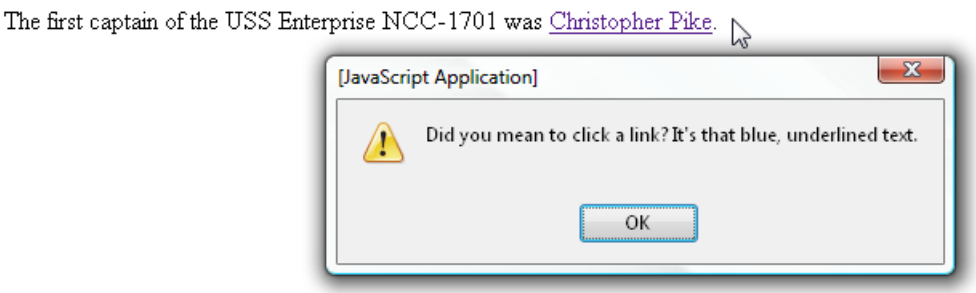


Figure 4.7. A stray click producing a helpful/annoying message

Here's the complete, and very helpful/annoying script:

strayclickcatcher.js

```
var strayClickCatcher =
{
  init: function()
  {
    var links = document.getElementsByTagName("a");

    if (typeof document.addEventListener != "undefined")
    {
      document.addEventListener("click",
          strayClickCatcher.strayClickListener, false);
      for (var i = 0; i < links.length; i++)
      {
        links[i].addEventListener("click",
            strayClickCatcher.linkClickListener, false);
      }
    }
    else if (typeof document.attachEvent != "undefined")
    {
      document.attachEvent("onclick",
          strayClickCatcher.strayClickListener);
      for (var i = 0; i < links.length; i++)
      {
        links[i].attachEvent("onclick",
            strayClickCatcher.linkClickListener);
      }
    }
  },

  strayClickListener: function(event)
  {
    alert("Did you mean to click a link? " +
        "It's that blue, underlined text.");
  },

  linkClickListener: function(event)
  {
    if (typeof event == "undefined")
    {
      event = window.event;
    }

    if (typeof event.stopPropagation != "undefined")
```

```
    {
      event.stopPropagation();
    }
    else
    {
      event.cancelBubble = true;
    }
  }
};

Core.start(strayClickCatcher);
```

The `this` Keyword

Earlier in this chapter, we grappled with the problem of how to write a single event handler that could be applied to many different elements. We learned that we could use `this` to reference the element to which the handler was assigned, and retrieve from it information that we could use to control how the handler would respond. It would be nice if we could do the same with our event listeners … so, can we?

Keep in mind that the value of `this` within a JavaScript function is determined by the way in which a function is called. If it's called as a method of an object, `this` refers to that object. If it's called as a standalone function, `this` refers to the global object that contains all global variables and functions. The question is, how are event listeners called by the browser?

Surprisingly, the W3C event model standard has nothing to say on the subject, so it's left up to each browser to decide which object `this` refers to when it's used within an event listener. Thankfully, every browser that supports the W3C event model calls an event listener as a method of the element to which it was assigned, so `this` refers to that element, just like it does in an event handler.

Less surprisingly, Internet Explorer went a different way with its event listeners: Internet Explorer event listeners are called as standalone functions, so `this` points to the relatively useless global object.

So far, we've been able to deal with every incompatibility between the two event listener models by using simple object detection to run the code required by each

browser. When it comes to solving the issue with `this`, however, things aren't so simple. In case you're curious, here's the solution:

```
if (typeof element.addEventListener != "undefined")
{
  element.addEventListener("event", eventListener, false);
}
else if (typeof element.attachEvent != "undefined")
{
  var thisListener = function()
  {
    var event = window.event;
    if (Function.prototype.call)
    {
      eventListener.call(element, event);
    }
    else
    {
      target._currentListener = eventListener;
      target._currentListener(event);
      target._currentListener = null;
    }
  };
  element.attachEvent("onevent", thisListener);
}
```

Now, leaving aside for the moment the details of how this code works, can you imagine having to type out that entire monstrosity every time you want to set up an event listener? I don't mind telling you that I wouldn't be sitting here writing this book if that's how JavaScript made us do things.

There's a way to make life easier for yourself, and I'll show it to you in the section called "Putting it All Together."

The Internet Explorer Memory Leak

When it comes to making your life difficult, Internet Explorer has one more trick up its sleeve. If the disaster surrounding the `this` keyword wasn't enough to make you want to give up on Internet Explorer's event model, this will.

In Internet Explorer, if you set up for an element an event listener that contains a reference to that element (or indeed, any other node in the DOM), the memory oc-

cupied by that listener and the associated DOM nodes will not be released when the user navigates to another page. I'll grant you that this is pretty technical stuff, but what it comes down to is that certain types of event listeners can cause Internet Explorer to leak memory, thereby causing users' computers to slow down until those users are forced to restart their browsers.

The solution to this issue is to set up a special event listener for the `unload` event of the global `window` object, which represents the browser window containing the page. When the user navigates to another page, the `unload` event will occur and the special event listener will be triggered. Within this event listener, you can take the opportunity to remove all of the event listeners that you set up in your document, thereby preventing them from causing memory leaks.

While this solution is quite elegant, the JavaScript code to make it happen is not. For the moment, I'll spare you the details, because as we'll see in the next section, you shouldn't have to worry about them.

Putting it All Together

Have you been keeping track of all the cross-browser compatibility issues that we need to deal with when using event listeners? That's okay—I have:

- Internet Explorer uses `attachEvent`/`detachEvent` to add and remove event listeners, instead of `addEventListener`/`removeEventListener`.
- Internet Explorer uses an `"on`*`event`*`"` naming style for its events, instead of just `"`*`event`*`"`.
- Internet Explorer uses a global `event` variable instead of passing the event object to the listener as an argument.
- To prevent a default action from taking place, Internet Explorer requires you to set the event object's `returnValue` property, instead of calling its `preventDefault` method.
- Internet Explorer doesn't support the capture phase of event propagation.
- To stop an event from propagating, Internet Explorer requires you to set the event object's `cancelBubble` property, instead of calling its `stopPropagation` method.

- Internet Explorer calls event listeners as standalone functions, rather than as methods, requiring the developer to jump through hoops to get a reference to the target element, instead of just using `this`.
- When using event listeners in a certain way, Internet Explorer leaks memory unless you go to great lengths to clean up all of your event listeners.

… and that's without even getting into the differences when it comes to retrieving the details of keyboard and mouse events.[8]

These problems have led some developers to throw in the towel and just use event handlers. In fact, some experts have gone so far as to write their own event listener systems using event handlers as a foundation.[9] While there is certainly a case to be made for such an approach, I believe in embracing the support for event listeners that's built into most browsers, and saving the complex code for the browser that really needs it—Internet Explorer.

To that end, the **core.js** library that you'll find in the code archive for this book includes four methods that enable you to use event listeners without worrying about any of the issues I listed above. Here's how they work:

`Core.addEventListener`

This method sets up an event listener function for a particular event type on a particular element. It works just like the standard `addEventListener` method, except that we pass the element as the first argument of the method. It doesn't support the creation of capturing event listeners, and therefore doesn't take a final Boolean argument:

```
Core.addEventListener(element, "event", eventListener);
```

In Internet Explorer, this method sets up the event listener so that the event object is passed to the listener function as an argument (just as it is in the standard event model), and so that the listener is called as a method of the element to which it is assigned (so that we can use `this` to refer to that element).

[8] For complete coverage of these headaches, and some of the solutions that are available, pick up a copy of *The JavaScript Anthology* (Melbourne: SitePoint, 2006).
[9] http://dean.edwards.name/weblog/2005/10/add-event/

It also takes care of automatically cleaning up the listener when the document is unloaded, so that it does not cause memory leaks in Internet Explorer.

`Core.removeEventListener`

This method removes an event listener that was previously added to an element. It works just like the standard `removeEventListener` method, except that we pass the element as the first argument of the method, and like `Core.addEventListener`, it takes no final Boolean argument:

```
Core.removeEventListener(element, "event", eventListener);
```

`Core.preventDefault`

This method prevents the default action associated with an event from occurring. It works just like the standard `preventDefault` method, except that we pass the event object as an argument of the method:

```
Core.preventDefault(event);
```

`Core.stopPropagation`

This method stops the event from propagating further, and potentially triggering event listeners assigned to ancestors of the element to which the current event listener is assigned:

```
Core.stopPropagation(event);
```

And just like that, all your event listener compatibility headaches vanish! Through the rest of this book, whenever we deal with event listeners, we'll use these methods.

In fact, if you've been following along up until this point, you'll already have used one of these methods without knowing it! The `Core.start` method that we used to start running a script only once the document had finished loading relies on `Core.addEventListener`. Take a look for yourself:

core.js *(excerpt)*

```
Core.start = function(runnable)
{
  Core.addEventListener(window, "load", runnable.init);
};
```

As you can see, `Core.start` simply sets up the `init` method of the script you pass it as an event listener for the `load` event of the `window` object, which represents the browser window that contains the current page.

Now, you might be feeling a little uneasy about trusting those four methods to handle all of your event listening tasks without having seen how they work. If you aren't, you should be! As you can imagine, a lot of developers have put their minds to solving these problems, and the solutions they've produced have not always been particularly good. How do you know that this solution is the one you should be using?

Before drinking the Kool-Aid,[10] you should take a look at Appendix A, in which the inner workings of the four methods are described in detail. Some of the code involved is rather advanced, and uses features of JavaScript that we won't talk about until later in this book—if at all. But if nothing else, the discussion there should let you rest assured that we've done our homework.

Now that we have these shiny, new event listener methods in hand, let's use them for something more exciting than displaying an alert box when the user clicks on the page.

Example: Rich Tooltips

In some browsers, when the user mouses over, or gives keyboard focus to a link (for instance, when **Tab**bing to it), a tooltip will appear, displaying the value of the link's `title` attribute. In many browsers, however, this attribute is never displayed to the user, or is limited to a brief, one-line description. In any case, plain-text

[10] The closest that many geeks will come to making a pop culture reference, "drinking the Kool-Aid" refers to the all-too-common practice of enthusiastically embracing a library or other programming aid without understanding how it works (or doesn't work, as the case may be).

tooltips look fairly boring, and tend to be overlooked by users even when they *are* displayed. But, as shown in Figure 4.8, we can use JavaScript—with a healthy dose of event listeners—to produce our own, more eye-catching tooltips.

Figure 4.8. A rich tooltip

The Static Page

Let's start by looking at the HTML code for this example. We want the browser's standard tooltips to display if the user has JavaScript disabled for some reason, so we'll code our hyperlinks with the tooltip text in the `title` attribute as usual:

tooltips.html *(excerpt)*

```
<p><a class="federation" title="Read more …" href="…">James
  Tiberius Kirk</a> (2233 - 2293/2371), played by William Shatner,
  is the leading character in the original Star Trek TV series and
  the films based on it. Captain Kirk commanded the starship
  Enterprise (<a class="federation" title="Read more …"
      href="…">NCC-1701</a> and later <a class="federation"
      title="Read more …" href="…">NCC-1701-A</a>).</p>
```

Making Things Happen

With a finely-crafted static page all ready to go, we can look at setting up some event listeners to make the changes we want to occur in response to user events.

For each link that has a `title` attribute, we want to show a rich tooltip in two situations: when the cursor hovers over the link (a `mouseover` event), or the link receives keyboard focus (a `focus` event). When the mouse is moved away (a `mouseout` event), or keyboard focus is removed (a `blur` event), we want to hide that tooltip. You should be getting fairly good at belting out the `init` method for a script like this:

tooltips.js *(excerpt)*

```
var Tooltips =
{
  init: function()
  {
    var links = document.getElementsByTagName("a");

    for (var i = 0; i < links.length; i++)
    {
      var title = links[i].getAttribute("title");

      if (title && title.length > 0)
      {
        Core.addEventListener(links[i], "mouseover",
            Tooltips.showTipListener);
        Core.addEventListener(links[i], "focus",
            Tooltips.showTipListener);
        Core.addEventListener(links[i], "mouseout",
            Tooltips.hideTipListener);
        Core.addEventListener(links[i], "blur",
            Tooltips.hideTipListener);
      }
    }
  },
```

As you can see, this code uses `getElementsByTagName` to obtain a list of all the `a` elements in the page, and get the value of the `title` attribute for each one. If the `title` attribute exists, and has a value longer than zero characters (`if (title &&`

`title.length > 0`)), we set up event listeners for the four different events that we're interested in.

Although we've registered four event listeners on each of our links, you can see that there are actually only two event listener methods: `showTipListener`, which will display a tooltip in response to either a `mouseover` or `focus` event, and `hideTipListener`, which hides the tooltip in response to a `mouseout` or a `blur`.

Now, we could write the code that actually shows and hides tooltips directly inside these two methods, but I prefer to implement the "actions" in my scripts in separate methods, so that the code that controls *what* happens in response to an event is separate from the code that controls *how* it happens. Our event listener methods, therefore, tend to be relatively simple:

tooltips.js *(excerpt)*

```
showTipListener: function(event)
{
  Tooltips.showTip(this);
  Core.preventDefault(event);
},

hideTipListener: function(event)
{
  Tooltips.hideTip(this);
}
```

`showTipListener` calls `showTip`, the method that will actually display the tooltip, passing it a reference to the hyperlink that has been moused-over, or given keyboard focus. It then calls `preventDefault` to keep the browser from displaying a tooltip of its own in response to the event.

`hideTip method` is even simpler: it just calls `hideTip`, which will do the work of hiding the tooltip.

The Workhorse Methods

We've put it off as long as we can, but the time has come to write the code that will actually create our rich tooltips.

Until this point, every script we've written has either displayed a message box of some sort, or modified the style of an existing element in the page. To actually display something *new*, however—to dynamically add content to the page on the fly—is a very different trick.

There are two ways to modify the HTML content of a page using JavaScript:

- via the DOM API
- through the nonstandard `innerHTML` property

In this book, we'll only use the first option. As we learned in Chapter 3, the DOM API is a W3C standard that is likely to be supported in all web browsers for the foreseeable future, and its document modification features are up to almost any task. The `innerHTML` property, by contrast, is not described in any standard, and indeed browsers like Firefox have on occasion struggled to implement it consistently in combination with new standards like XHTML. That said, there is an argument to be made for `innerHTML`, and if you're curious you can read more about this alternative in The `innerHTML` Option, at the end of this section.

In Chapter 3, we concentrated on how to access and modify existing elements in an HTML document, but the DOM also lets us create and add new content to the page. Web developers who secretly wish they had the biceps of bricklayers call this **DOM building**.

To display a tooltip on the page, we'll add a `span` element that contains the text of the tooltip just inside the relevant link. Here's what the markup would look like if the tooltip were coded right into the document:

```
<a href="…">Link text<span class="tooltip">Tooltip text</span></a>
```

To create this element via the DOM, we'll use the `createElement` method of the `document` node:

tooltips.js *(excerpt)*

```
var tip = document.createElement("span");
```

Once we've created the `span`, we can set its `class` attribute:

tooltips.js *(excerpt)*

```
tip.className = "tooltip";
```

Next, we need to put the tooltip text inside the `span`. In the DOM, this will be a text node, which we can create with the `document` node's `createTextNode` method. We need to pass this method the text that we want the node to contain; we can grab it using the link's `title` property:

tooltips.js *(excerpt)*

```
var tipText = document.createTextNode(link.title);
```

To put our new text node inside our new `span`, we need to use the `span`'s `appendChild` method:

tooltips.js *(excerpt)*

```
tip.appendChild(tipText);
```

Every element in the DOM tree supports the `appendChild` method, which you can use to add any node as the last child of the element upon which you are calling the method. It's the DOM builder's best friend! To show you what I mean, we'll use it again—to add the tooltip to the document as a child of the link:

tooltips.js *(excerpt)*

```
link.appendChild(tip);
```

That's our DOM building done, and with just a few finishing touches, we have our `showTip` method:

tooltips.js *(excerpt)*

```
showTip: function(link)
{
  Tooltips.hideTip(link);
```

```
  var tip = document.createElement("span");
  tip.className = "tooltip";
  var tipText = document.createTextNode(link.title);
  tip.appendChild(tipText);
  link.appendChild(tip);

  link._tooltip = tip;
  link.title = "";

  // Fix for Safari2/Opera9 repaint issue
  document.documentElement.style.position = "relative";
},
```

Before building a new tooltip, this method calls `hideTip` to make sure that any existing tooltip has been removed, so that we don't end up with two (which might happen if the user hovered the cursor over a link that already had keyboard focus).

Once the new tooltip has been built and inserted, this method stores a reference to the tooltip as a property of the link named `_tooltip`.[11] This will make it easier for `hideTip` to remove the tooltip later, using only the reference to the link that it gets as an argument. Finally, the method sets the link's `title` property to an empty string, so the document doesn't contain the tooltip text twice. Cleanliness is next to godliness, they say!

Finally, both Safari 2 and Opera 9 have difficulty with some dynamically-inserted content like our tooltip, and won't refresh the page display fully. We can force these browsers to fully refresh the page display by changing the value of the CSS `position` property on the `html` element (`document.documentElement`).

That takes care of the creation and inserting of new DOM nodes, but to hide the tooltip you need to be able to *remove* content from the page. Predictably, the DOM provides a method to do this: `removeChild`. To see how it works, take a look at the code for `hideTip`:

[11] The underscore (_) at the start of this property name indicates that it's a "private" property—a property that isn't meant to be used by other scripts. It doesn't actually *prevent* other scripts from accessing it, but it's a clear indication to other developers that it's not a standard DOM property, which will make them think twice about using it.

tooltips.js *(excerpt)*

```
hideTip: function(link)
{
  if (link._tooltip)
  {
    link.title = link._tooltip.childNodes[0].nodeValue;
    link.removeChild(link._tooltip);
    link._tooltip = null;

    // Fix for Safari2/Opera9 repaint issue
    document.documentElement.style.position = "static";
  }
},
```

Before removing the tooltip, this method needs to check if there is actually a tooltip to remove. Since we stored a reference to the currently displayed tooltip in the link's `_tooltip` property, we just have to check if the property has a value.

With the certain knowledge that a tooltip is currently displayed, we need to retrieve from it the tooltip text and store it in the link's `title` property. You can get the text stored in a text node using its `nodeValue` property, and since the text node is the first child node of the tooltip element, we can access this as `link._tooltip.childNodes[0].nodeValue`. It's a little long-winded, but it works.

With the tooltip text safely tucked away, we can remove the tooltip using `removeChild`. Since the tooltip is a child of the link, we call `removeChild` on the link, and pass it a reference to the node that we want to remove from the document—the tooltip.

And last of all, to indicate that there is no longer a tooltip displayed for this link, we set its `_tooltip` property to `null`.

As with `showTip`, we need to cap this method off with the fix for the repainting bugs in Safari 2 and Opera 9. Since we set `position` to `relative` when showing the tooltip, we can just set it back to `static` when hiding the tooltip to force another repaint.

The `innerHTML` Option

Although it's not a part of any W3C standard, every major browser supports an `innerHTML` on every DOM element node. The value of this property is the HTML code of the content that it currently contains, and by changing that value, you can change the content of that element.

The biggest advantage offered by `innerHTML` is performance. If you're creating or modifying complex document structures, it can be a lot quicker for the browser to make document modifications in bulk using `innerHTML` than by stepping through a series of separate DOM modifications. In some cases, very complex JavaScript applications *must* use `innerHTML` to achieve reasonable performance.

Additionally, many developers quickly tire of writing the verbose JavaScript code that DOM manipulation requires, and the "one stop shop" that `innerHTML` offers is a tempting alternative. As a result, most of the major JavaScript libraries contain utilities for making DOM manipulation more convenient, and a number of mini-libraries like Dan Webb's DOM Builder have even sprung up to tackle this issue specifically.[12]

The Dynamic Styles

Having written all the JavaScript code required to add and remove tooltips on cue, all that's left for us to do is to write the CSS code that will make these ordinary `span`s really grab users' attention.

To begin with, we need to make sure our tooltips sit on top of the surrounding document content. Since our tooltips are generated inside hyperlinks, we can apply the necessary styles to our links. First, we set the positioning mode of all links to `relative`:

tooltips.css *(excerpt)*

```
a:link, a:visited {
  position: relative;
}
```

[12] http://www.webstandards.org/2006/04/13/dom-builder/

This alone does nothing to the appearance of our links, but it does enable us to modify the `z-index` property of these links when we need to—specifically, when the link is hovered or has keyboard focus:

tooltips.css *(excerpt)*

```
a:hover, a:focus, a:active {
  ⋮
  z-index: 1;
}
```

That takes care of displaying the tooltips on top of the surrounding elements. Now let's look at the tooltips themselves:

tooltips.css *(excerpt)*

```
/* Tooltips (dynamic styles) */

.tooltip {
  display: block;
  font-size: smaller;
  left: 0;
  padding: 5px;
  position: absolute;
  text-decoration: none;
  top: 1.7em;
  width: 15em;
}
```

Here's a breakdown of the various property declarations in this rule:

`text-decoration: none;`	removes the underline from the text that is inherited from the link in some browsers
`display: block;` **`width: 15em;`**	displays the tooltip as a block 15 ems wide
`position: absolute;` **`top: 1.7em;`** **`left: 0;`**	positions the box just below the link, in alignment with its left edge

`font-size: smaller;`	tweaks the appearance of the tooltip to more clearly
`padding: 5px;`	set it apart from the main document content

Those declarations look after the basic appearance of a tooltip. We can now play with details like text color, background, and border to get the effect we want. In this example, I've put together a couple of different tooltip styles, based on the class of the link to which the tooltip refers:

tooltips.css *(excerpt)*

```
.federation .tooltip {
  background: #COCOFF url(starfleet.png) top left no-repeat;
  color: #2E2E33;
  min-height: 54px;
  padding-left: 64px;
}

.klingon .tooltip {
  background: #BF0000 url(klingonempire.png) top left no-repeat;
  color: #FFF;
  min-height: 54px;
  padding-left: 64px;
}
```

Putting it All Together

That's all you need to produce great looking tooltips! Here's the complete JavaScript code for easy reference:

tooltips.js

```
var Tooltips =
{
  init: function()
  {
    var links = document.getElementsByTagName("a");

    for (var i = 0; i < links.length; i++)
    {
      var title = links[i].getAttribute("title");

      if (title && title.length > 0)
```

```
    {
      Core.addEventListener(
          links[i], "mouseover", Tooltips.showTipListener);
      Core.addEventListener(
          links[i], "focus", Tooltips.showTipListener);
      Core.addEventListener(
          links[i], "mouseout", Tooltips.hideTipListener);
      Core.addEventListener(
          links[i], "blur", Tooltips.hideTipListener);
    }
  }
},

showTip: function(link)
{
  Tooltips.hideTip(link);

  var tip = document.createElement("span");
  tip.className = "tooltip";
  var tipText = document.createTextNode(link.title);
  tip.appendChild(tipText);
  link.appendChild(tip);

  link._tooltip = tip;
  link.title = "";

  // Fix for Safari2/Opera9 repaint issue
  document.documentElement.style.position = "relative";
},

hideTip: function(link)
{
  if (link._tooltip)
  {
    link.title = link._tooltip.childNodes[0].nodeValue;
    link.removeChild(link._tooltip);
    link._tooltip = null;

  // Fix for Safari2/Opera9 repaint issue
  document.documentElement.style.position = "static";
  }
},

showTipListener: function(event)
```

```
  {
    Tooltips.showTip(this);
    Core.preventDefault(event);
  },

  hideTipListener: function(event)
  {
    Tooltips.hideTip(this);
  }
};

Core.start(Tooltips);
```

Example: Accordion

As shown in Figure 4.9, an accordion control collapses content to save space on the page, allowing the user to expand one "fold" of content at a time to read it.

This sort of interface enhancement is a great example of how JavaScript can improve the user experience on a page that works just fine without it. Making such enhancements work smoothly not just for mouse users, but for visitors who navigate using the keyboard (not to mention users of assistive technologies like screen readers), requires careful thought, and extensive use of event listeners.

The Static Page

As usual, we'll start by creating a static page with clean HTML and CSS code before we add any JavaScript. The accordion is essentially a collapsible list, so we'll use a `ul` element of class `accordion` to represent it:

accordion.html *(excerpt)*

```
<ul class="accordion">
  <li id="archer">
    <h2><a href="#archer">Jonathan Archer</a></h2>
    <p>Vessel registry: NX-01</p>
    <p>Assumed command: 2151</p>
    <div class="links">
      <h3>Profiles</h3>
      <ul>
```

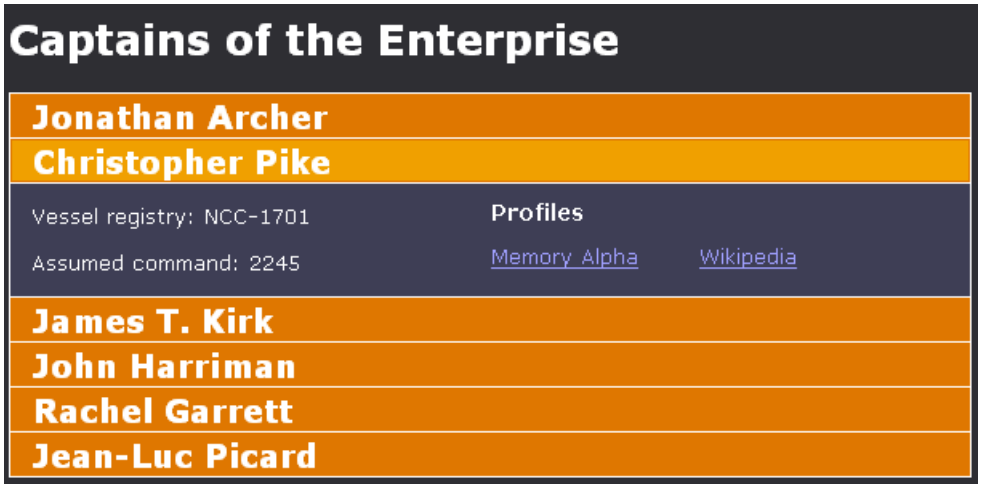


Figure 4.9. An accordion control

```
        <li><a href="…">Memory Alpha</a></li>
        <li><a href="…">Wikipedia</a></li>
      </ul>
    </div>
  </li>
  <li id="pike">
    ⋮
  </li>
  ⋮
</ul>
```

Note that we've applied an ID to each list item, and linked to that ID from the heading just inside the item. Although we're focused on creating a sensible HTML document at this stage, you can also look for opportunities to add meaningful structure that will help you to script the dynamic behaviour you want.

In this case, we can predict that we'll want users to be able to click on a heading in order to expand the corresponding fold of the accordion. Although we could implement this functionality by adding a `click` event listener directly to the heading, using a link makes it easier to support keyboard users. Links can be tabbed to and "clicked" using the browser's built-in keyboard support, whereas extra JavaScript code would be necessary to make a clickable heading accessible from the keyboard.

The next step is to write the CSS code that will style the static version of the page, so that it looks nice even in browsers where JavaScript is not available. Since this isn't a CSS book, I'll spare you the code and just show you the result in Figure 4.10. You can always check out the code in the code archive if you're curious.

Figure 4.10. The styled page as it will appear without JavaScript

The Workhorse Methods

With a perfectly good page in hand, we can now write a script to enhance it. We'll start with our usual script skeleton:

accordion.js *(excerpt)*

```
var Accordion =
{
  init: function()
  {
    ⋮
  },
  ⋮
};

Core.start(Accordion);
```

This script will be concerned primarily with expanding and collapsing the folds of our accordion list, so let's begin by writing the methods that will accomplish this task.

As we learned in Chapter 3, the best approach for modifying the appearance of elements on the fly is to assign CSS classes to them, and then define the actual appearance of each of those classes in our style sheet. So let's assign the expanded and collapsed folds the class names `expanded` and `collapsed`, respectively. With this decision made, writing a method that collapses a given fold couldn't be simpler:

accordion.js *(excerpt)*

```
collapse: function(fold)
{
  Core.removeClass(fold, "expanded");
  Core.addClass(fold, "collapsed");
},
```

Now feel free to go a different way on this point, but personally, I want to allow only one fold of the accordion to be expanded at a time. Expanding a fold, therefore, should also collapse all the other folds on the page. Because collapsing all the folds on the page is a useful thing to be able to do in any case, we'll write a separate method to accomplish this:

accordion.js *(excerpt)*

```
collapseAll: function(accordion)
{
  var folds = accordion.childNodes;
  for (var i = 0; i < folds.length; i++)
  {
    if (folds[i].nodeType == 1)
    {
      Accordion.collapse(folds[i]);
    }
  }
},
```

This method takes as an argument a reference to the accordion element you want to collapse completely, so logically the very first thing we want to do is get a list of the children of that element—the `li` elements of the list that represent the folds of the accordion.

This code uses a new trick in the DOM access department: the `if` statement that checks the `nodeType` property of each of those list items. We need to do this because there's no guarantee that all the child nodes of the list will be element nodes (that is, list items). Depending on the browser in which our accordion is being used,[13] the whitespace between the list items in our code may be represented as text nodes, which will be listed as children of the `ul` element as well. The `nodeType` property of a node tells you whether you're dealing with an element node, a text node, or an attribute node. Since we're only interested in elements (the list items in the list), we check for a `nodeType` value of `1`.

Other than this new trick, the method is fairly straightforward. It gets a list of all the folds in the accordion, then calls `Accordion.collapse` to collapse each one.

We now have what we need to write our `expand` method:

accordion.js *(excerpt)*

```
expand: function(fold)
{
  Accordion.collapseAll(fold.parentNode);
  Core.removeClass(fold, "collapsed");
  Core.addClass(fold, "expanded");
},
```

The Dynamic Styles

Now that we have the methods responsible for changing the appearance of our accordion, let's switch gears and write the CSS code that will control exactly what those appearance changes are.

[13] Safari is particularly prone to interpreting whitespace as a text node when you least expect it.

The appearance changes that we need for an accordion are really simple. With the exception of the heading of each fold, which should always remain visible, we need to hide the contents of a collapsed fold (an `li` element of class `collapsed`).

Now, we could naively achieve this by setting the `display` property to `none` or `block` where appropriate:

```
ul.accordion li.collapsed * {
  display: none;
}

ul.accordion li.collapsed h2, ul.accordion li.expanded h2,
ul.accordion li.collapsed h2 a:link,
ul.accordion li.collapsed h2 a:visited,
ul.accordion li.expanded h2 a:link,
ul.accordion li.expanded h2 a:visited {
  display: block;
}
```

… but doing it this way would effectively prevent users of screen readers from accessing the contents of our accordion. Screen readers don't read content that's hidden with `display: none`, even when it is later revealed. If there's another way to show and hide content dynamically in your JavaScript projects, you should use it.

In this case, there is definitely another way to hide the contents of the accordion. It's a technique called **offleft positioning**, which is just a fancy way of saying "hide stuff by positioning it off the left-hand side of the page:"

accordion.css *(excerpt)*

```
/* Accordion styles (dynamic) */

ul.accordion li.collapsed * {
  position: absolute;
  left: -10000px;
}

ul.accordion li.collapsed h2, ul.accordion li.expanded h2,
ul.accordion li.collapsed h2 a:link,
ul.accordion li.collapsed h2 a:visited,
ul.accordion li.expanded h2 a:link,
```

```
ul.accordion li.expanded h2 a:visited {
  position: static;
}
```

Offleft positioning is a great boon to screen reader users, because the screen reader software will read the hidden content just as if it were visible on the page.

The inconvenient side-effect of this approach is that the content also remains a part of the page for keyboard users, who will be forced to **Tab** through any links or form elements within that hidden content, even though the content isn't visible to them. In this particular example, however, we can make that work to our advantage, so offleft positioning is the best choice.

The work we've done so far takes care of the essentials, but for a little added spice, we can change the background color of the header of a fold that is either expanded, has the cursor over it, or has keyboard focus:

accordion.css *(excerpt)*

```
ul.accordion li.collapsed h2 a:hover,
ul.accordion li.collapsed h2 a:focus,
ul.accordion li.collapsed h2 a:active,
ul.accordion li.expanded h2 a:link,
ul.accordion li.expanded h2 a:visited {
  background-color: #F0A000;
}
```

Putting it All Together

Back in the world of JavaScript, we can now write the code that makes things happen, setting up the event listeners that will respond to user actions. Since we want to set up our event listeners as soon as the document has finished loading, we turn our attention to our script's `init` method.

Let's break down the tasks we want to achieve in this method:

1. Find the accordion list(s) in the page.
2. For each accordion, collapse each of the folds it contains.

3. When the user clicks on the link in the title of an accordion fold, expand it, or collapse it if it's already expanded.

That's not a bad start, so let's convert these plans into JavaScript code. If you're feeling confident, try doing this yourself before looking at the following code—you can implement each of those steps using only the techniques and features of JavaScript that we have already seen in this book:

accordion.js *(excerpt)*

```
init: function()
{
  var accordions = Core.getElementsByClass("accordion");

  for (var i = 0; i < accordions.length; i++)
  {
    var folds = accordions[i].childNodes;
    for (var j = 0; j < folds.length; j++)
    {
      if (folds[j].nodeType == 1)
      {
        Accordion.collapse(folds[j]);
        var foldLinks = folds[j].getElementsByTagName("a");
        var foldTitleLink = foldLinks[0];
        Core.addEventListener(foldTitleLink, "click",
            Accordion.clickListener);
```

This code gets a list of all elements with a class of `accordion` and uses a `for` loop to process them one at a time. For each accordion list, it retrieves the list of its child nodes (its folds) and again uses a `for` loop to step through them one at time.

After confirming that each node in the list is in fact an element node, it calls the `collapse` method to collapse the fold.

It then obtains a reference to the first link in the fold (which will be the one inside the fold's title), and adds to it an event listener that will respond to the user clicking on it. Here's that event listener:

accordion.js *(excerpt)*

```
clickListener: function(event)
{
  var fold = this.parentNode.parentNode;
  if (Core.hasClass(fold, "collapsed"))
  {
    Accordion.expand(fold);
  }
  else
  {
    Accordion.collapse(fold);
  }
  Core.preventDefault(event);
},
```

Again, things here are relatively straightforward. The listener obtains a reference to the fold, which is the parent node of the parent node of the link that has just been clicked. It then checks the fold's current CSS class to determine if the fold is collapsed or not. If it's collapsed, the code expands it. If it's expanded, the code collapses it.

The listener ends with a call to `Core.preventDefault`, which keeps the browser from attempting to follow the link. Although it wouldn't be disastrous if the browser *did* follow the link (after all, it just links to the fold itself), this could cause the page to scroll to the fold, when what we're after is a slick, seamless effect.

With the code we've seen so far, the accordion will work exactly the way we want it to for mouse and screen reader users, but we still have to solve the problem that offleft positioning causes for keyboard users. Remember, even though the contents of collapsed folds are hidden off the left of the screen, keyboard users will still find themselves **Tab**bing through the contents of those folds.

On the surface, this seems like a no-win situation, but let's try a strategy in use at most major software companies: what if we think of this as feature and not a bug?

If we can't keep keyboard users from **Tab**bing into hidden folds, why not make something *useful* happen when they do? Specifically, we can expand a fold when the user **Tab**s into it!

In an ideal world, an easy way to do this would be to add a `focus` event listener to each fold and catch the `focus` events as they bubbled up from the specific element within the fold that has received focus, but as we learned earlier in this chapter, `focus` events do not bubble.[14]

Instead, we need to attach a `focus` event listener to every element within our accordion that we expect might receive keyboard focus. For this example, we'll limit ourselves to hyperlinks, but if you were to add form elements to an accordion, you'd want to set them up with this listener as well:[15]

accordion.js *(excerpt)*

```
init: function()
{
  var accordions = Core.getElementsByClass("accordion");

  for (var i = 0; i < accordions.length; i++)
  {
    var folds = accordions[i].childNodes;
    for (var j = 0; j < folds.length; j++)
    {
      if (folds[j].nodeType == 1)
      {
        Accordion.collapse(folds[j]);
        var foldLinks = folds[j].getElementsByTagName("a");
        var foldTitleLink = foldLinks[0];
        Core.addEventListener(foldTitleLink, "click",
            Accordion.clickListener);

        for (var k = 1; k < foldLinks.length; k++)
        {
          Core.addEventListener(foldLinks[k], "focus",
              Accordion.focusListener);
        }
      }
    }
```

[14] If you want to get picky, they do bubble in Mozilla browsers like Firefox, but this behavior is in direct violation of the W3C standard for events, so I wouldn't be surprised to see it changed in future versions of these browsers.

[15] If you wanted to be especially thorough, you could add this listener to *every* element within your accordion, but the sheer number of listeners this might entail could put a real drag on the browser's performance, so I recommend the lighter—if riskier—approach we've used here.

Here's the `focus` event listener:

accordion.js *(excerpt)*

```
focusListener: function(event)
{
  var element = this;
  while (element.parentNode)
  {
    if (element.parentNode.className == "accordion")
    {
      Accordion.expand(element);
      return;
    }
    element = element.parentNode;
  }
}
```

This code showcases a common way of using the `parentNode` property to reach up through the DOM tree. Since all we know is that `this` is an element somewhere inside a fold of an accordion, you need to use a `while` loop to climb up through parent nodes of parent nodes until you find an element whose parent node has a class of `accordion`. That tells you that you've found the fold element, which you can promptly expand in the usual way.

As a finishing touch to our menu, let's add a bonus feature that takes advantage of the fact that, in the HTML code of the page, each of our folds has a unique ID:

accordion.html *(excerpt)*

```
<ul class="accordion">
  <li id="archer">
    ⋮
  </li>
  <li id="pike">
    ⋮
  </li>
  ⋮
</ul>
```

If a link on another page points to a specific fold of our accordion (e.g. `<a href="accordion.html#pike">`), it would be nice to automatically expand that fold when the page is loaded. All this effect takes is a brief addition to our `init` method:

accordion.js *(excerpt)*

```
init: function()
{
  var accordions = Core.getElementsByClass("accordion");

  for (var i = 0; i < accordions.length; i++)
  {
    ⋮

    if (location.hash.length > 1)
    {
      var activeFold = document.getElementById(
          location.hash.substring(1));
      if (activeFold && activeFold.parentNode == accordions[i])
      {
        Accordion.expand(activeFold);
      }
    }
  }
},
```

The global `location` variable is an object that contains information about the URL of the current page. Its `hash` property contains the fragment identifier portion of that URL (e.g. `"#pike"`), which we can use to identify and expand the requested fold.

First, we obtain a reference to the element with the specified ID. Since the ID we're after is the fragment identifier minus the leading # character, we can use the `substring` method that's built into every JavaScript string value to fetch a portion of the fragment identifier, starting at the second character (`location.hash.substring(1)`):

accordion.js *(excerpt)*

```
var activeFold = document.getElementById(
      location.hash.substring(1));
```

Before expanding the element, we need to check if that ID actually corresponds to an element in the page, and if the parent of that element is the accordion that we're currently setting up:

accordion.js *(excerpt)*

```
if (activeFold && activeFold.parentNode == accordions[i])
{
  Accordion.expand(activeFold);
}
```

And there you have it: a robust and accessible accordion control that you can easily plug into any web page you like. Load it up in your favorite browser and take it for a spin. Be sure to try using the keyboard to **Tab** through the accordion and its contents to see how a little extra code can go a long way toward making your site work well for a wider audience.

Here's the complete JavaScript code:

accordion.js

```
var Accordion =
{
  init: function()
  {
    var accordions = Core.getElementsByClass("accordion");

    for (var i = 0; i < accordions.length; i++)
    {
      var folds = accordions[i].childNodes;
      for (var j = 0; j < folds.length; j++)
      {
        if (folds[j].nodeType == 1)
        {
          Accordion.collapse(folds[j]);
          var foldLinks = folds[j].getElementsByTagName("a");
          var foldTitleLink = foldLinks[0];
          Core.addEventListener(
              foldTitleLink, "click", Accordion.clickListener);

          for (var k = 1; k < foldLinks.length; k++)
          {
```

```
          Core.addEventListener(
              foldLinks[k], "focus", Accordion.focusListener);
        }
      }
    }

    if (location.hash.length > 1)
    {
      var activeFold = document.getElementById(
          location.hash.substring(1));
      if (activeFold && activeFold.parentNode == accordions[i])
      {
        Accordion.expand(activeFold);
      }
    }
  }
},

collapse: function(fold)
{
  Core.removeClass(fold, "expanded");
  Core.addClass(fold, "collapsed");
},

collapseAll: function(accordion)
{
  var folds = accordion.childNodes;
  for (var i = 0; i < folds.length; i++)
  {
    if (folds[i].nodeType == 1)
    {
      Accordion.collapse(folds[i]);
    }
  }
},

expand: function(fold)
{
  Accordion.collapseAll(fold.parentNode);
  Core.removeClass(fold, "collapsed");
  Core.addClass(fold, "expanded");
},

clickListener: function(event)
```

```
  {
    var fold = this.parentNode.parentNode;
    if (Core.hasClass(fold, "collapsed"))
    {
      Accordion.expand(fold);
    }
    else
    {
      Accordion.collapse(fold);
    }
    Core.preventDefault(event);
  },

  focusListener: function(event)
  {
    var element = this;
    while (element.parentNode)
    {
      if (element.parentNode.className == "accordion")
      {
        Accordion.expand(element);
        return;
      }
      element = element.parentNode;
    }
  }
};

Core.start(Accordion);
```

Exploring Libraries

Most JavaScript libraries contain solutions to the browser compatibility issues surrounding event listeners that we tackled in this chapter. Although we believe the Core library that we have developed for this book to be well worth using, depending on your preferences, you may find these alternatives slightly more convenient, efficient, or confusing.

The Prototype library, for example, offers cross-browser alternatives for the standard `addEventListener` and `removeEventListener` methods, called `Event.observe` and `Event.stopObserving`, respectively:

```
Event.observe(element, "event", eventListener);
```

```
Event.stopObserving(element, "event", eventListener);
```

These methods work much like our own `Core.addEventListener` and `Core.removeEventListener` methods, in that they transparently call the correct methods to add and remove event listeners in the current browser, and ensure that the necessary cleanup is done to avoid listener-related memory leaks in Internet Explorer.

These methods leave a couple of issues unresolved, however—the inconsistency surrounding the value of `this` within the listener function, for example. Instead of automatically ensuring that `this` refers to the element to which the event listener was applied, Prototype lets you specify exactly which object you'd like `this` to refer to by adding a `bindAsEventListener` method to every function.

Therefore, the correct way to set up an event listener in Prototype is as follows:

```
Event.observe(element, "event",
    eventListener.bindAsEventListener(element));
```

I've assumed here that we want `this` to refer to the element to which the listener was added, but we can pass any object we like as an argument to `bindAsEventListener`, and it will be used as the value of `this` when the event listener function is called.

Calling `bindAsEventListener` also resolves the remaining cross-browser compatibility issues, for example, ensuring that the event object is passed to the listener function as an argument, even in Internet Explorer.

Unfortunately, this approach complicates the process of *removing* an event listener slightly. As we need to pass the same arguments to `Event.stopObserving` in order for it to work, we need to store the result of calling `bindAsEventListener`:

```
var myEventListener = eventListener.bindAsEventListener(element);
Event.observe(element, "event", myEventListener);
⋮
Event.stopObserving(element, "event", myEventListener);
```

If Prototype makes registering event listeners a little complicated, it simplifies the control of event propagation and default actions. Instead of separate `preventDefault` and `stopPropagation` methods, Prototype gives you a single method that does both:

```
Event.stop(event);
```

At the opposite end of the spectrum in terms of complexity is the jQuery library, which makes the code for managing event listeners extremely simple. It has extremely easy-to-use `bind` and `unbind` methods that take care of all the cross-browser compatibility headaches associated with adding and removing event listeners:

```
$("#id").bind("event", eventListener);
```

```
$("#id").unbind("event", eventListener);
```

jQuery also provides for each event type a convenient method that lets us set up an event listener for that event with even less code than the above. For example, the `click` method lets us set up a `click` event listener:

```
$("#id").click(clickListener);
```

Note that we can call any of these methods on a jQuery node list in order to add the listener to every node in the list. For example, we can add a `click` listener to every link in the document with just one statement:

```
$("a[href]").click(clickListener);
```

Nestled halfway between Prototype and jQuery in terms of complexity is the Yahoo! UI Library, which works almost exactly like our own `Core.addEventListener` and `Core.removeEventListener` methods.

Summary

You've taken a huge step in this chapter! Instead of just running JavaScript code as the page is loaded, painting a pretty picture for the user to gaze at in vague disappointment, you can now attach your code to events, following the user's lead like a practiced dance partner, adjusting and responding to his or her every whim.

But in some ways, this can still be quite limiting. After all, just because the user stops dancing doesn't mean your code should have to!

In Chapter 5, we'll finally set JavaScript free! We'll explore the secrets of animation, and learn how to produce effects that take place over a period of time, independent of the user's interaction with the page. We'll also learn how this freedom can be used to further improve the examples we created in this chapter.

Chapter 5

Animation

Are you ready for your journey into the … *fourth dimension!?* (Cue spooky theramin music.)

Animation is all about time—stopping time, starting time, and bending time to produce the effect you want: movement. You don't actually have to learn much new JavaScript in order to create web page animation—just two little functions. One and a half, really. What are they? You'll have to read on to find out!

But first, let's take a look at exactly what animation is, and how we can best go about producing it on a computer screen.

The Principles of Animation

If, like me, you're a connoisseur of Saturday morning cartoons, you'll probably be familiar with the basic principles of animation. As illustrated in Figure 5.1, what we see on the TV screen isn't actually Astroboy jetting around, but a series of images played rapidly, with minute changes between each one, producing the illusion that Astroboy can move.

Standard TV displays between 25 and 30 pictures (or **frames**) per second, but we don't perceive each picture individually—we see them as one continuous, moving image. Animation differs from live TV only in the way it is produced. The actual display techniques are identical—a series of rapidly displayed images that fools our brains into seeing continuous movement.

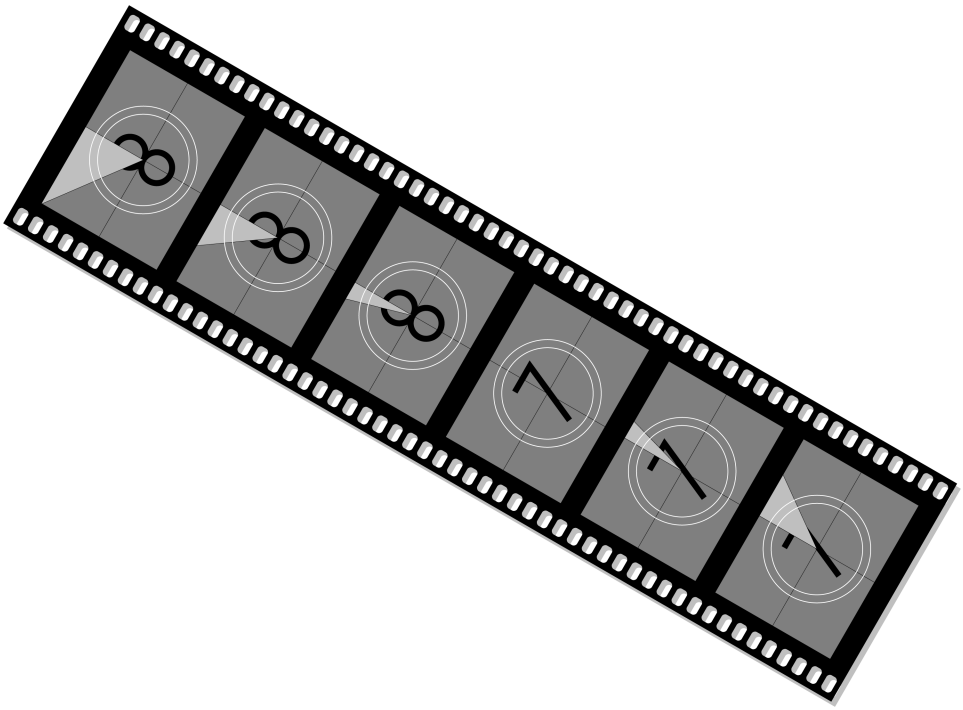


Figure 5.1. Creating the illusion of movement using a series of progressively changing images

Adding animation to your interfaces is no different from animating something for TV, although it may seem somewhat less artistic. If you want a drop-down menu to actually drop *down*, you start it closed, end it open, and include a whole bunch of intermediate steps that make it look like it moved.

The faster you want to move it, the fewer steps you put in between the starting and finishing frames. As it is, a drop-down menu without animation is akin to teleportation—the menu moves directly from point A to point B, without any steps in between. If you add just one step, it's not quite teleportation, but it's still faster than you can blink. Add ten steps and the menu begins to look like it's moving smoothly. Add 50 and it's probably behaving a bit too much like a tortoise. It's really up to

you to find the optimal intervals that give you that nice, pleasing feeling of the menu opening.

Controlling Time with JavaScript

The most intuitive way of thinking about animations is as slices of time. We're capturing and displaying discrete moments in time so that they look like one continuous experience. But, given the way we handle animation in JavaScript, it's probably more accurate to think of ourselves as inserting the *pauses*—putting gaps *between* the slices, as is illustrated in Figure 5.2.

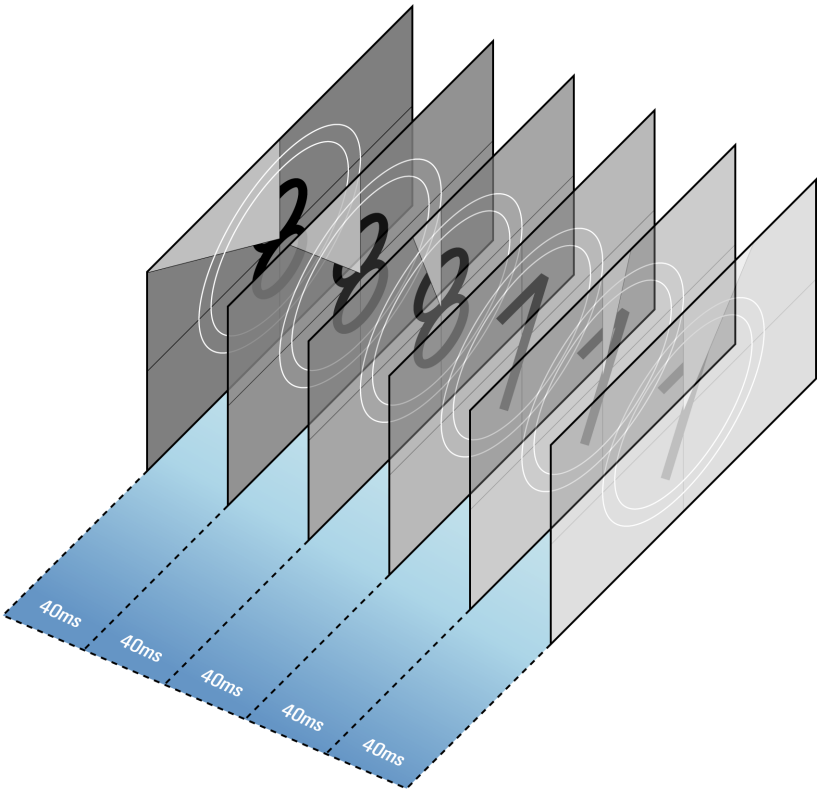


Figure 5.2. An animation running at 25 frames per second, in which the gap between each "slice" of time is 40 milliseconds

In normal program flow a browser will execute JavaScript as quickly as it can, pumping instructions to the CPU whenever they can be processed. This means that your entire, 5,000-line program could execute faster than you can take a breath. That's great if you're performing long calculations, but it's useless if you're creating

animation. Watching *Snow White and the Seven Dwarfs* in 7.3 seconds might be efficient, but it doesn't do much for the storyline.

To create smooth, believable animation we need to be able to control when each frame is displayed. In traditional animation, the **frame rate** (that is, the number of images displayed per second) is fixed. Decisions about how you slice the intervals, where you slice them, and what you slice, are all up to you, but they're displayed at the same, measured pace all the time.

In JavaScript, you can create the same slices and control the same aspects, but you're also given control of the pauses—how long it takes for one image to be displayed after another. You can vary the pauses however you like, slowing time down, speeding it up, or stopping it entirely (though most of the time you'll want to keep it steady). We control time in JavaScript using `setTimeout`.

Instead of allowing your program to execute each statement as quickly as it can, `setTimeout` tells your program to wait awhile—to take a breather. The function takes two arguments: the statement you want it to run next, and the time that the program should wait before executing that statement (in milliseconds).

In order to stop the statement inside the `setTimeout` call from being evaluated immediately, we must make it into a string. So if we wanted to display an alert after one second, the `setTimeout` call would look like this:

```
setTimeout("alert('Was it worth the wait?')", 1000);
```

From the moment this statement is executed, our program will wait 1000 milliseconds (one second) before it executes the code inside the string, and produces the popup shown in Figure 5.3.

Technically speaking, what happens when `setTimeout` is called is that the browser adds the statement to a "to-do list" of sorts, while the rest of your program continues to run without pausing. Once it finishes what it's currently doing (for instance, executing an event listener), the browser consults its to-do list and executes any tasks it finds scheduled there.

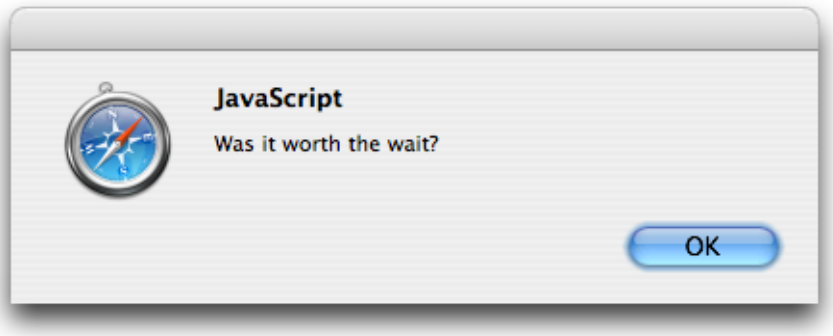


Figure 5.3. Using `setTimeout` to execute code after a specified number of milliseconds have elapsed

Usually, when you call a function from inside another function, the execution of the outer function halts until the inner function has finished executing. So a function like the one shown here actually executes as depicted in Figure 5.4:

```
function fabric()
{
  ⋮
  alert("Stop that man!");
  ⋮
}
```

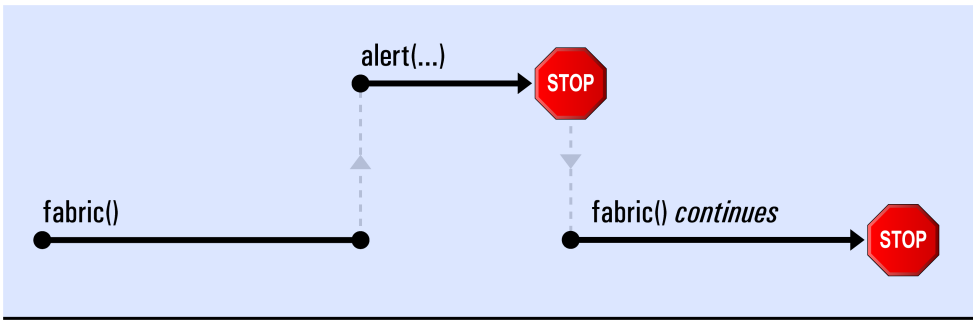


Figure 5.4. Calling a function inside another function

But what happens if you have a `setTimeout` call in the middle of your code?

```
function fabric()
{
  ⋮
```

```
  setTimeout("alert('Was it worth the wait?');", 1000);
  ⋮
}
```

In this case, the execution of the statement passed to `setTimeout` is deferred, while the rest of `fabric` executes immediately. The scheduled task waits until the specified time has elapsed, *and* the browser has finished what it's doing, before executing. Figure 5.5 illustrates this concept.

Strictly speaking, the delay that you specify when calling `setTimeout` will be the *minimum* amount of time that must elapse before the scheduled task will occur. If the browser is still tied up doing other things when that time arrives, your task will be put off until all others have been completed, as Figure 5.6 indicates.[1]

Using Variables with `setTimeout`

Statements passed to `setTimeout` are run in global scope. We talked about scope in Chapter 2, so if you'd like to refresh your memory, flick back to that chapter. The effect that scope has here is that the code run by `setTimeout` will not have access to any variables that are local to the function from which `setTimeout` was called.

Take a look at this example:

```
function periscope()
{
  var message = "Prepare to surface!";

  setTimeout("alert(message);", 2000);
}
```

Here, the variable `message` is local to `periscope`, but the code passed to the `setTimeout` will be run with global scope, entirely separately from `periscope`. Therefore, it won't have access to `message` when it runs, and we'll receive the error shown in Figure 5.7.

[1] If you specify a delay of zero milliseconds when you call `setTimeout`, the browser will execute the task as soon as it's finished what it's currently doing.

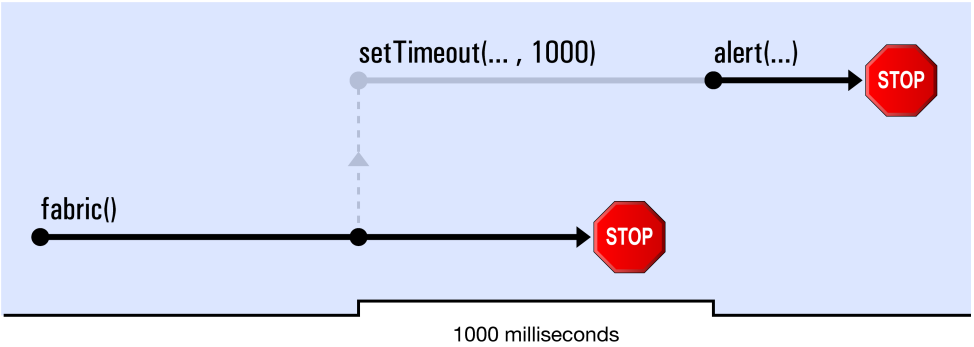


Figure 5.5. Calling `setTimeout` schedules a task to be run after a delay

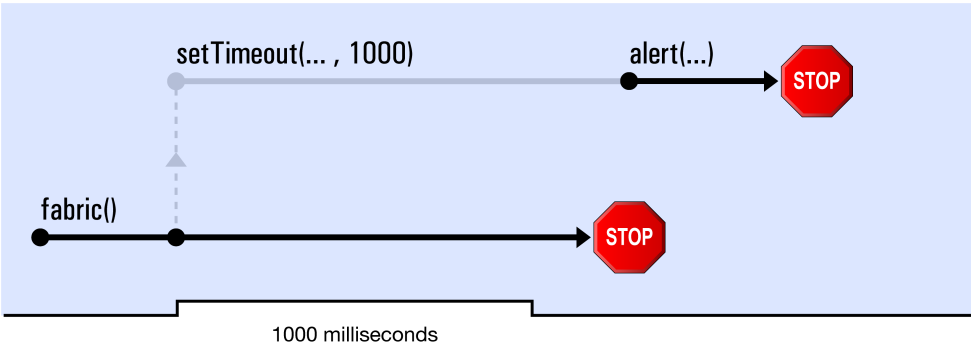


Figure 5.6. `setTimeout` waits until JavaScript is free before running the newly specified task

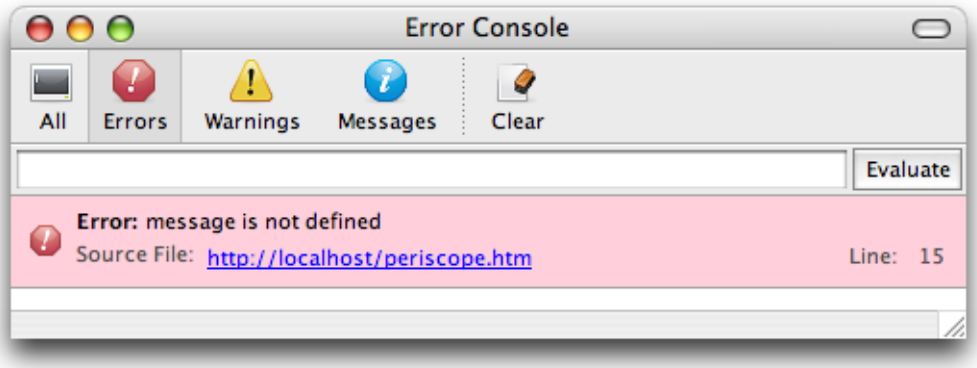


Figure 5.7. The error that occurs when we try to use a locally scoped variable as part of `setTimeout` code

There are three ways around this problem.

The first is to make `message` a global variable. This isn't that great an idea, because it's not good practice to have variables floating around, cluttering up the global namespace, and causing possible clashes with other scripts. But if we *were* to go down this path, we could just leave the `var` keyword off our variable declaration:

```
function periscope()
{
  message = "Prepare to surface!";

  setTimeout("alert(message)", 2000);
}
```

`message` would be a global variable that's accessible from any function, including the code passed to `setTimeout`.

The second option is available when the variable we're using is a string. In this case, we can encode the value of the variable directly into the `setTimeout` code simply by concatenating the variable into the string:

```
function periscope()
{
  message = "Prepare to surface!";

  setTimeout("alert('" + message + "')", 2000);
}
```

Since we want the string to be interpreted as a string—not a variable name—we have to include single quote marks inside the `setTimeout` code string, and in between those marks, insert the value of the variable. Though the above code is equivalent to the code below, we have the advantage of being able to use dynamically assigned text with the variable:

```
setTimeout("alert('Prepare to surface!')", 2000);
```

Troublesome Quotes

Of course, if the variable's string value happens to contain one or more single quotes, this approach will fall to pieces unless you go to the trouble of escaping

each of the single quotes with a backslash. And while that is certainly possible, the code involved isn't much fun.

The final option, and the one that's used more regularly in complex scripts, is to use a **closure**. A closure is a tricky JavaScript concept that allows any function that's defined inside another function to access the outer function's local variables, regardless of when it's run. Put another way, functions have access to the context in which they're defined, even if that context is a function that's no longer executing. So, even though the outer function may have finished executing minutes ago, functions that were declared while it was executing will still be able to access its local variables.

How does this help us given the fact that `setTimeout` takes a string as its first argument? Well, it can also take a function:

```
function periscope()
{
  var message = "Prepare to surface!";

  var theFunction = function()
  {
    alert(message);
  };

  setTimeout(theFunction, 2000);
}
```

Or, more briefly:

```
function periscope()
{
  var message = "Prepare to surface!";

  setTimeout(function(){alert(message);}, 2000);
}
```

When you pass a function to `setTimeout`, `setTimeout` doesn't execute it then and there; it executes the function after the specified delay. Since that function was declared within `periscope`, the function creates a closure, giving it access to `periscope`'s local variables (in this case, `message`).

Even though `periscope` will have finished executing when the inner function is run in two seconds' time, `message` will still be available, and the call to `alert` will bring up the right message.

The concept of closures is difficult to get your head around, so for the moment, you can be satisfied if you've just got a general feel for them.

Stopping the Timer

When `setTimeout` is executed, it creates a timer that "counts down" to the moment when the specified code should be executed. The `setTimeout` function returns the ID of this timer so that you can access it later in your script. So far, we've been calling `setTimeout` without bothering about its return value, but if you ever want to intervene in a timer's countdown, you have to pay attention to this value.

To stop a timer before its countdown has finished, we need to capture the timer's ID inside a variable and pass it to a function called `clearTimeout`. `clearTimeout` will immediately cancel the countdown of the associated timer, and the scheduled task will never occur:

```
var timer = setTimeout("alert('This will never appear')", 3000);
clearTimeout(timer);
```

The alert in the code above will never be displayed, because we stop the `setTimeout` call immediately using `clearTimeout`.

Let's consider something that's a little more useful. Suppose we create a page that displays two buttons:

clear_timeout.html *(excerpt)*

```
<button id="start">Start</button>
<button id="stop">Stop</button>
```

We can add some behavior to those buttons with a short program:

clear_timeout.js

```
var ClearTimer =
{
  init: function()
  {
    var start = document.getElementById("start");
    Core.addEventListener(start, "click", ClearTimer.clickStart);

    var stop = document.getElementById("stop");
    Core.addEventListener(stop, "click", ClearTimer.clickStop);
  },
  clickStart: function()
  {
    ClearTimer.timer = setTimeout("alert('Launched')", 2000);
  },
  clickStop: function()
  {
    if (ClearTimer.timer)
    {
      clearTimeout(ClearTimer.timer);
    }

    alert("Aborted");
  }
};

Core.start(ClearTimer);
```

This program begins by running `ClearTimer.init` via `Core.start`, which initializes the page by adding a `click` event listener to each of the buttons. When the **Start** button is clicked, `ClearTimer.clickStart` will be called, and when the **Stop** button is clicked, `ClearTimer.clickStop` will be called.

`ClearTimer.clickStart` uses `setTimeout` to schedule an alert, but we store the ID to that scheduled alert inside the variable `ClearTimer.timer`. Whenever `ClearTimer.clickStop` is pressed we pass `ClearTimer.timer` to `clearTimeout`, and the timer will be stopped.

So, once you click **Start** you will have two seconds to click **Stop**; otherwise, the alert dialog will appear, displaying the message "Launched." This simple example

provides a good illustration of what's involved in controlling timers via user interaction; we'll look at how it can be used in a more complex interface later in this chapter.

Creating a Repeating Timer

There's another timing function that's very similar to `setTimeout`, but which allows you to schedule a repeating piece of code. This function is called `setInterval`.

`setInterval` takes exactly the same arguments as `setTimeout`, and when it's executed, it waits in exactly the same way. But the fun doesn't stop once the scheduled code has finished executing—`setInterval` schedules the code again, and again, and again, and again, until you tell it to stop. If you tell it to wait 1,000 milliseconds, `setInterval` will schedule the code to run once every 1,000 milliseconds.

Each interval is scheduled as a separate task, so if a particular occurrence of the scheduled code takes longer to run than the interval time, the next occurrence will execute immediately after it finishes, as the browser scrambles to catch up.

At the start of this chapter I mentioned that there were really one and a half functions that could be used to control time in JavaScript. That's because `setInterval` isn't used very often.

As `setInterval` relentlessly schedules tasks at the specified interval no matter how long those tasks actually take, long-running tasks can generate an undesirable backlog that causes fast-running tasks to run instantly, with no pause between them. In an animation, this can be disastrous, as the pause between each frame is crucial to generating that illusion of motion.

When it comes to animation, it's usually best to rely on chained `setTimeout` calls, including at the end of your scheduled code a `setTimeout` call to that same piece of code. This approach ensures that you only have one task scheduled at any one time, and that the crucial pauses between frames are maintained throughout the animation. You'll see examples of this technique in each of the animation programs we write in this chapter.

Stopping `setInterval`

Just like `setTimeout`, `setInterval` returns a timer ID. To stop `setInterval` from executing any more scheduled code, we pass this ID to `clearInterval`:

```
var timer = setInterval("alert('Do this over and over!')", 3000);
clearInterval(timer);
```

Done! No more timer.

Revisiting Rich Tooltips

It's probably easiest to get a feel for working with `setTimeout` if we start simply—using it to put in a short delay on an action. I'll demonstrate this by modifying the Rich Tooltips script from Chapter 4, so that the tooltips behave more like regular tooltips and appear after the user has hovered the mouse over the element for a moment. We won't need to change much of the program—just the two event listeners that handle the `mouseover`/`focus` and `mouseout`/`blur` events.

Let's think about how the delay should work. We want the tooltip to pop up about 500 milliseconds after the `mouseover` or `focus` event occurs. In our original script, the event listener `showTipListener` immediately made a call to `Tooltips.showTip`. But in the delayed script, we want to move that call into a `setTimeout`:

tooltips_delay.js *(excerpt)*

```
showTipListener: function(event)
{
  var link = this;
  this._timer = setTimeout(function()
      {
        Tooltips.showTip(link);
      }, 500);
  Core.preventDefault(event);
},
```

In order to pass `showTip` a reference to the link in question, we need to store that reference in a local variable called `link`. We can then make use of a closure by passing `setTimeout` a newly created function. That new function will have access

to `showTipListener`'s local variables, and when it's run, it will be able to pass the link to `showTip`.

The other point to note about this `setTimeout` call is that we assign the ID it returns to a custom property of the anchor element, called `_timer`. We store this value so that if a user mouses off the anchor before the tooltip has appeared, we can stop the `showTip` call from going ahead. To cancel the method call, we update `hideTipListener` with a `clearTimeout` call:

tooltip_delay.js *(excerpt)*

```
hideTipListener: function(event)
{
  clearTimeout(this._timer);
  Tooltips.hideTip(this);
}
```

The ID of the `setTimeout` call is passed to `clearTimeout` every time that `hideTipListener` is executed, and this will prevent premature "tooltipification."

And that's it: our Rich Tooltips script has now been modified into a moderately less annoying Rich *Delayed* Tooltips script.

Old-school Animation in a New-school Style

To demonstrate how to create a multi-stepped animation sequence in JavaScript we're going to do something novel, though you probably wouldn't implement it on a real web page. We're going to re-create a film reel in HTML.

First of all, we need a strip of film—or a graphic that looks almost like a real piece of film. As you can see in Figure 5.8, our reel contains a series of progressively changing images that are joined together in one long strip.

If you had a real reel, you'd pass it through a projector, and the frames would be projected individually onto the screen, one after the other. For this virtual film reel, our projector will be an empty `div`:

robot_animation.html *(excerpt)*

```
<div id="robot"></div>
```

The `div` will be styled to the exact dimensions of a frame (150x150px), so that we can show exactly one frame inside it:

robot_animation.css *(excerpt)*

```
#robot {
  width: 150px;
  height: 150px;
```

If we use the strip of robots as a background image, we can change the `background-position` CSS property to move the image around and display different parts of the strip:

robot_animation.css *(excerpt)*

```
#robot {
  width: 150px;
  height: 150px;
  background-image:
    url(../images/robot_strip.gif);
  background-repeat: no-repeat;
  background-position: 0 0;
}
```

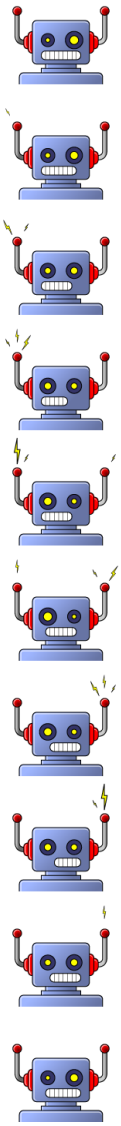


Figure 5.8. The "reel" we'll use to create animation

Getting the idea now? You can think of the `div` as a little window inside which we're moving the strip around, as Figure 5.9 illustrates.

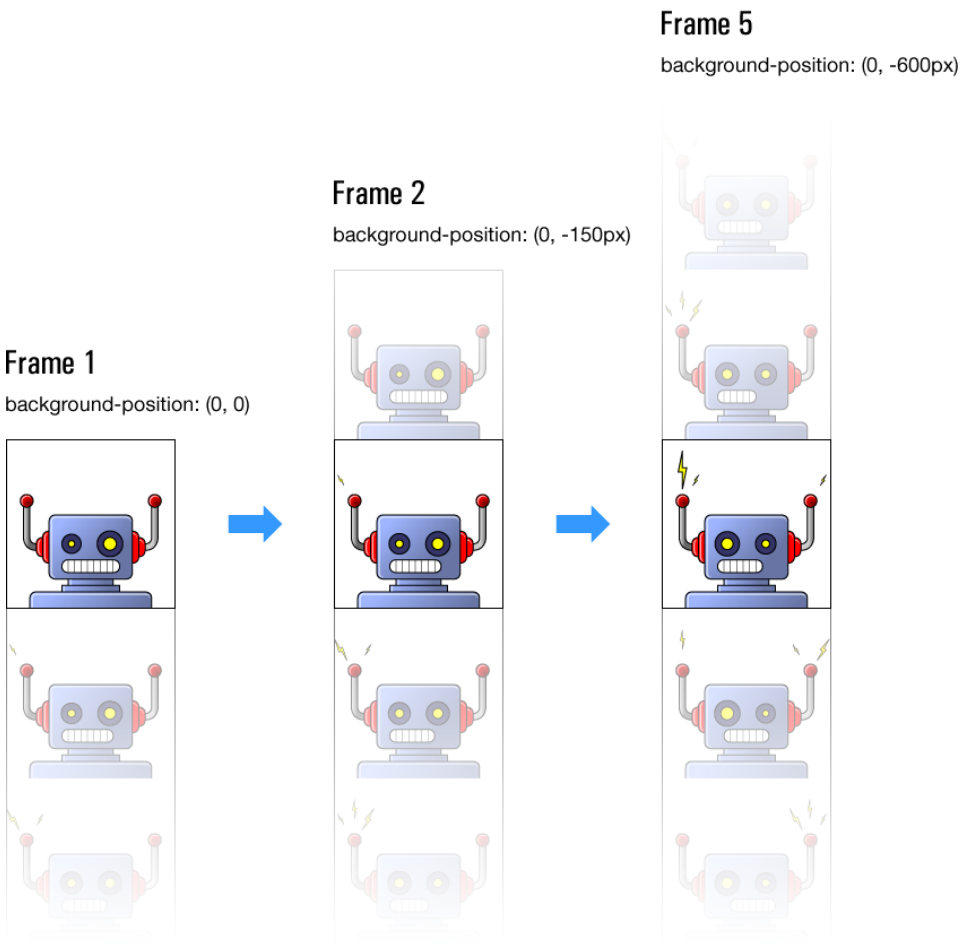


Figure 5.9. Changing the position of the background image to specify which frame is in view

So, we know how to make any one frame appear inside the `div` window, but the page is still static—there's no animation. That's what that ol' JavaScript magic is for!

In order to create a fluid animation, we're going to need to change the `background-position` at a regular interval, flicking through the frames so that they look like one fluid, moving image. With all the talk of `setTimeout` in this chapter,

you could probably take a guess that the function is the key to it all. And you'd be wrong. Sorry, *right*:

robot_animation.js

```
var Robot =
{
  init: function()
  {
    Robot.div = document.getElementById("robot"); (1)
    Robot.frameHeight = 150;
    Robot.frames = 10;
    Robot.offsetY = 0; (2)

    Robot.animate(); (3)
  },
  animate: function() (4)
  {
    Robot.offsetY -= Robot.frameHeight; (5)

    if (Robot.offsetY <= -Robot.frameHeight * Robot.frames) (6)
    {
      Robot.offsetY = 0;
    }

    Robot.div.style.backgroundPosition =
        "0 " + Robot.offsetY + "px"; (7)

    setTimeout(Robot.animate, 75); (8)
  }
};

Core.start(Robot);
```

The `Robot` object contains two methods. `Robot.init` is included mainly to declare some variables and kick-start the animation. I include the variables here, rather than in `Robot.animate`, because `Robot.animate` will be called a lot, and it's more efficient to declare the variables just once than to have them declared for each frame of our animation.

Here are the highlights of this script:

1. The initialized variables include a reference to the `div` element we're modifying, as well as two constant properties— `frameHeight` and `frames`—which tell the program that each frame is 150 pixels high, and that there are ten frames in total. These values will remain fixed throughout an animation, but by declaring them up front, you'll find it easier to modify the frame dimensions and the number of frames later if you need to. If you create your own animation, you should modify these variables according to your needs.

2. The other variable, `Robot.offsetY`, is the only variable we'll be updating dynamically. It's set to `0` initially, but it's used to record the current vertical position of the animation image, just so we know which frame we're up to.

3. After the `init` method does its thing, we step into our first iteration of `Robot.animate`.

4. If you look to the end of the function, you'll see that the function calls itself with a `setTimeout` delay. This lets you know that we're setting up a timed, repeating task.

5. The first statement in the function reduces `Robot.offsetY` by the height of one frame. What we're calculating here is the vertical `background-position` required to view the next frame. However, our animation graphic isn't infinitely long; if we just kept reducing the value of `Robot.offsetY`, we'd eventually run out of frames and end up with an empty box. To avoid this, we need to return to the beginning of the strip at the right moment.

6. Next up is an `if` statement that checks whether the new `Robot.offsetY` value is outside the range of our animation strip. If it is, we set it back to `0`.

7. With our new offset value in hand, we're ready to update the position of the image. To do so, we modify the `div`'s `style.backgroundPosition` property, using `Robot.offsetY` for the vertical coordinate and `0` for the horizontal coordinate (because we don't need to move the image horizontally at all).

Units Required

Remember to include units at the end of all non-zero CSS values, otherwise your changes will be ignored. We're using pixels in this case, so we add `px` to the end of the value.

8. Changing the style of the `div` commits the new frame to the browser for the user to see. Once that's done, we have to schedule the next change using `setTimeout`. The wait until the next frame change is 75 milliseconds, which will produce a frame rate of approximately 13 frames per second. You might want to tweak this rate depending on how fast or smooth you want your animation to be. The most pleasing results are often reached by trial and error.

Once the program has made a few passes through `Robot.animate`, you have yourself an animation. Since there's nothing in our code to stop the repeated calls to that method, the animation will keep going forever. If you wanted to, you could provide start and stop buttons as we did previously in this chapter to allow users to stop and restart the animation. Alternatively, you could limit the number of cycles that the animation goes through by including a counter variable that's incremented each time `Robot.offsetY` returns to `0`.

Path-based Motion

The JavaScript techniques we used to animate our digital film reel can be applied whenever you want to make a series of progressive changes to the display over time. All you have to do is replace `background-position` with the attribute you want to modify: color, width, height, position, opacity—anything you can think of.

In our next example, we're going to look at moving an HTML element along a linear path. So, instead of changing the `background-position` of the target element, we change its actual position.

In cases where you want to move an element, you usually know where it is now (that is, its starting location), and where you want it to be (its end location), as Figure 5.10 shows.

Figure 5.10. Visualizing a path from the object's starting location to the point where you want it to end up

Once you've defined the two end points of the path, it's the job of the animation program to figure out all the steps that must occur to let the animated element move smoothly from point A to point B, as shown in Figure 5.11.

Figure 5.11. Calculating the steps required to move an element from point A to point B

Given that the movement is going to be automated by an animation function, all we really have to identify at the moment is:

- the element we want to move
- where we want to move it to
- how long we want it to take to reach its destination

We'll use the soccer ball and grassy background as the basis for a working example. The HTML for this document is fairly simple:

path-based_motion.html *(excerpt)*

```
<div id="grass">
  <div id="soccerBall"></div>
</div>
```

There are a number of ways we could position the soccer ball, but for this example I've chosen to use absolute positioning:

path-based_motion.css *(excerpt)*

```
#soccerBall {
  background-image: url(soccer_ball.png);
  background-repeat: no-repeat;
  height: 125px;
  left: 0;
  margin-top: 25px;
  position: absolute;
  top: 75px;
  width: 125px;
}
```

Positioning and Animation

If you're going to animate an element's movement, the element will need to be positioned relatively or absolutely; otherwise, changing its `left` and `top` properties won't have any effect.

The JavaScript we used to animate our previous film reel example makes a fairly good template for any animation, so we'll use its structure to help us define the movement we need in this new animation. Inside `init`, we declare some of the variables we'll need, then start the actual animation:

path-based_motion.js *(excerpt)*

```
var SoccerBall =
{
  init: function()
  {
    SoccerBall.frameRate = 25; (1)
    SoccerBall.duration = 2;
    SoccerBall.div = document.getElementById("soccerBall"); (2)
    SoccerBall.targetX = 600; (3)
    SoccerBall.originX = parseInt(
        Core.getComputedStyle(SoccerBall.div, "left"), 10); (4)
    SoccerBall.increment =
        (SoccerBall.targetX - SoccerBall.originX) /
        (SoccerBall.duration * SoccerBall.frameRate); (5)
    SoccerBall.x = SoccerBall.originX; (6)

    SoccerBall.animate(); (7)
  },
  ⋮
};
```

(1) The first two variables control the speed of the animation. `SoccerBall.frameRate` specifies the number of frames per second at which we want the animation to move. This property is used when we set the delay time for the `setTimeout` call, and determines the "smoothness" of the soccer ball's movement. `SoccerBall.duration` determines how long the animation should take to complete (in seconds) and affects the speed with which the ball appears to move.

(2) `SoccerBall.div` is self-explanatory.

(3) `SoccerBall.targetX` specifies the location to which we're moving the soccer ball. Once it reaches this point, the animation should stop.

(4) In order to support an arbitrary starting position for the soccer ball, `SoccerBall.originX` is actually calculated from the browser's computed style for the `div`.

The **computed style** of an element is its style information *after* all CSS rules and inline styles have been applied to it. So, if an element's `left` position has been

specified inside an external style sheet, obtaining the element's computed style will let you access that property value.

Unfortunately, there are differences between the way that Internet Explorer implements computed style and the way that other browsers implement it. Internet Explorer exposes a `currentStyle` property on every element. This properties has exactly the same properties as `style`, so you could ascertain an element's computed `left` position using `element.currentStyle.left`. Other browsers require you to retrieve a computed style object using the method `document.defaultView.getComputedStyle`. This method takes two arguments—the first is the element you require the styles for; the second must always be `null`—then returns a style object that has the same structure as Internet Explorer's `currentStyle` property.

To get around these browser differences, we'll create a new Core library function that will allow us to get a particular computed style property from an element:

core.js *(excerpt)*

```
Core.getComputedStyle = function(element, styleProperty)
{
  var computedStyle = null;

  if (typeof element.currentStyle != "undefined")
  {
    computedStyle = element.currentStyle;
  }
  else
  {
    computedStyle =
        document.defaultView.getComputedStyle(element, null);
  }

  return computedStyle[styleProperty];
};
```

`Core.getComputedStyle` does a little object detection to check which way we should retrieve the computed style, then passes back the value for the appropriate property. Using method, we can get the correct starting point for `SoccerBall.originX` without requiring a hard-coded value in our script.

Let's return to our `init` method above:

5. With `SoccerBall.originX` and `SoccerBall.targetX` in hand, we can calculate the increment by which we'll need to move the soccer ball in each frame so that it ends up at the target location within the duration specified for the animation. This is a simple matter of finding the distance to be traveled (`SoccerBall.targetX – SoccerBall.originX`) and dividing it by the total number of frames in the animation (`SoccerBall.duration * SoccerBall.frameRate`). Calculating this figure during initialization—rather than inside the actual animating function—reduces the number of calculations that we'll have to complete for each step of the animation.

6. The last variable we declare is `SoccerBall.x`. It acts similarly to `Robot.offsetY`, keeping track of the horizontal position of the element. It would be possible to keep track of the soccer ball's position using its actual `style.left` value, however, that value has to be an integer, whereas `SoccerBall.x` can be a floating point number. This approach produces more accurate calculations and smoother animation.

7. After `SoccerBall.init` has finished declaring all the object properties, we start the animation by calling `SoccerBall.animate`.

Here's the code for this method:

path-based_motion.js *(excerpt)*

```
animate: function()
{
  SoccerBall.x += SoccerBall.increment;

  if ((SoccerBall.targetX > SoccerBall.originX &&
      SoccerBall.x > SoccerBall.targetX) ||
      (SoccerBall.targetX <= SoccerBall.originX &&
      SoccerBall.x <= SoccerBall.targetX))
  {
    SoccerBall.x = SoccerBall.targetX;
  }
  else
  {
    setTimeout(SoccerBall.animate, 1000 / SoccerBall.frameRate)
  }
```

```
  SoccerBall.div.style.left = Math.round(SoccerBall.x) + "px";
}
```

The similarities between this `animate` method and `Robot.animate`, which we saw in the previous example, are striking. The process for both is basically:

1. Calculate the new position.
2. Check whether the new position exceeds the limit.
3. Apply the new position to the element.
4. Repeat the process with a delay.

In this case, we're calculating a new position by adding to the current position one "slice" of the total distance to be traveled:

path-based_motion.js *(excerpt)*

```
SoccerBall.x += SoccerBall.increment;
```

Then, we check whether that new position goes beyond the end point of the animation:

path-based_motion.js *(excerpt)*

```
if ((SoccerBall.targetX > SoccerBall.originX &&
    SoccerBall.x >= SoccerBall.targetX) ||
    (SoccerBall.targetX < SoccerBall.originX &&
    SoccerBall.x <= SoccerBall.targetX))
```

That condition looks a little daunting, but we can break it down into smaller chunks. There are actually two possible states represented here, separated by an OR operator.

The first of these states (`SoccerBall.targetX > SoccerBall.originX && SoccerBall.x >= SoccerBall.targetX`) checks whether `SoccerBall.targetX` is greater than `SoccerBall.originX`. If it is, we know that the soccer ball is moving

to the right. If that's the case, the soccer ball will be beyond the end point if `SoccerBall.x` is greater than `SoccerBall.targetX`.

In the second state (`SoccerBall.targetX < SoccerBall.originX && SoccerBall.x <= SoccerBall.targetX`), if `SoccerBall.targetX` is *less than* `SoccerBall.originX`, the soccer ball will be moving to the *left*, and it will be beyond the end point if `SoccerBall.x` is *less than* `SoccerBall.targetX`. By including these two separate states inside the condition, we allow the soccer ball to move in any direction without having to modify the code.

If the newly calculated position for the soccer ball exceeds the end point, we automatically set the new position to be the end point:

path-based_motion.js *(excerpt)*

```
if ((SoccerBall.targetX > SoccerBall.originX &&
    SoccerBall.x >= SoccerBall.targetX) ||
    (SoccerBall.targetX < SoccerBall.originX &&
    SoccerBall.x <= SoccerBall.targetX))
{
  SoccerBall.x = SoccerBall.targetX;
}
```

Otherwise, the soccer ball needs to keep moving, so we schedule another animation frame:

path-based_motion.js *(excerpt)*

```
else
{
  setTimeout(SoccerBall.animate, 1000 / SoccerBall.frameRate)
}
```

Notice that the delay for the `setTimeout` is specified as 1000 milliseconds (one second) divided by the specified frame rate. This calculation transforms the frame rate into the millisecond format that's required for the delay.

The last thing we need to do for each frame is apply the newly calculated position to the `style.left` property of the soccer ball. This step causes the browser to display the update:

path-based_motion.js *(excerpt)*

```
SoccerBall.div.style.left = Math.round(SoccerBall.x) + "px";
```

That statement converts `SoccerBall.x` to an integer using `Math.round`, which rounds any number up or down to the nearest integer. We need to do this because `SoccerBall.x` might be a floating point number, and CSS pixel values can't be decimals.

The last two statements (the `setTimeout` and the `style` change) would normally be written in the reverse of the order shown here, but the `setTimeout` is placed inside the `else` statement for code efficiency. If we were to place it after the `style` assignment, `setTimeout` would require an additional conditional check. By doing it this way, we can avoid adversely affecting the performance of our script.

Once we assemble both the `init` and `animate` functions inside the one object, which we initialize using `Core.start`, we have our finished program. We're all set to roll that soccer ball over a lush, green field:

path-based_motion.js

```
var SoccerBall =
{
  init: function()
  {
    SoccerBall.frameRate = 25;
    SoccerBall.duration = 2;
    SoccerBall.div = document.getElementById("soccerBall");
    SoccerBall.targetX = 600;
    SoccerBall.originX = parseInt(
        Core.getComputedStyle(SoccerBall.div, "left"), 10);
    SoccerBall.increment =
        (SoccerBall.targetX - SoccerBall.originX) /
        (SoccerBall.duration * SoccerBall.frameRate);
    SoccerBall.x = SoccerBall.originX;
```

```
    SoccerBall.animate();
  },

  animate: function()
  {
    SoccerBall.x += SoccerBall.increment;

    if ((SoccerBall.targetX > SoccerBall.originX &&
        SoccerBall.x >= SoccerBall.targetX) ||
        (SoccerBall.targetX < SoccerBall.originX &&
        SoccerBall.x <= SoccerBall.targetX))
    {
      SoccerBall.x = SoccerBall.targetX;
    }
    else
    {
      setTimeout(SoccerBall.animate, 1000 / SoccerBall.frameRate)
    }

    SoccerBall.div.style.left = Math.round(SoccerBall.x) + "px";
  }
};

Core.start(SoccerBall);
```

Animating in Two Dimensions

The example above only animated the soccer ball horizontally, but it's quite easy to modify our program to deal with vertical movement as well:

path-based_motion2.js

```
var SoccerBall =
{
  init: function()
  {
    SoccerBall.frameRate = 25;
    SoccerBall.duration = 2;
    SoccerBall.div = document.getElementById("soccerBall");
    SoccerBall.targetX = 600;
    SoccerBall.targetY = 150;
    SoccerBall.originX = parseInt(
```

```
        Core.getComputedStyle(SoccerBall.div, "left"), 10);
    SoccerBall.originY = parseInt(
        Core.getComputedStyle(SoccerBall.div, "top"), 10);
    SoccerBall.incrementX =
        (SoccerBall.targetX - SoccerBall.originX) /
        (SoccerBall.duration * SoccerBall.frameRate);
    SoccerBall.incrementY =
        (SoccerBall.targetY - SoccerBall.originY) /
        (SoccerBall.duration * SoccerBall.frameRate);
    SoccerBall.x = SoccerBall.originX;
    SoccerBall.y = SoccerBall.originY;

    SoccerBall.animate();
  },
  animate: function()
  {
    SoccerBall.x += SoccerBall.incrementX;
    SoccerBall.y += SoccerBall.incrementY;

    if ((SoccerBall.targetX > SoccerBall.originX &&
        SoccerBall.x >= SoccerBall.targetX) ||
        (SoccerBall.targetX < SoccerBall.originX &&
        SoccerBall.x <= SoccerBall.targetX))
    {
      SoccerBall.x = SoccerBall.targetX;
      SoccerBall.y = SoccerBall.targetY;
    }
    else
    {
      setTimeout(SoccerBall.animate, 1000 / SoccerBall.frameRate)
    }

    SoccerBall.div.style.left = Math.round(SoccerBall.x) + "px";
    SoccerBall.div.style.top = Math.round(SoccerBall.y) + "px";
  }
};

Core.start(SoccerBall);
```

For every instance in which we performed a calculation with the *x* coordinate, we add an equivalent statement for the *y* coordinate. So, `SoccerBall.init` ends up with a vertical end-point in `SoccerBall.targetY`, a vertical origin in

`SoccerBall.originY`, and a vertical position tracker in `SoccerBall.y`. These variables are used by `SoccerBall.animate` to increment the vertical position and write it to the `style.top` property.

The only other change we need to make is a small tweak to the starting position of the ball:

path-based_motion2.css *(excerpt)*

```
#soccerBall {
  ⋮
  top: 0;
  ⋮
}
```

Once that's done, we've got a ball that moves in both dimensions, as Figure 5.12 illustrates.

Figure 5.12. Adding vertical movement to the animation

Creating Realistic Movement

The animation that we've created so far treats the movement of the soccer ball homogeneously—each frame moves the ball by the same amount, and the ball stops without any deceleration. But this isn't how objects move in the real world: they

speed up, they slow down, they bounce. It's possible for us to mimic this type of behavior by using different algorithms to calculate the movement of our soccer ball.

If we wanted to get our soccer ball to slow to a halt, there are just a few minor tweaks we'd have to make to our program:

path-based_motion3.js

```
var SoccerBall =
{
  init: function()
  {
    SoccerBall.frameRate = 25;
    SoccerBall.deceleration = 10;
    SoccerBall.div = document.getElementById("soccerBall");
    SoccerBall.targetX = 600;
    SoccerBall.targetY = 150;
    SoccerBall.originX = parseInt(
        Core.getComputedStyle(SoccerBall.div, "left"), 10);
    SoccerBall.originY = parseInt(
        Core.getComputedStyle(SoccerBall.div, "top"), 10);
    SoccerBall.x = SoccerBall.originX;
    SoccerBall.y = SoccerBall.originY;

    SoccerBall.animate();
  },

  animate: function()
  {
    SoccerBall.x += (SoccerBall.targetX - SoccerBall.x) /
        SoccerBall.deceleration;
    SoccerBall.y += (SoccerBall.targetY - SoccerBall.y) /
        SoccerBall.deceleration;

    if ((SoccerBall.targetX > SoccerBall.originX &&
        Math.round(SoccerBall.x) >= SoccerBall.targetX) ||
        (SoccerBall.targetX < SoccerBall.originX &&
        Math.round(SoccerBall.x) <= SoccerBall.targetX))
    {
      SoccerBall.x = SoccerBall.targetX;
      SoccerBall.y = SoccerBall.targetY;
    }
    else
    {
```

```
      setTimeout(SoccerBall.animate, 1000 / SoccerBall.frameRate)
    }

    SoccerBall.div.style.left = Math.round(SoccerBall.x) + "px";
    SoccerBall.div.style.top = Math.round(SoccerBall.y) + "px";
  }
};

Core.start(SoccerBall);
```

In this code, we've replaced `SoccerBall.duration` with `SoccerBall.deceleration`—a deceleration factor that's applied to our new position calculation inside `animate`. This time, instead of dividing the total distance into neat little increments, the position calculator takes the distance *remaining* to the end-point, and divides it by the deceleration factor. In this way, the steps start out being big, but as the ball moves closer to its goal, the steps become smaller and smaller. The ball travels a smaller distance between frames, creating the sense that it's slowing down, or decelerating.

One pitfall that you need to watch out for when you're using an equation like this is that, if left to its own devices, `SoccerBall.x` will never reach the end point. The animation function will continually divide the remaining distance into smaller and smaller steps, producing an infinite loop. To counteract this (and stop the animation from going forever), we round `SoccerBall.x` to the nearest integer, using `Math.round`, before checking whether it has reached the end point. As soon as `SoccerBall.x` is within 0.5 pixels of the end point—which is close enough for the ball to reach its goal when its position is expressed in pixels—the rounding will cause the animation to end.

Using this deceleration algorithm, the movement of the ball looks more like that depicted in Figure 5.13.

Figure 5.13. Modeling realistic deceleration by changing the algorithm that moves the soccer ball

If you want the ball to move more slowly—gliding more smoothly to its final position—increase the deceleration factor. If you want the ball to reach its destination more quickly—coming to a more sudden stop—decrease the deceleration factor. Setting the factor to 1 causes the ball to jump directly to its destination.

Faster!

If you wanted to accelerate the soccer ball—make it start off slow and get faster—then you'd have to reverse the algorithm we used for deceleration:

path-based_motion4.js *(excerpt)*

```
var SoccerBall =
{
  init: function()
  {

    SoccerBall.frameRate = 25;
    SoccerBall.acceleration = 2;
    SoccerBall.threshold = 0.5;
    SoccerBall.div = document.getElementById("soccerBall");
    SoccerBall.targetX = 600;
    SoccerBall.targetY = 150;
    SoccerBall.originX = parseInt(
        Core.getComputedStyle(SoccerBall.div, "left"));
```

```
    SoccerBall.originY = parseInt(
        Core.getComputedStyle(SoccerBall.div, "top"));

    if (SoccerBall.targetX < SoccerBall.originX)
    {
      SoccerBall.x = SoccerBall.originX - SoccerBall.threshold;
    }
    else
    {
      SoccerBall.x = SoccerBall.originX + SoccerBall.threshold;
    }

    SoccerBall.distanceY = SoccerBall.targetY - SoccerBall.originY;

    SoccerBall.animate();
  },

  animate: function()
  {
    SoccerBall.x += (SoccerBall.x - SoccerBall.originX) /
        SoccerBall.acceleration;
    var movementRatio = (SoccerBall.x - SoccerBall.originX) /
        (SoccerBall.targetX - SoccerBall.originX);
    var y = SoccerBall.originY + SoccerBall.distanceY *
        movementRatio;

    if ((SoccerBall.targetX > SoccerBall.originX &&
        SoccerBall.x >= SoccerBall.targetX) ||
        (SoccerBall.targetX < SoccerBall.originX &&
        SoccerBall.x <= SoccerBall.targetX))
    {
      SoccerBall.x = SoccerBall.targetX;
      y = SoccerBall.targetY;
    }
    else
    {
      setTimeout(SoccerBall.animate, 1000 / SoccerBall.frameRate)
    }

    SoccerBall.div.style.left = Math.round(SoccerBall.x) + "px";
    SoccerBall.div.style.top = Math.round(y) + "px";
  }
```

```
};

Core.start(SoccerBall);
```

Here, the `SoccerBall.deceleration` variable has been replaced by `SoccerBall.acceleration`, and the value has been lowered to give us a quicker start. The algorithm that calculates the increments now uses the distance to the start point to determine them, so as the ball gets further from the start point the increments get bigger, making the ball move faster.

As this algorithm uses `SoccerBall.x – SoccerBall.originX` as the basis for its calculations, we need `SoccerBall.x` to initially be offset slightly from `SoccerBall.originX`, otherwise the ball would never move. `SoccerBall.x` must be offset in the direction of the destination point, so during initialization we check the direction in which the destination lies from the origin, and add or subtract a small value—`SoccerBall.threshold`—as appropriate. For a more accurate model, you could reduce `SoccerBall.threshold` below `0.5`, but you won't see much difference.

The other difference between this algorithm and the decelerating one is the way in which the vertical positions are calculated. Since we use a fixed value for acceleration, if it was calculated separately, the acceleration in either dimension would be the same—the *y* position would increase in speed at the same rate as the *x* position. If your *y* destination coordinate is closer than your *x* destination coordinate, the soccer ball would reach its final *y* position faster than it reached its final *x* position, producing some strange (non-linear) movement.

In order to prevent this eventuality, we calculate the *y* position of the soccer ball based on the value of its *x* position. After the new *x* position has been calculated, the ratio between the distance traveled and the total distance is calculated and placed in the variable `movementRatio`. This figure is multiplied by the total vertical distance between the origin and the target, and gives us an accurate *y* position for the soccer ball, producing movement in a straight line.

As the y position is calculated in terms of the *x* position, we no longer need `SoccerBall.y`, so this property has been removed from the object. Instead, we just

calculate a normal *y* variable inside the `animate` function. As a quick reference for the vertical distance, `SoccerBall.distanceY` is calculated upon initialization.

Moving Ahead

You can give countless different types of movements to an object. Deceleration and acceleration are just two of the more simple options; there's also bouncing, circular movements, elastic movements—the list goes on. If you'd like to learn the mathematics behind other types of object movement, I highly recommend visiting Robert Penner's site.[2] He created the movement calculations for the Flash environment, so I'm pretty sure he knows his stuff.

Revisiting the Accordion Control

Well, you learned to make an accordion control in Chapter 4, and I'm sure you were clicking around it, collapsing and expanding the different sections, but thinking, "This isn't quite there; it needs some more … snap!" Well, it's now time to make it snap, move, and jiggle. Because we're going to add some animation to that accordion. After all, what's an accordion without moving parts, and a little monkey in a fez?

This upgrade will take a little more effort than the modifications we made to our Rich Tooltips script, but the effect is well worth it—we'll wind up with an animated accordion that gives the user a much better sense of what's going on, and looks very, *very* slick.

Making the Accordion Look Like it's Animated

Before we dive into the updated code, let's take a look at *how* we're going to animate the accordion. To produce the effect of the content items collapsing and expanding, we have to modify the height of an element that contains all of that content. In this case, the list items in the menu are the most likely candidates.

Normally, if no height is specified for a block-level element, it will expand to show all of the content that it contains. But if you *do* specify a height for a block-level element, it will always appear at that specified height, and any content that doesn't fit inside the container will spill out, or **overflow**, as shown in Figure 5.14.

[2] http://www.robertpenner.com/easing/

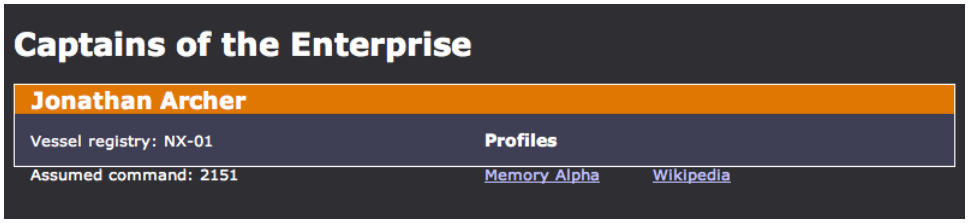


Figure 5.14. Content overflowing the bounds of a block-level element for which height is specified

Now, we don't want the content to overflow when we're animating our accordion; we want the parent element to hide any content that would ordinarily overflow. The way to do this is to adjust the CSS on the parent container, and specify the property `overflow: hidden`. When this is done, and a height (or a width) is specified, any overflowing content will be hidden from view, as illustrated in Figure 5.15.

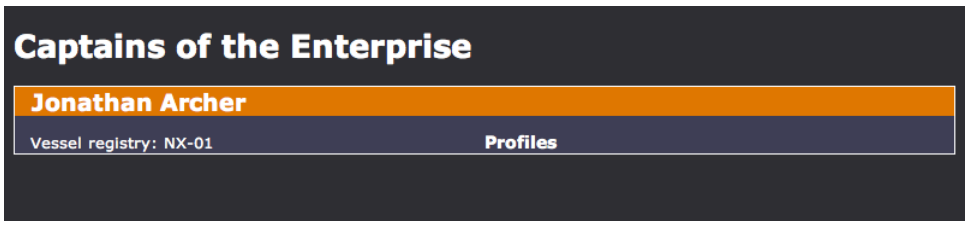


Figure 5.15. Specifying `overflow: hidden` on the block-level element causing content that doesn't fit inside it to be hidden

Once we've made sure that the overflowing content is being hidden correctly, we can start to animate the container. To do this, we'll gradually change its height to make it look like it's expanding or collapsing.

Changing the Code

Now that you've got the general idea of how we're going to animate our accordion, our first stop is the initialization method:

accordion_animated.js *(excerpt)*

```
init: function()
{
  Accordion.frameRate = 25;
  Accordion.duration = 0.5;
```

```
var accordions = Core.getElementsByClass("accordion");

for (var i = 0; i < accordions.length; i++)
{
  var folds = accordions[i].childNodes;
  for (var j = 0; j < folds.length; j++)
  {
    if (folds[j].nodeType == 1)
    {
      var accordionContent = document.createElement("div"); (1)
      accordionContent.className = "accordionContent";

      for (var k = 0; k < folds[j].childNodes.length; k++) (2)
      {
        if (folds[j].childNodes[k].nodeName.toLowerCase() !=
            "h2") (3)
        {
          accordionContent.appendChild(folds[j].childNodes[k]); (4)
          k--; (5)
        }
      }

      folds[j].appendChild(accordionContent); (6)
      folds[j]._accordionContent = accordionContent; (7)

      Accordion.collapse(folds[j]);
      var foldLinks = folds[j].getElementsByTagName("a");
      var foldTitleLink = foldLinks[0];
      Core.addEventListener(foldTitleLink, "click",
          Accordion.clickListener);

      for (var k = 1; k < foldLinks.length; k++)
      {
        Core.addEventListener(foldLinks[k], "focus",
            Accordion.focusListener);
      }
    }
  }

  if (location.hash.length > 1)
  {
    var activeFold =
        document.getElementById(location.hash.substring(1));
    if (activeFold && activeFold.parentNode == accordions[i])
```

```
      {
        Accordion.expand(activeFold);
      }
    }
  }
}
```

In this code, we've added two familiar animation constants, as well as a whole new block of code that modifies the HTML of the menu. Why?

One point you should note about using the `overflow: hidden` CSS property is that it can produce some weird visual effects in Internet Explorer 6. Given the way we've set up the accordion headings, if we applied `overflow: hidden` to those list items, they'd look completely warped in IE. To circumvent this pitfall, we separate the heading (the part that the user clicks on) from the rest of the content in a given fold of the accordion by putting that content inside its own `div` element. It's also a lot easier to deal with the animation if we can collapse the container to zero height, rather than having to worry about leaving enough height for the heading.

But that's no reason to go in and modify your HTML by hand! We can easily make these changes with JavaScript, as the code above shows:

1. Inside `init`, we create a new `div` and assign it a `class` of `accordionContent`.
2. Next, we have to move the current contents of the list item into this new container. We do so using a `for` loop that iterates through each of the list item's `childNodes`.
3. We don't want to move the `h2` into that new container, because that's the clickable part of the accordion, so we do a check for that, skip it, and include everything else.
4. You can move an element from one parent to another simply by appending the element to the new parent. The DOM will automatically complete the process of removing and adding the element in the position you're identified.
5. One trick with that `for` loop is that it decrements the counter every time a child element is moved. The reason for this is that the counter automatically *incre-*

ments every time the loop executes, but if we remove an element from `childNodes`, its next sibling moves down to take its index, so if we actually incremented the index, we'd start to skip elements. Decrementing cancels out this effect.

6. Once all the existing content has been moved into the new container, we're able to append that container into the list item, completing our modification of the DOM. Now our list item contains only two elements—the title and the content—and no one's the wiser!

7. Finally, as a shortcut for later, we store a reference to the `accordionContent` element as a property of the list item: `_accordionContent`.

Another advantage of adding the `accordionContent` `div` is that it simplifies the CSS. For example, the CSS code that hides the contents of collapsed folds can be distilled down to this:

accordion_animated.css *(excerpt)*

```
.accordionContent {
  overflow: hidden;
  ⋮
}

li.collapsed .accordionContent {
  position: absolute;
  left: -9999px;
}

/* Fixes Safari bug that prevents expanded content from displaying.
   See http://betech.virginia.edu/bugs/safari-stickyposition.html */
li.collapsed .accordionContent p {
  position: relative;
}
```

Thanks to this code, `overflow: hidden` will always be specified on elements with the `class` `accordionContent`, enabling us to animate them properly.

In the original script, our `expand` function changed some classes to make the selected accordion item pop open, but in *animating* the accordion, we need to use `expand` to do a little setup first:

accordion_animated.js *(excerpt)*

```
expand: function(fold)
{
  var content = fold._accordionContent; (1)

  Accordion.collapseAll(fold.parentNode);
  if (!Core.hasClass(fold, "expanded")) (2)
  {
    content.style.height = "0"; (3)
    content._height = 0; (4)
    Core.removeClass(fold, "collapsed");
    Core.addClass(fold, "expanded");
    content._increment = content.scrollHeight /
        (Accordion.frameRate * Accordion.duration); (5)
    Accordion.expandAnimate(content); (6)
  }
},
```

(1) The `content` variable is just a shortcut to the `fold._accordionContent` property we created earlier—it saves plenty of typing. The `collapseAll` method is then called to reset the accordion menu items, and we can focus our attention on the currently selected content.

(2) Although previously it didn't matter if `expand` was called on an already-expanded fold (for example, in response to the keyboard focus moving from link to link within an expanded fold), we've changed this method so that it kicks off an animation. As such, we need this `if` statement to avoid animating already-expanded folds.

(3) Before we change its classes to make it visible, we set `content`'s `style.height` to `0`. This step ensures that when the class change switches the content from collapsed to expanded, it will remain invisible, because it will have no height.

(4) To keep track of the actual calculated height of the content, we create the variable `_height` as a property of content. As in the previous animation ex-

amples, this property allows us to keep an accurate calculation of the accordion's movement. Once the height has been reset, we can remove the "collapsed" class and add the "expanded" class.

5. We're almost ready to perform the animation, but before we do that, we need to check the height to which we're going to expand the accordion item. As we'll be increasing the height of the item from zero, we have to know when all the content is displayed, so we can work out when to stop. The `scrollHeight` property lets us do this—it calculates the height of the content irrespective of whether we cut it off with an explicit height and `overflow: hidden`.

   We can save a bit of repetitive calculation within the animating function by calculating in advance the movement increment that we'll apply in each step of the animation. This is determined by dividing the total height of the content (`content.scrollHeight`) by the total number of frames (`Accordion.frameRate * Accordion.duration`). The result of this calculation is assigned to the `_increment` property of `content`.

6. After all that setup, we can call `expandAnimate`, our new method that's designed to animate the expansion of an accordion item.

Let's take a look at that animation method:

accordion_animated.js *(excerpt)*

```
expandAnimate: function(content)
{
  var newHeight = content._height + content._increment; 1

  if (newHeight > content.scrollHeight) 2
  {
    newHeight = content.scrollHeight; 3
  }
  else
  {
    content._timer = setTimeout(function() 4
        {
          Accordion.expandAnimate(content);
        }, 1000 / Accordion.frameRate);
  }
```

```
  content._height = newHeight; ❺
  content.style.height = Math.round(newHeight) + "px";
  content.scrollTop = 0; ❻
},
```

❶ `expandAnimate` starts off by calculating the new height of the content: `content._height + content._increment`.

❷ This `newHeight` variable is used inside a conditional test to detect whether the animation should finish.

❸ If `newHeight` is larger than `content.scrollHeight`, we change `newHeight` to equal the height of the content, and we don't schedule any more animation cycles.

❹ But if `newHeight` is less than the content's height, we schedule another `setTimeout` call to the same function. This step produces the iterative animation we're looking for. The `setTimeout` ID is assigned to the `_timer` property of the `content` element, so that if another accordion item is clicked mid-animation, we can stop the expansion animation from continuing.

❺ After we've figured out what the new height of the content should be, we use that new height to update the element's `_height` property, then change its appearance using a rounded value for `style.height`. One frame of the animation is now complete.

❻ In certain browsers, if keyboard focus moves to a hyperlink inside a collapsed fold of the accordion, the browser will make a misguided attempt to scroll that collapsed content in order to make the link visible. If left this way, the content, once expanded, will not display properly. Thankfully, the fix is easy—after each frame of the animation, we reset the `content`'s `scrollTop` property to zero, which resets its vertical scrolling position.

With those two methods, you can see how animation works—the event listener is fired only once, and sets up the initial values for the animation. Then the iterative function is called via `setTimeout` to produce the visual changes needed to get to the final state.

Collapsing an item works in much the same fashion:

accordion_animated.js *(excerpt)*

```
collapse: function(fold)
{
  var content = fold._accordionContent;
  content._height = parseInt(content.style.height, 10);
  content._increment = content._height /
      (Accordion.frameRate * Accordion.duration);

  if (Core.hasClass(fold, "expanded"))
  {
    clearTimeout(fold._accordionContent._timer);
    Accordion.collapseAnimate(fold._accordionContent);
  }
  else
  {
    Core.addClass(fold, "collapsed");
  }
},
```

For the `collapse` method, we have to move the class changes into the animation method because we need the item to remain expanded until the animation has finished (otherwise it will disappear immediately, before the animation has had a chance to take place). To kick off the animation, we check whether the current item has the class `expanded`, and if it does, we send the browser off to execute `collapseAnimate`, but not before we cancel any expansion animation that's currently taking place. If we didn't cancel the expansion first, the collapsing animation might coincide with an expansion animation that was already in progress, in which case we'd end up with a stuck accordion (never a pretty sight—or sound).

If an element doesn't have the `expanded` class on it when a collapse is initiated, we simply add the `collapsed` class and forego the animation. This takes care of the circumstance in which the page first loads and we need to hide all the menu items.

`collapseAnimate` is a lot like `expandAnimate`, but in reverse:

accordion_animated.js *(excerpt)*

```
collapseAnimate: function(content)
{
  var newHeight = content._height - content._increment; (1)

  if (newHeight < 0)
  {
    newHeight = 0;
    Core.removeClass(content.parentNode, "expanded"); (2)
    Core.addClass(content.parentNode, "collapsed");
  }
  else
  {
    content._timer = setTimeout(function()
        {
          Accordion.collapseAnimate(content);
        }, 1000 / Accordion.frameRate);
  }

  content._height = newHeight;
  content.style.height = Math.round(newHeight) + "px";
}
```

(1) We're aiming to get the `height` of the `content` element to `0`, so instead of adding the increment, we subtract it.

(2) The `if-else` statement that's used when we reach our target height is slightly different than the one we saw above, because we have to change the classes on the current item. Otherwise, it's identical to `expandAnimate`.

With animation occurring in both directions, we now have a fully functioning animated accordion. Of course, you'll get the best view of its effect if you check out the demo in the example files, but Figure 5.16 shows what happens when you click to open a new accordion item.

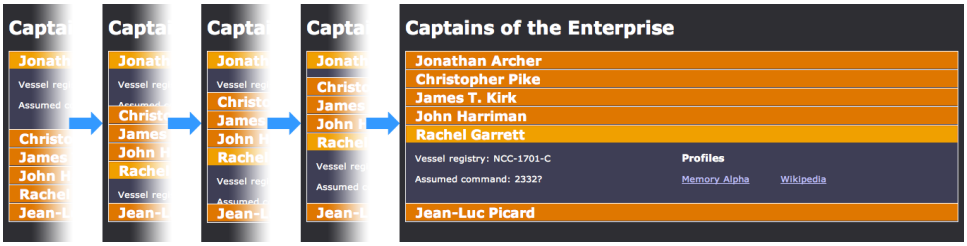


Figure 5.16. The progression of our animated accordion as one item collapses and another expands

Exploring Libraries

Quite a few of the main JavaScript libraries don't tackle animation; instead, they focus on core tasks like DOM manipulation and styling. However, around these have sprung up a number of little libraries devoted to animation tasks, and they do the job quite well.

The animation libraries are a lot more generalized than the scripts we've written here, so it's possible to use them to apply a range of effects to almost any element, depending on how adventurous you're feeling.

script.aculo.us

script.aculo.us is probably the most well-known effects library available. It's actually an add-on to Prototype, which it uses for its DOM access and architecture capabilities, so if you want to run script.aculo.us, you'll also need to include Prototype.

script.aculo.us comes with a host of effects that have become popular in the so-called Web 2.0 age: fading, highlighting, shrinking, and many other transitions. It has also more recently included larger pieces of functionality such as drag-and-drop and slider widgets. Here, we'll focus on the effects.

All of the effects in script.aculo.us are available through the `Effect` object, which will be available in your programs if you include the **scriptaculous.js** file on your page.

Let's imagine that we have a paragraph of text that we want to highlight:

scriptaculous_highlight.html *(excerpt)*

```
<p id="introduction">
  Industrial Light &amp;amp; Magic (ILM) is a motion picture visual
  effects company, founded in May 1975 by George Lucas and owned
  by Lucasfilm Ltd. Lucas created the company when he discovered
  that the special effects department at Twentieth Century Fox was
  shut down after he was given the green light for his production
  of Star Wars.
</p>
```

We can do so by passing the ID string to `Effect.Highlight`:

```
new Effect.Highlight("introduction");
```

As soon as you make this call, the effect will be applied to the element, as shown in Figure 5.17.

Figure 5.17. Creating a yellow fade effect with script.aculo.us

`Effect.Highlight` also allows you to pass it a DOM node reference, so you could apply the effect to the paragraph like this:

```
new Effect.Highlight(document.getElementsByTagName("p")[0]);
```

Most of the effects in script.aculo.us have optional parameters that allow you to customize various aspects of the effect. These parameters are specified as properties inside an object literal, which itself is passed as an argument to the effect.

For instance, by default, `Effect.Highlight` fades the background from yellow to white, but it's possible to specify the start color and the end color, so we could make a particularly lurid fade from red to blue like this:

```
new Effect.Highlight("introduction",
    {startcolor: "#FF0000", endcolor: "#0000FF"});
```

These optional parameters will differ from effect to effect, so you'll have to read through the script.aculo.us documentation if you wish to customize a particular effect.

Most of the effects have a duration parameter, which lets you specify how quickly you want the effect to occur. This parameter is specified in seconds, so if you wanted a two-second, lurid red-blue fade, you'd include all these parameters:

scriptaculous_highlight.js *(excerpt)*

```
new Effect.Highlight("introduction",
    {startcolor: "#FF0000", endcolor: "#0000FF",
    duration: 2});
```

script.aculo.us also has a nice little event model that allows you to trigger functions while effects are happening, or after they have finished. These events are again specified as parameters in the object literal, and take a function name as a value. That function will be called when the event is fired:

```
function effectFinished()
{
  alert("The introduction has been effected");
}

new Effect.Highlight("introduction",
    {afterFinish: effectFinished});
```

That code pops up an alert dialog once the fade has finished, to tell us it's done.

script.aculo.us is certainly a powerful and flexible effects library. Its popularity has largely been fueled by its ease of integration and execution. In case these benefits weren't already apparent, I'll leave you with this nugget: using script.aculo.us, we could have animated our soccer ball with just one line of code:

```
new Effect.MoveBy("soccerBall", 150, 600);
```

But that wouldn't have been half as much fun, would it?

Summary

HTML was designed to be a static medium, but using JavaScript, we can bring it to life.

There's no doubt that animation can add a lot of polish to an interface—it doesn't have to be mere eye candy! Animation provides real benefits in guiding users around and providing them with visual cues as to functionality and state. I'm sure quite a few useful ideas have sprung into your mind while you've been reading. This chapter has given you a taste of what you can do with time-based processing, but there's so much more to explore once you've learned the basics.

Next up, we'll delve deeper into a subject that's central to the development of web applications: forms.

Chapter 6

Form Enhancements

HTML hasn't changed much in the past ten years. Since browsers implemented the changes in HTML 4 around 1997, the collection of tags and attributes that we use on the web has essentially stayed the same. Thanks to ongoing improvements to CSS support in browsers, we have been able to create richer, more intricate designs with that limited markup. Like the designer fashions that are paraded down the runway each season, the styles are always fresh and new, even if the models underneath look eerily alike.

But there is one aspect of web design for which CSS can't hide the stagnation of HTML: forms. No matter how we dress them up, HTML has for the past decade supported only this limited set of form controls:

- text input fields
- checkboxes
- radio buttons
- drop-down menus and lists
- multi-line text areas
- buttons

Since this selection is so limited, forms were one of the first parts of HTML to receive the attention of developers experimenting with JavaScript enhancements.

In this chapter, we'll build a number of useful, reusable form enhancements of our own. This will give you an opportunity to apply many of the skills you've gathered in the book so far, and to learn a new trick or two.

HTML DOM Extensions

Form controls aren't your everyday HTML elements. They offer built-in behavior and interactivity far beyond that which can be described with events like `click`, `mouseover`, `mouseout`, `focus`, and `blur`, and properties like `id`, and `className`.

Consequently, in addition to the standard DOM properties, methods, and events that we've already played with in this book, form-related elements support a bunch of extra properties and methods that are defined in a separate section of the DOM standard.[1] The most useful of these properties and methods are listed in Table 6.2 and Table 6.1, respectively. Form controls also support a handful of extra events, which are listed in Table 6.3.

Table 6.1. Additional DOM Methods for HTML Form Controls

Method	Element(s)	Description

`blur`	`input`
	`select`
	`textarea`
		removes keyboard focus from this form control

`click`	`input`	simulates a mouse click on this control

`focus`	`input`
	`select`
	`textarea`
		gives keyboard focus to this form control

`reset`	`form`	restores all of this form's controls to their default values
`select`	`input` `textarea`	selects the text contents of this control
`submit`	`form`	submits the form without triggering a `submit` event

[1] http://www.w3.org/TR/DOM-Level-2-HTML/

In the next two examples, we'll play with some—but not all—of these form-specific DOM features. Once you've got a feel for the extra functionality that JavaScript can provide, come back to these tables and think about *other* ways you can use these tools to enhance your forms.

Table 6.2. Additional DOM Properties for HTML Form Controls

Property	Element(s)	Description
`elements`	`form`	a node list containing all of the form controls in the form
`checked`	`input`	for `input`s of `type` `checkbox` and `radio`, this property's value is `true` if the control is selected

`disabled`	`button`
	`input`
	`optgroup`
	`option`
	`select`
	`textarea`
		when `true`, the control is unavailable to the user

`form`	`button`
	`input`
	`option`
	`select`
	`textarea`
		a reference to the `form` element that contains this control

`index`	`option`	the index of this `option` within the `select` element that contains it (0 for the first)
`options`	`select`	a node list containing all of the `option` elements in this menu
`selected`	`option`	`true` if this option is currently selected, `false` if not
`selectedIndex`	`select`	the index of the currently-selected `option` in the list (0 for the first)

`value`	`button`
	`input`
	`option`
	`select`
	`textarea`
		the current value of this control, as it would be submitted to the server

Table 6.3. Additional DOM Events for HTML Form Controls

Event	Element(s)	Triggered when...

`change`	`input`
	`select`
	`textarea`
		the control has lost focus after its value has been changed

`select`	`input` `textarea`	the user has selected some text in the field
`submit`	`form`	the user has requested that the form be submitted

Example: Dependent Fields

You don't always need your users to fill in every field in a form. Sometimes, the value a user enters into *one* form field renders *another* field irrelevant. Instead of relying on the user to figure out which fields are relevant, why not use the `disabled` property mentioned in Table 6.2 to disable the fields that the user can safely ignore?

In Figure 6.1, the checkbox following the second radio button is only relevant if that radio button is selected. We can use JavaScript to disable it in all other cases.

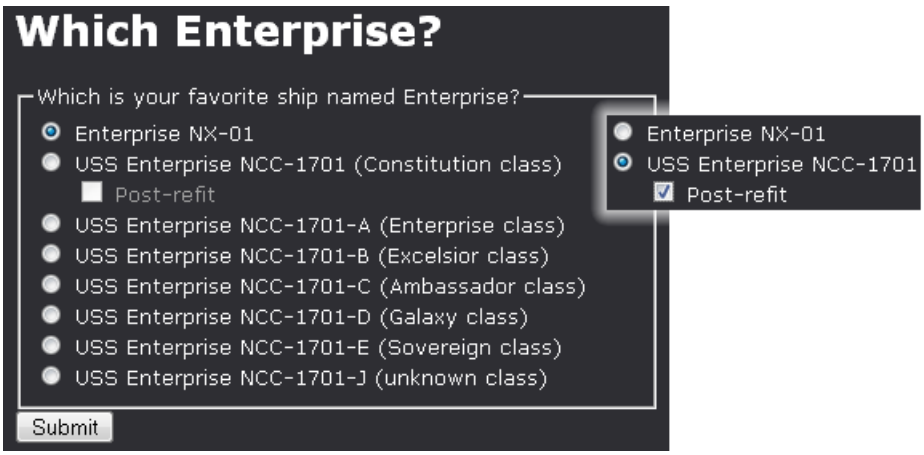


Figure 6.1. The checkbox is relevant only if the user selects the second radio button

The best way to implement dependent fields will depend (so to speak) on the specific logic of your form, but for the purposes of this example we can take an approach that's general enough to make the script useful in many situations. To this end, we'll make the following assumptions:

- Each form will have only one level of dependency (a dependent field cannot be dependent on another dependent field).
- Dependent fields can only be `input` elements of `type` `text`, `password`, `checkbox`, or `radio`.
- Dependent fields can only depend on `input` elements of these types.
- Dependent fields will immediately follow the fields on which they depend in the document.
- Dependent fields will be contained within `label` elements.

Here's how we'll mark up a form field (in this case, a radio button), and another field (a checkbox) that's dependent on it:

dependentfields.html *(excerpt)*

```
<div>
  <label for="ncc1701">
    <input type="radio" name="whichenterprise"
        value="ncc1701" id="ncc1701" />
    USS Enterprise NCC-1701 (Constitution class)
  </label>
  <label for="ncc1701refit" class="secondary">
    <input type="checkbox" class="dependent"
        name="ncc1701refit" id="ncc1701refit"
        value="ncc1701refit" />
    Post-refit
  </label>
</div>
```

Note that the checkbox has a `class` of `dependent`—that's how we indicate that the checkbox is dependent on the field that precedes it. If the radio button had more than one dependent field, each one would have a `class` of `dependent`.

When this markup is given to a browser in which JavaScript is disabled, the form's dependent fields will simply remain enabled at all times.

We'll put our script in an object called `DependentFields`:

dependentfields.js *(excerpt)*

```
var DependentFields =
{
  init: function()
  {
    ⋮
  },
  ⋮
};

Core.init(DependentFields);
```

Let's start our script by writing the workhorse functions, which will actually enable and disable form fields as needed:

dependentfields.js *(excerpt)*

```
disable: function(field)
{
  field.disabled = true;
  Core.addClass(field, "disabled");
  Core.addClass(field.parentNode, "disabled");
},

enable: function(field)
{
  field.disabled = false;
  Core.removeClass(field, "disabled");
  Core.removeClass(field.parentNode, "disabled");
},
```

That's simple enough, right? Both functions start by setting the field's `disabled` property. The first function then applies the `disabled` class on both the field itself and the element that contains it (the field's `label`); the second function removes this class from the field and its containing element. We'll use the `disabled` class to style disabled fields (and their labels):

dependentfields.css *(excerpt)*

```
label.disabled {
  color: #A0A0A0;
}
```

If you prefer, you can actually use a technique like offleft positioning (which I described in Chapter 4) to hide disabled fields entirely!

Now, we can approach the code in a number of ways that will actually call the `disable` and `enable` methods we've just seen. The most obvious approach would be to add event listeners to each `input` element that had one or more dependent fields, and have those listeners enable or disable the relevant dependent fields.

Having attempted this, I can tell you now that this approach, though obvious, is problematic. For example, a radio button won't always produce a useful event when it's deselected. Sometimes, the only hint that JavaScript gets that a radio button has been deselected is a `click` event from the radio button next to it.

Another option is to add a couple of event listeners to each form on the page. These event listeners will watch for `click` and `change` events bubbling up from *any* of the fields in the form, and update the state of all the dependent fields in that form after every such event:

dependentfields.js *(excerpt)*

```
init: function()
{
  var forms = document.getElementsByTagName("form");

  for (var i = 0; i < forms.length; i++)
  {
    Core.addEventListener(
        forms[i], "change", DependentFields.changeListener);
    Core.addEventListener(
        forms[i], "click", DependentFields.clickListener);
```

To make the jobs of these listeners easier (after all, they'll be running quite often), the `init` method will also scan each form to build a list of the dependent fields it contains, and to store for each one a reference to the field upon which it depends:

dependentfields.js *(excerpt)*

```
var fields = forms[i].getElementsByTagName("input"); (1)
var lastIndependentField = null;
forms[i]._dependents = []; (2)
for (var j = 0; j < fields.length; j++)
{
  if (!Core.hasClass(fields[j], "dependent")) (3)
  {
    lastIndependentField = fields[j]; (4)
  }
  else
  {
    if (lastIndependentField) (5)
    {
      forms[i]._dependents[forms[i]._dependents.length] =
          fields[j]; (6)
      fields[j]._master = lastIndependentField; (7)
    }
  }
}
```

This code may seem a little convoluted at first glance, so let me break it down for you:

1. We get a list of all the `input` elements in the form.
2. We create for the form a custom property named `_dependents`, in which we'll store a list of all the dependent fields in the form. To start with, however, we initialize it with an empty array.
3. For each field in the form, we perform a check to see if it's a dependent field or not.
4. If it's an independent field, we store a reference to it in a variable called `lastIndependentField`.

5. If it's a dependent field, we double-check that we've already got a reference to the field on which it will depend.

6. We then add to the form's `_dependents` array a reference to this dependent field.

7. Finally, we store in a custom property named `_master` a reference to the field upon which the dependent field depends.

What this code gives us is a list of all the dependent fields in each form (*`form`*`._dependents`), as well as a link from each dependent field to the field upon which it depends (*`dependentField`*`._master`).

The last thing our `init` method will do for each form is set the initial states of all the dependent fields. Since this is a fairly complex process, we'll write a separate method, called `updateDependents`, to achieve it:

dependentfields.js *(excerpt)*

```
    DependentFields.updateDependents(forms[i]);
  }
},
```

Both of our event listeners will kick off the same process of updating the dependent field states, so both listeners will call this same method:

dependentfields.js *(excerpt)*

```
changeListener: function(event)
{
  DependentFields.updateDependents(this);
},

clickListener: function(event)
{
  DependentFields.updateDependents(this);
}
```

All that's left is to write that all-important `updateDependents` method, which will either enable or disable each dependent field in a form on the basis of the state of the field on which it depends:

dependentfields.js *(excerpt)*

```
updateDependents: function(form)
{
  var dependents = form._dependents; (1)
  if (!dependents)
  {
    return;
  }

  for (var i = 0; i < dependents.length; i++) (2)
  {
    var disabled = true; (3)
    var master = dependents[i]._master; (4)

    if (master.type == "text" || master.type == "password") (5)
    {
      if (master.value.length > 0)
      {
        disabled = false;
      }
    }
    else if (master.type == "checkbox" ||
        master.type == "radio") (6)
    {
      if (master.checked)
      {
        disabled = false;
      }
    }

    if (disabled) (7)
    {
      DependentFields.disable(dependents[i]);
    }
    else
    {
      DependentFields.enable(dependents[i]);
```

```
    }
  }
},
```

Again, let's take this one step at a time:

1. We start by fetching the list of the form's dependent fields that was compiled by the `init` method.
2. We loop through this list, one dependent field at a time.
3. For each field, we start by assuming it will be disabled, then set out to determine whether that assumption is wrong.
4. To find out, we need to look at the field upon which this field depends—information that we can obtain from the `_master` property that we created in the `init` method.
5. If the master field is a `text` or `password` field, we check the `length` of its `value` property. If it's greater than zero, this dependent field should not be disabled.
6. If the master field is of `type` `checkbox` or `radio`, we look to see if it's `checked`. If it is, this dependent field should not be disabled.
7. Now that we know for sure whether this dependent field should or should not be disabled, we can call the `disable` or `enable` methods as required to set the appropriate state.

And there you have it! Our script has used a number of the HTML DOM extensions:

- We set up an event listener for the `change` event generated by some form controls.
- We used the `disabled` property, which is supported by all form controls, to disable dependent fields when appropriate.
- We used the `type`, `checked`, and `value` properties of various form controls to determine whether a field had been filled in, and thus whether we should enable its dependent fields.

Although our example only contains one dependent field, you can reuse this script on any page with any number of dependent fields, as long as the assumptions we stated at the start of this section are met. Here's the complete JavaScript code:

dependentfields.js *(excerpt)*

```
var DependentFields =
{
  init: function()
  {
    var forms = document.getElementsByTagName("form");

    for (var i = 0; i < forms.length; i++)
    {
      Core.addEventListener(forms[i], "change",
          DependentFields.changeListener);
      Core.addEventListener(forms[i], "click",
          DependentFields.clickListener);

      var fields = forms[i].getElementsByTagName("input");
      var lastIndependentField = null;
      forms[i]._dependents = [];
      for (var j = 0; j < fields.length; j++)
      {
        if (!Core.hasClass(fields[j], "dependent"))
        {
          lastIndependentField = fields[j];
        }
        else
        {
          if (lastIndependentField)
          {
            forms[i]._dependents[
                forms[i]._dependents.length] = fields[j];
            fields[j]._master = lastIndependentField;
          }
        }
      }
      DependentFields.updateDependents(forms[i]);
    }
  },

  disable: function(field)
  {
```

```
  field.disabled = true;
  Core.addClass(field, "disabled");
  Core.addClass(field.parentNode, "disabled");
},

enable: function(field)
{
  field.disabled = false;
  Core.removeClass(field, "disabled");
  Core.removeClass(field.parentNode, "disabled");
},

updateDependents: function(form)
{
  var dependents = form._dependents;
  if (!dependents)
  {
    return;
  }

  for (var i = 0; i < dependents.length; i++)
  {
    var disabled = true;
    var master = dependents[i]._master;

    if (master.type == "text" || master.type == "password")
    {
      if (master.value.length > 0)
      {
        disabled = false;
      }
    }
    else if (master.type == "checkbox" ||
        master.type == "radio")
    {
      if (master.checked)
      {
        disabled = false;
      }
    }

    if (disabled)
    {
      DependentFields.disable(dependents[i]);
```

```
      }
      else
      {
        DependentFields.enable(dependents[i]);
      }
    }
  },

  changeListener: function(event)
  {
    DependentFields.updateDependents(this);
  },

  clickListener: function(event)
  {
    DependentFields.updateDependents(this);
  }
};

Core.start(DependentFields);
```

Example: Cascading Menus

Another situation in which it can be useful to tie multiple form fields together with JavaScript arises when you have a complex `select` menu like that shown in Figure 6.2.

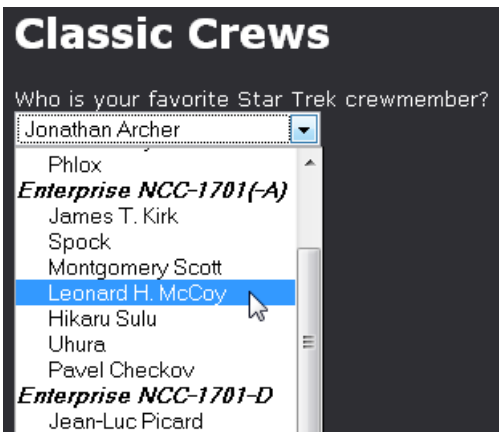


Figure 6.2. A two-level menu made up of `optgroups` and `options`

Here we have a `select` containing a number of `optgroup` elements, each of which contains a number of `option` elements. In effect, we have two levels of menus here—the user must first find the right option group, then select the desired option. Why not make it easier on the user by splitting this into two menus, as shown in Figure 6.3?

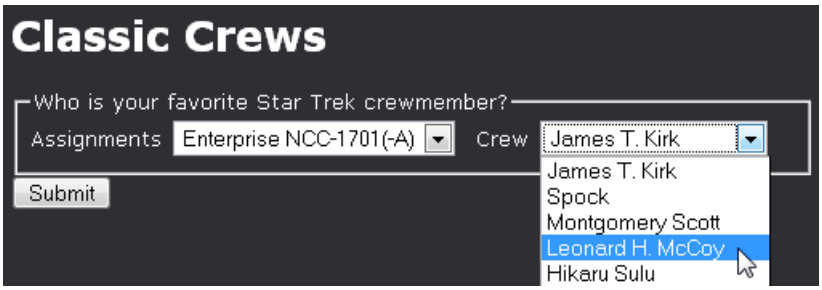


Figure 6.3. Splitting the single menu into a pair of cascading menus to improve usability

When the user selects an option from the first menu, our JavaScript code will swing into action, updating the list of options in the second menu. This saves users from having to scroll through menu choices that aren't in the group they're interested in.

Of course, our form must still work when JavaScript is disabled, so we'll write the HTML code for our form using the single `select`, then split it into the two menus during the initialization of our script. To make our script reusable, however, we need to make sure that the single menu contains all the information that's needed to produce the two cascaded menus:

cascadingmenu.html *(excerpt)*

```
<label for="crewselect">
  Who is your favorite Star Trek crewmember?
</label>
<select id="crewselect" name="crewselect"
    class="cascading" title="Assignments">
  <optgroup title="Crew" label="Enterprise NX-01">
    <option value="1">Jonathan Archer</option>
    <option value="2">T'Pol</option>
    <option value="3">Charles Tucker III</option>
    <option value="4">Malcolm Reed</option>
    <option value="5">Hoshi Sato</option>
    <option value="6">Travis Mayweather</option>
```

```
    <option value="7">Phlox</option>
  </optgroup>
  <optgroup title="Crew" label="Enterprise NCC-1701(-A)">
    ⋮
  </optgroup>
  ⋮
</select>
```

Notice in particular the `title` attributes that are highlighted in bold. The title describes the items that are listed in each element—the `select` contains a list of assignments, and each `optgroup` contains a crew listing. As you can see in Figure 6.3, these attribute values will become the `label`s for the menus that our script will create.

This transformation from one menu into two will form the bulk of the work that the `init` method of our script will complete. As Figure 6.4 shows, we're making a fairly drastic change to the document's structure.

We can break the process into a number of steps to make it more manageable:

1. Find within the page every `select` with a `class` of `cascading`.
2. For each one, convert the single `select`'s `label` into a `fieldset` that surrounds the `select`, with the former `label`'s content contained in the `fieldset`'s `legend`.
3. Create a new "master" `select` before the existing `select`. Give it the same `title` as the existing `select`, but new `name` and `id` attributes. Fill it with `option`s based on the `optgroup`s of the existing `select`, then change the existing `select`'s `title` to match the `optgroup`s' `title`s before removing all the contents of the existing `select`.
4. Just before each of the two `select`s that now exist, create a `label` containing the text from the `select`'s `title` attribute.
5. Fill the now-empty second `select` with the `option`s that correspond to the currently-selected `option` in the new master `select`. Repeat this process whenever the selection in the master `select` changes.

We can use this breakdown as a blueprint for our script's `init` method:

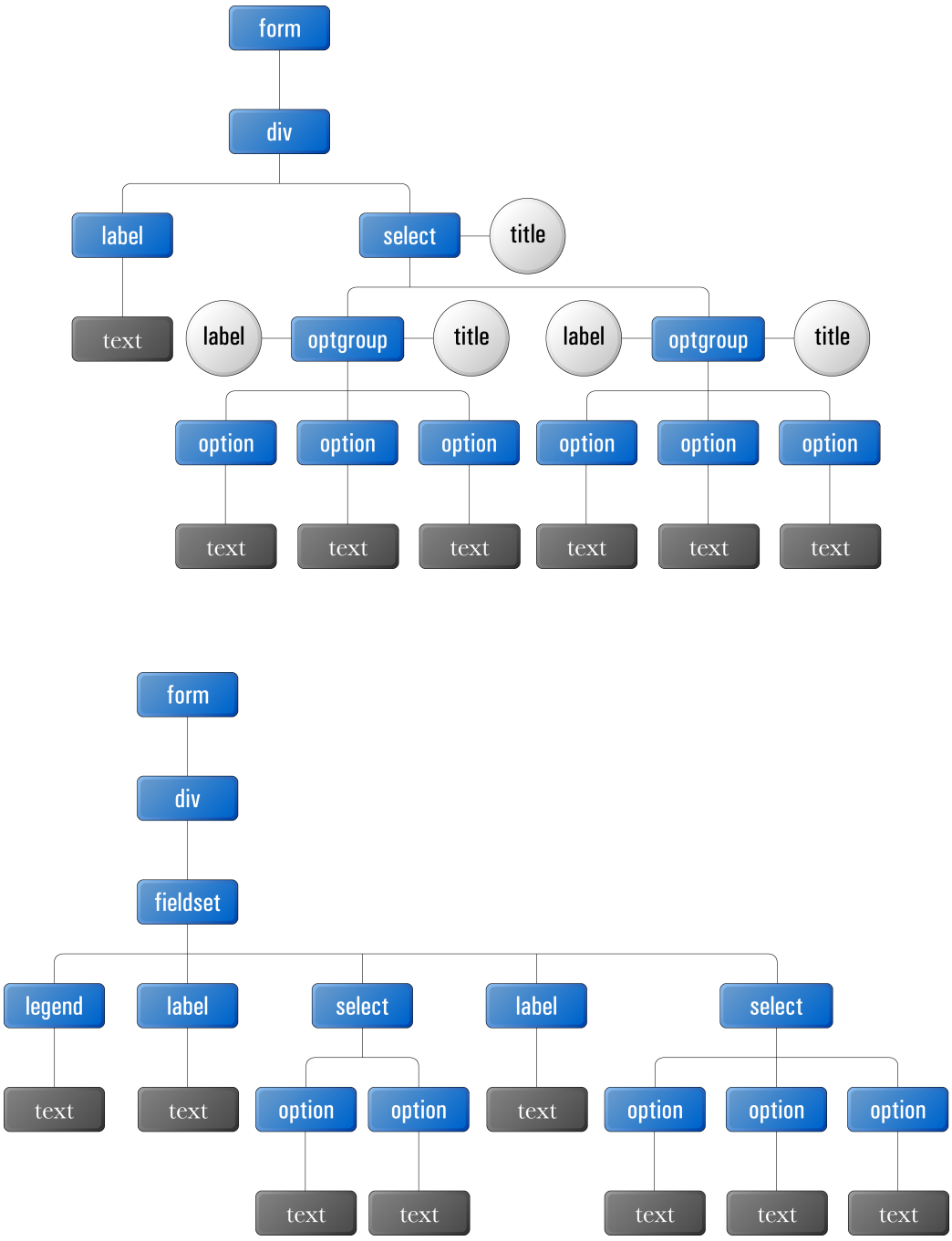


Figure 6.4. Before and after: the required alterations to the DOM structure

cascadingmenu.js *(excerpt)*

```
var CascadingMenu =
{
  init: function()
  {
    var menus = Core.getElementsByClass("cascading");

    for (var i = 0; i < menus.length; i++)
    {
      CascadingMenu.convertLabelToFieldset(menus[i]);
      var masterMenu = CascadingMenu.extractMasterMenu(menus[i]);
      CascadingMenu.createLabelFromTitle(masterMenu);
      CascadingMenu.createLabelFromTitle(menus[i]);

      CascadingMenu.updateSlaveMenu(masterMenu);
      Core.addEventListener(
          masterMenu, "change", CascadingMenu.changeListener);
    }
  },
  ⋮
};

Core.start(CascadingMenu);
```

The rest of the work simply consists of implementing the methods in this code: `convertLabelToFieldset`, `extractMasterMenu`, `createLabelFromTitle`, `updateSlaveMenu`, and `changeListener`. No sweat, right?

Let's start with an easy one: `convertLabelToFieldset` needs to find the label for the `select` element that it's given, and convert it into a `fieldset` and `legend` that will surround the `select`:

cascadingmenu.js *(excerpt)*

```
convertLabelToFieldset: function(menu)
{
  var menuId = menu.id;
  var labels = document.getElementsByTagName("label");

  for (var i = 0; i < labels.length; i++) 1
  {
```

```
    if (labels[i].getAttribute("for") == menuId) 2
    {
      var label = labels[i];
      label.parentNode.removeChild(label); 3

      var legend = document.createElement("legend"); 4
      while (label.hasChildNodes()) 5
      {
        legend.appendChild(label.firstChild);
      }

      var fieldset = document.createElement("fieldset"); 6
      fieldset.appendChild(legend);

      menu.parentNode.replaceChild(fieldset, menu); 7
      fieldset.appendChild(menu);

      return; 8
    }
  }
},
```

This method demonstrates the process of transplanting DOM nodes from one element to another with a `while` loop and the `hasChildNodes` method. It also shows how to replace one element with another, using the `replaceChild` method. Both these processes are new to us, so let's break the code down:

1. In order to find the `label` that belongs to this field, we fetch a list of all the `label` elements in the document and loop through them with a `for` loop as usual.

2. For each `label`, we compare the `for` attribute to the ID of the specified `select` menu. When we find a match, we'll have found the `label` that we need to convert to a `fieldset` and `legend`.

3. We start by removing the `label` from its parent node, effectively removing it from the document. This also removes it from the node set `labels`, which is why we first created the variable `label` to store a reference to the element.

4. Next, we'll create the `legend` element, into which we need to move all the child nodes of the `label`.

5. Moving DOM nodes from one element to another is surprisingly easy. Simply create a `while` loop that uses the `hasChildNodes` method to check if the source element has any more child nodes, and as long as it does, keeps appending the first of its child nodes to the target element. Since a node can only exist in one location at a time, adding it to the target element automatically removes it from the source element.

6. We can now create the `fieldset` element, and add to it the `legend` that we've just created

7. Replacing the `select` menu with the newly created `fieldset` is easy, thanks to the `replaceChild` method, which will insert the `fieldset` in place of the `select` in the `select`'s parent node. Once that's done, we can insert the `select` into the `fieldset`.

8. Since we've found and dealt with the `label` we were looking for, we can `return` from the `convertLabelToFieldset` method immediately, rather than letting the `for` loop continue searching through the document.

Next up is the `extractMasterMenu` method, which will create and return a new `select` menu that contains as `options` the `optgroups` from the original menu:

cascadingmenu.js *(excerpt)*

```
extractMasterMenu: function(menu)
{
  var masterMenu = document.createElement("select"); (1)
  masterMenu.id = menu.id + "master";
  masterMenu.setAttribute("name", masterMenu.id);
  masterMenu.setAttribute("title", menu.getAttribute("title"));
  masterMenu._slave = menu; (2)

  while (menu.hasChildNodes()) (3)
  {
    var optgroup = menu.firstChild;
    if (optgroup.nodeType == 1) (4)
    {
      var masterOption = document.createElement("option"); (5)
```

```
      masterOption.appendChild(document.createTextNode(
          optgroup.getAttribute("label")));
      masterMenu.appendChild(masterOption);

      var slaveOptions = []; ❻
      while (optgroup.hasChildNodes())
      {
        var option = optgroup.firstChild;
        slaveOptions[slaveOptions.length] = option;
        optgroup.removeChild(option);
      }
      masterOption._slaveOptions = slaveOptions;

      menu.setAttribute("title",
          optgroup.getAttribute("title")); ❼
    }
    menu.removeChild(optgroup); ❽
  }

  menu.parentNode.insertBefore(masterMenu, menu); ❾

  return masterMenu; ❿
},
```

It's a doozy, but there's almost nothing in there that you haven't seen before. Nevertheless, let me break down the major steps of this method:

❶ We start by creating a new `select` element for the new menu. We set its `id` property and `name` attribute to match the ID of the existing `select`, but we append the word "master" to it. We grab the `title` directly from the existing `select`.

❷ Since we'll need to update the contents of the existing "slave" menu every time the user makes a selection from the new master menu, we store a reference to the slave in a custom property of the master, called `_slave`.

❸ Now we need to create in the master menu an `option` for each `optgroup` in the slave menu. As we did in the `convertLabelToFieldset` method, we'll use a `while` loop with the `hasChildNodes` method, grabbing the first `optgroup` from

the slave menu, then removing it from that menu in preparation for the next time we cycle through the loop.

4. Since the menu's child nodes may include whitespace text nodes, we need to check the `nodeType` of each child to make sure we're dealing with one of the `optgroup` elements.

5. For each `optgroup` in the slave menu, we'll create an `option` that we'll insert into the master menu. The text for each `option` will be taken from the `label` attribute of the corresponding `optgroup`.

6. So that we can fill the slave menu with the `options` that correspond to the selection in the master menu, we'll bundle up the relevant `option` elements into an array, which we'll store in a custom `_slaveOptions` property of the `option` element within the master menu.

7. As we process each `optgroup`, we grab its `title` attribute and set the `title` of the slave menu to that value. We assume here that each `optgroup` has the same `title`, so the fact that the final `title` of the slave menu will be taken from the last `optgroup` that's processed isn't a problem.

8. Finally, we remove the `optgroup` from the slave menu in preparation for the next cycle of the `while` loop.

9. Now that we've filled the master menu with its `options`, we can insert it into the document, right before the existing menu, using the `insertBefore` method—yet another useful DOM method that we haven't seen before. Like `appendChild`, it adds the first node passed to it as a child of the element upon which it is called. Unlike `appendChild`, however, it doesn't place this child at the end of the element; it places it *before* the second node that's passed to it.

10. At last, we `return` a reference to the new master menu so that `init` can perform additional setup tasks on the menu.

Compared to the last method, `createLabelFromTitle` is dead simple:

cascadingmenu.js *(excerpt)*

```
createLabelFromTitle: function(menu)
{
  var title = menu.getAttribute("title");
  menu.setAttribute("title", "");

  var label = document.createElement("label");
  label.setAttribute("for", menu.id);
  label.appendChild(document.createTextNode(title));

  menu.parentNode.insertBefore(label, menu);
},
```

This method takes the `title` from the `select` element it was given, builds for the `select` a `label` element that contains that text, then inserts it into the document before the `select`. There's nothing to it!

`updateSlaveMenu`, which fills the slave menu with the `options` corresponding to the current master menu selection is almost as straightforward:

cascadingmenu.js *(excerpt)*

```
updateSlaveMenu: function(masterMenu)
{
  var selectedOption =
      masterMenu.options[masterMenu.selectedIndex]; (1)

  while (masterMenu._slave.hasChildNodes()) (2)
  {
    masterMenu._slave.removeChild(masterMenu._slave.firstChild);
  }

  for (var i = 0; i < selectedOption._slaveOptions.length; i++) (3)
  {
    masterMenu._slave.appendChild(
        selectedOption._slaveOptions[i]);
  }
  masterMenu._slave.selectedIndex = 0; (4)
},
```

This method does what you'd expect:

1. It finds the selected `option` in the master menu using its `selectedIndex` property.

2. Using a `while` loop with `hasChildNodes`, it empties the slave menu, which it finds using the custom `_slave` property that we created for the master menu in `extractMasterMenu`.

3. It inserts into the slave menu the array of `options` that corresponds to the selected `option` in the master menu. Again, this task is made simple by the custom `_slaveOptions` property that we set up in `extractMasterMenu`.

4. Finally, it sets the slave menu's `selectedIndex` property to `0`, which selects the first `option` in the list.

As usual, since we're doing all the work in our workhorse methods, our event listener is as simple as they come:

cascadingmenu.js *(excerpt)*

```
changeListener: function(event)
{
  CascadingMenu.updateSlaveMenu(this);
}
```

That does it! Fire the example up in your browser and take it for a spin. Also, be sure to try disabling JavaScript to check that the single menu works just fine in that environment. Here's the completed script:

cascadingmenu.js

```
var CascadingMenu =
{
  init: function()
  {
    var menus = Core.getElementsByClass("cascading");

    for (var i = 0; i < menus.length; i++)
    {
      CascadingMenu.convertLabelToFieldset(menus[i]);
      var masterMenu = CascadingMenu.extractMasterMenu(menus[i]);
      CascadingMenu.createLabelFromTitle(masterMenu);
```

```
    CascadingMenu.createLabelFromTitle(menus[i]);

    CascadingMenu.updateSlaveMenu(masterMenu);
    Core.addEventListener(
        masterMenu, "change", CascadingMenu.changeListener);
  }
},

convertLabelToFieldset: function(menu)
{
  var menuId = menu.id;
  var labels = document.getElementsByTagName("label");

  for (var i = 0; i < labels.length; i++)
  {
    if (labels[i].getAttribute("for") == menuId)
    {
      var label = labels[i];
      label.parentNode.removeChild(label);

      var legend = document.createElement("legend");
      while (label.hasChildNodes())
      {
        legend.appendChild(label.firstChild);
      }

      var fieldset = document.createElement("fieldset");
      fieldset.appendChild(legend);

      menu.parentNode.replaceChild(fieldset, menu);
      fieldset.appendChild(menu);

      return;
    }
  }
},

extractMasterMenu: function(menu)
{
  var masterMenu = document.createElement("select");
  masterMenu.id = menu.id + "master";
  masterMenu.setAttribute("name", masterMenu.id);
  masterMenu.setAttribute(
      "title", menu.getAttribute("title"));
```

```
    masterMenu._slave = menu;

    while (menu.hasChildNodes())
    {
      var optgroup = menu.firstChild;
      if (optgroup.nodeType == 1)
      {
        var masterOption = document.createElement("option");
        masterOption.appendChild(document.createTextNode(
            optgroup.getAttribute("label")));
        masterMenu.appendChild(masterOption);

        var slaveOptions = [];
        while (optgroup.hasChildNodes())
        {
          var option = optgroup.firstChild;
          slaveOptions[slaveOptions.length] = option;
          optgroup.removeChild(option);
        }
        masterOption._slaveOptions = slaveOptions;

        menu.setAttribute("title",
            optgroup.getAttribute("title"));
      }
      menu.removeChild(optgroup);
    }

    menu.parentNode.insertBefore(masterMenu, menu);

    return masterMenu;
  },

  createLabelFromTitle: function(menu)
  {
    var title = menu.getAttribute("title");
    menu.setAttribute("title", "");

    var label = document.createElement("label");
    label.setAttribute("for", menu.id);
    label.appendChild(document.createTextNode(title));

    menu.parentNode.insertBefore(label, menu);
  },
```

```
  updateSlaveMenu: function(masterMenu)
  {
    var selectedOption =
        masterMenu.options[masterMenu.selectedIndex];

    while (masterMenu._slave.hasChildNodes())
    {
      masterMenu._slave.removeChild(masterMenu._slave.firstChild);
    }

    for (var i = 0; i < selectedOption._slaveOptions.length; i++)
    {
      masterMenu._slave.appendChild(
          selectedOption._slaveOptions[i]);
    }
    masterMenu._slave.selectedIndex = 0;
  },

  changeListener: function(event)
  {
    CascadingMenu.updateSlaveMenu(this);
  }
};

Core.start(CascadingMenu);
```

Form Validation

Usability tweaks like those we've seen so far in this chapter are all well and good, but by far the most common use of JavaScript when dealing with forms is client-side validation.

Now let me make one thing clear: every web site that accepts user input of any kind needs a program on the receiving end—the server side—to make sure that input is provided in the expected format, and is safe to use. That's **server-side validation**, and it's an absolute must, no matter what you do with JavaScript in the browser.

Client-side validation, on the other hand, is merely an early warning system. It does the same job as server-side validation—and does it much faster—but you can't rely on it to work every time.

Client-side validation takes place in the browser, before the user input is submitted for processing. The advantage here is that the user doesn't have to wait for the request to be transferred to the server, for validation to occur there, and for the response to come back in the form of a new page. With JavaScript, you can tell the user more or less instantly if there is something wrong with the values that are about to be submitted.

Of course, the user can always disable JavaScript to circumvent your client-side validation, so it's important that you also have server-side validation in place to intercept any invalid submissions.

Intercepting Form Submissions

The key to client-side validation is the `submit` event that I mentioned in Table 6.3. Whenever the user attempts to submit a form (whether by clicking a submit button or just by hitting **Enter** in a text field), a `submit` event is triggered on the corresponding `form` element. The default action for that event is to submit the form, but as we learned in Chapter 4, you can use JavaScript to cancel the default action for an event if you want to.

The basic technique for client-side validation, then, is to set up either an event handler or an event listener for the form's `submit` event, and cancel the default action if the form's contents are not acceptable. Here's roughly what this process looks like, using an event handler:

```
form.onsubmit = function()
{
  if (form input is not valid)
  {
    notify the user
    return false;
  }
};
```

Here's how it looks if you're using an event listener:

```
Core.addEventListener(form, "submit", function(event)
{
  if (form input is not valid)
```

```
  {
    notify the user
    Core.preventDefault(event);
  }
});
```

As usual, I recommend that you stick with using an event listener so that you can enjoy the benefits I described in Chapter 4, but remember that Safari versions 2.0.3 and earlier do not support the cancelling of default actions from event listeners. If you need to support client-side validation in those older Safari versions, look in the code archive for the JavaScript files whose names end in **-dom0.js**. These use event handlers instead of event listeners for each of the examples that follow.

As a trivial example, we could verify that the user had filled in a value for a particular text field:

requiredfield.js *(excerpt)*

```
var RequiredField =
{
  init: function()
  {
    var requiredField = document.getElementById("requiredfield");
    var theForm = requiredField.form; (1)

    Core.addEventListener(
        theForm, "submit", RequiredField.submitListener);
  },

  submitListener: function(event)
  {
    var requiredField = document.getElementById("requiredfield");

    if (requiredField.value == "") (2)
    {
      requiredField.focus(); (3)
      alert("Please fill in a value for this required field."); (4)
      Core.preventDefault(event); (5)
    }
  }
```

```
};

Core.start(RequiredField);
```

Most of this code should come pretty naturally to you by now, but here's a run-down of the highlights:

1. Because the `submit` event's target is the `form` element, not any single form control, we need to use the required field's `form` property to obtain a reference to the form that contains it before we can register our event listener.
2. We can check the value of a text field with its `value` property.
3. Once we've identified a problem with a field, we call its `focus` method to give it keyboard focus and help the user find and correct the mistake.
4. We display a helpful message to let the user know what he or she did wrong, as shown in Figure 6.5.
5. We prevent the form from submitting by cancelling the `submit` event's default action.

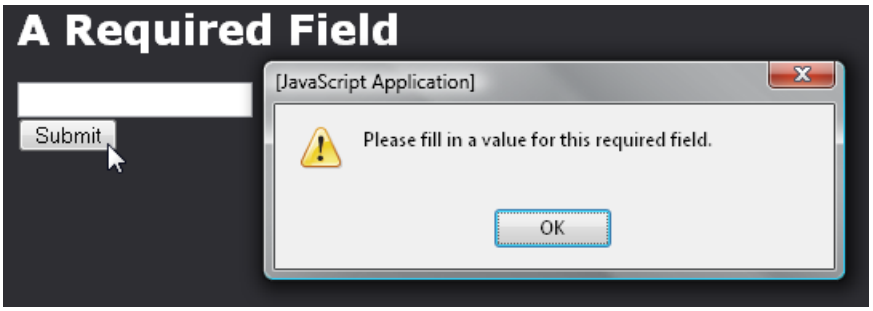


Figure 6.5. A helpful error message

Of course, sometimes it's not enough for a field just to be filled in; sometimes, the value has to conform to a particular format. That's where regular expressions come in handy.

Regular Expressions

We use **regular expressions** to search for and replace patterns of text. They're available in many different programming languages and environments, and are especially prevalent in web development languages like JavaScript.

The popularity of regular expressions has everything to do with how useful they are, and absolutely nothing to do with how easy they are to use—they're not easy at all. In fact, to most people who encounter them for the first time, regular expressions look like something that might eventuate if you fell asleep with your face on the keyboard.

Here, for example, is a relatively simple (yes, really!) regular expression that will match any string that might be a valid email address:

```
^[\w\.\-]+@([\w\-]+\.)+[a-zA-Z]+$
```

Scary, huh? By the end of this section, you'll actually be able to make sense of that.

To create a regular expression in JavaScript, use forward slashes (/) around the expression as you would use single or double quotes around a string:

```
var regex = /pattern/;
```

Escape the Forward Slash!

To include a forward slash as part of a regular expression, you must escape it with a preceding backslash (\/); otherwise, it will be interpreted as marking the end of the pattern.

Alternatively, if you need to build a regular expression dynamically (say, using the value of a variable), you can use this alternative syntax to create the regular expression from a string:

```
var regex = new RegExp("pat" + variable + "tern");
```

Essentially, a regular expression represents a pattern specified with ordinary and special characters. For instance, if you wanted a pattern that matched the string "JavaScript," your regular expression pattern could be:

```
JavaScript
```

However, by including special characters, your pattern could also be:

```
^Java.*
```

The caret (`^`), the dot (`.`), and the asterisk (`*`) are all special characters that have a specific meaning inside a regular expression. Specifically, the caret means "the start of the string," the dot means "any character," and the asterisk means "zero or more of the preceding character."

Therefore, the pattern `^Java.*` matches not only the string "JavaScript," but "Javascript," "JavaHouse," "Java, the most populous island in the world," and any other string beginning with "Java."

Here are some of the most commonly used regular expression special characters (try not to lose too much sleep attempting to memorize these):

`.` (dot) — This is the wildcard character. It matches any single character except line break characters (`\r` and `\n`).

`*` (asterisk) — An asterisk requires that the preceding character appear zero or more times.

When matching, the asterisk will be **greedy**, including as many characters as possible. For example, for the string "a word here, a word there," the pattern `"a.*word"` will match "a word here, a word." In order to make a minimal match (just "a word"), use the question mark character (explained below).

`+` (plus) — This character requires that the preceding character appears one or more times. When matching, the plus will be greedy (just like the asterisk, described above)

unless you use the question mark character (explained below).

? (question mark) This character makes the preceding character optional.

If placed after a plus or an asterisk, it instead dictates that the match for this preceding symbol will be a **minimal match**, including as few characters as possible.

^ (caret) The caret matches the start of the string. This does not include any characters—it considers merely the position itself.

$ (dollar) A dollar character matches the end of the string. This does not include any characters—it considers merely the position itself.

| (pipe) The pipe causes the regular expression to match either the pattern on the left of the pipe, or the pattern on the right.

(…) (round brackets) Round brackets define a group of characters that must occur together, to which you can then apply a modifier like `*`, `+`, or `?` by placing it after the closing bracket.

You can also refer to a bracketed portion of a regular expression later to obtain the *portion* of the string that it matched.

[…] (square brackets) Square brackets define a **character class**. A character class matches *one* character out of those listed within the square brackets.

A character class can include an explicit list of characters (for instance, `[aqz]`, which is the same as `(a|q|z)`), or a range of characters (such as `[a-z]`, which is the same as `(a|b|c|…|z)`.

A character class can also be defined so that it matches one character that's *not* listed in the brackets. To do this, simply insert a caret (`^`) after the opening square bracket (so `[^a]` will match any single character except "a").

If you want to use one of these special characters as a literal character to be matched by the regular expression pattern, escape it by placing a backslash (`\`) before it (for example, `1\+1=2` will match "1+1=2").

There are also a number of so-called **escape sequences** that will match a character that either is not easily typed, or is a certain type of character:

`\n` This sequence matches a newline character.

`\r` This matches a carriage return character.

`\t` This matches a tab character.

`\s` This sequence matches any whitespace character; it's the same as `[ \n\r\t]`.

`\S` This matches any non-whitespace character, and is the same as `[^ \n\r\t]`.

`\d` This matches any digit; it's the same as `[0-9]`.

`\D` This sequence matches anything but a digit, and is the same as `[^0-9]`.

`\w` This matches any "word" character. It's the same as `[a-zA-Z0-9_]`.

`\W` This sequence matches any "non-word" character, and is the same as `[^a-zA-Z0-9_]`.

`\b` This code is a little special because it doesn't actually match a character. Instead, it matches a word boundary—the start or end of a word.

`\B` Like `\b`, this doesn't actually match a character. Rather, it matches a position in the string that is *not* a word boundary.

`\\` Matches an actual backslash character. So if you want to match the string "\n" exactly, your regular expression would be `\\n`, not `\n` (which matches a newline

character). Similarly, if you wanted to match the string "\\" exactly, your regular expression would be `\\\\`.

We now have everything we need to be able to understand the email address regular expression I showed you at the start of this section:

```
^[\w\.\-]+@([\w\-]+\.)+[a-zA-Z]+$
```

`^`	We start by matching the beginning of the string, to make sure that nothing appears before the email address.
`[\w\.\-]+`	The name portion of the email address is made up of one or more (+) characters that are either "word" characters, dots, or hyphens (`[\w\.\-]`).
`@`	The name is followed by the @ character.
`([\w\-]+\.)+`	Then we have one or more (+) subdomains (such as "sitepoint."), each of which is one or more "word" characters or hyphens (`[\w\-]+`) followed by a dot (`\.`).
`[a-zA-Z]+`	Next, there's the top-level domain (for example, "com"), which is simply one or more letters (`[a-zA-Z]+`).
`$`	Finally, we match the end of the string, to make sure that nothing appears after the email address.

Got all that? If you're feeling anything like I was when I first learned regular expressions, you're probably a little nervous. Okay, you can follow along with a breakdown of a regular expression that someone else wrote for you, but can you really come up with this gobbledygook yourself? Don't sweat it: in the following example, we'll look at a bunch more regular expressions, and before you know it you'll be writing expressions of your own with confidence.

But hang on a minute … I've told you all about the syntax of regular expressions, but we still need to see how to actually use them in JavaScript! In fact, there are a number of different ways to do this, but the simplest—and the one we'll be using the most in this book—is the `test` method that's supported by all regular expressions:

```
regex.test(string)
```

The `test` method will return true if the regular expression it's called on matches the string that you pass to it as an argument.

For a simple example of this method in action, we can return to the `hasClass` method that we created back in Chapter 3:

core.js *(excerpt)*

```
Core.hasClass = function(target, theClass)
{
  var pattern = new RegExp("(^| )" + theClass + "( |$)"); (1)

  if (pattern.test(target.className)) (2)
  {
    return true;
  }

  return false;
};
```

This method determines if a particular class appears in the `class` attribute of a given HTML element. Remember that this attribute is a space-delimited list of classes (for example, `classa classb classc`).

(1) First, this method builds a regular expression containing the class name that we're searching for, of the form `(^| )`*`theClass`*`( |$)`. This expression should be fairly easy for you to decipher by now: it starts by matching either the start of the string or a space (`(^| )`), then the class name, and finally, either a space or the end of the string (`( |$)`).

(2) Next, the method tests if the value of the element's `class` attribute (`target.className`) matches the pattern. If it does, the method returns `true`. Otherwise, it returns `false`.

Not too bad, eh? Let's dive into a bigger example and see how we go. In the meantime, if you want a more extensive listing of regular expression syntax, visit regu-

larexpressions.info,[2] and for more information on regular expressions in JavaScript, see my article *Regular Expressions in JavaScript* on SitePoint.[3]

Example: Reusable Validation Script

Before we got distracted by the bewildering glory of regular expressions, I seem to remember this chapter being about forms. How about we try to combine the two by using regular expressions to validate forms?

Regular expressions are great for validating forms, because we can express quite a complex set of requirements for a field in just a single regular expression. For example, we can test if a field has been filled in with a very simple expression:

```
.
```

That right, it's just a dot (`.`). Since this will match any character, the only string that won't match this is the empty string (`""`).

Of course, a user could just type a space character into the field and satisfy this pattern, so how about we require at least one non-whitespace character in the field? Again, this is an easy expression to write:

```
\S
```

Let's make things a little more complicated. How about a field that requires a positive whole number, and nothing else?

```
^\d*[1-9]\d*$
```

This pattern checks that the string contains at least one digit between 1 and 9 (that is, not 0), which can be preceded and/or followed by any number of digits (including 0). The start-string (`^`) and end-string (`$`) characters ensure there is nothing else in the string.

[2] http://www.regularexpressions.info/
[3] http://www.sitepoint.com/article/expressions-javascript/

If we were happy to allow zero as a value, we could simplify the pattern to this:

```
^\d+$
```

Want to allow negative numbers too? Easy! Just add an optional minus sign:

```
^-?\d+$
```

How about decimal numbers like `12.345`? Easy enough:

```
^-?\d+(\.\d+)?$
```

This pattern allows for an optional decimal point (`\.`) followed by one or more digits.

Let's get a little trickier and check for a valid phone number. Phone number formats vary from country to country, but in general they consist of an optional country code (`+1` for North America, or `+61` for Australia), an optional area code that may or may not be enclosed in parentheses, then one or more groups of numbers. Each of these elements may be separated by spaces, dashes, or nothing at all.

Here's the expression we'll need:

```
^(\+\d+)?( |\-)?(\(?\d+\)?)?( |\-)?(\d+( |\-)?)*\d+$
```

If you can get that expression straight in your head, you can safely claim to have mastered the basics of regular expressions. Good luck! For an added challenge, try adapting the pattern to allow phone numbers that are spelled out in letters, such as "1-800-YOU-RULE."

To build a reusable form validation script, we can bundle all of the regular expressions we've seen so far into a JavaScript object:

formvalidation.js *(excerpt)*

```
var FormValidation =
{
  ⋮
  rules:
  {
```

```
    required: /./,
    requiredNotWhitespace: /\S/,
    positiveInteger: /^\d*[1-9]\d*$/,
    positiveOrZeroInteger: /^\d+$/,
    integer: /^-?\d+$/,
    decimal: /^-?\d+(\.\d+)?$/,
    email: /^[\w\.\-]+@([\w\-]+\.)+[a-zA-Z]+$/,
    telephone:
        /^(\+\d+)?( |\-)?(\(?\d+\)?)?( |\-)?(\d+( |\-)?)*\d+$/
  },
```

For each pattern, we can also store an error message for display when the pattern is not satisfied:

formvalidation.js *(excerpt)*

```
errors:
{
  required: "Please fill in this required field.",
  requiredNotWhitespace: "Please fill in this required field.",
  positiveInteger:
      "This field may only contain a positive whole number.",
  positiveOrZeroInteger: "This field may only contain a " +
      "non-negative whole number.",
  integer: "This field may only contain a whole number.",
  decimal: "This field may only contain a number.",
  email: "Please enter a valid email address into this field.",
  telephone:
      "Please enter a valid telephone number into this field."
},
```

We can then mark the fields in our form with the labels we defined in the code above. For example, a required field would have its `class` set to `required`:

formvalidation.html *(excerpt)*

```
<input type="text" class="required" id="username" name="username"
    />
```

Similarly, here's a field that must contain a valid email address:

formvalidation.html *(excerpt)*

```
<input type="text" class="email" id="email" name="email" />
```

All that's left is to write the JavaScript required to test the fields of a form with the specified regular expressions before allowing submission to proceed.

The `init` method is simple enough. All we need to do is add a `submit` event listener to every form on the page:

formvalidation.js *(excerpt)*

```
init: function()
{
  var forms = document.getElementsByTagName("form");

  for (var i = 0; i < forms.length; i++)
  {
    Core.addEventListener(
        forms[i], "submit", FormValidation.submitListener);
  }
},
```

In fact, the only tricky part of this script is in that `submit` event listener:

formvalidation.js *(excerpt)*

```
submitListener: function(event)
{
  var fields = this.elements; (1)

  for (var i = 0; i < fields.length; i++)
  {
    var className = fields[i].className; (2)
    var classRegExp = /(^| )(\S+)( |$)/g; (3)
    var classResult;

    while (classResult = classRegExp.exec(className)) (4)
    {
```

```
      var oneClass = classResult[2]; 5
      var rule = FormValidation.rules[oneClass]; 6
      if (typeof rule != "undefined") 7
      {
        if (!rule.test(fields[i].value)) 8
        {
          fields[i].focus(); 9
          alert(FormValidation.errors[oneClass]); 10
          Core.preventDefault(event); 11
          return; 12
        }
      }
    }
  }
}
```

Thanks to the power of regular expressions, that's all the code we need to write in order to validate a form—any form—with the regular expressions that we declared earlier. Thanks to the *complexity* of regular expressions, this code is particularly difficult to understand at a glance. Let me walk you through it:

1. First, we obtain a list of all the fields in the form, using the `elements` property of the `form` element described in Table 6.2. We can then loop through the list with a `for` loop.

2. For each field, we obtain the `class` attribute value, which we'll use to ascertain which of the regular expressions we'll test its value against. But remember that the class attribute can contain a list of *multiple* class names, so we need to do some work to break this value up into individual class names, and check them one at a time.

3. This regular expression will find one particular class name in a space-delimited list such as the `class` attribute value. It looks for the start of the string or a space, followed by one or more non-whitespace characters, and then a space or the end of the string. But, since we want to find not just *one* class name, but *all* of the class names in the attribute value, we need to apply the **global modifier** to this expression. See the `g` following the closing forward slash that marks the end of the regular expression? That's the global modifier, and it sets the

regular expression up so that we can use it to find *all* the matches in a given string, not just the first one.

4. Most of the time, the `test` method is all that's needed to use a regular expression. But when you're working with the global modifier, you're better off using a more advanced method: `exec`. This method works just like `test`, except that it returns information about the portion of the string that it matched. When `exec` is applied to a regular expression that includes the global modifier, and it finds a match, it returns an array, which we store in the `classResult` variable.

5. The first element of that array (with array index `0`) contains the portion of the string that was matched by the entire regular expression; in this case, the portion of the string may contain spaces before or after the class name, so it's no good to us. The second element in the array (array index `1`) contains the portion of the string that was matched by the first parenthesized portion of the regular expression (in this case, the space before the class name, if any). The third element in the array (array index `2`) contains the portion of the string matched by the second parenthesized portion of the expression, which in this case is the actual class name—exactly what we're after!

   When called repeatedly, the `exec` method will search for successive matches in the string, which is the reason why we can use it in a `while` loop in this way. During the first cycle of the loop, `classResult[2]` will contain the first class name in the `class` attribute; the second time through, `classResult[2]` will contain the second class name, and so on, until all the class names in the attribute have been found. At this point, `exec` will return `null` and the `while` loop will end.

6. Now that we have an individual class name, we can check if we have a regular expression for validating fields with that class.

   Remember that we've stored all of our validation expressions as properties of the `FormValidation.rules` object. Usually, you access the property of an object using a dot and the property name (for example, `FormValidation.rules.required`), but we can't do that if the name of the property is stored as a string in a variable, as it is now. If we typed `FieldValidation.rules.oneClass`, JavaScript would actually look for a

property named `oneClass`, rather than using the name stored in the `oneClass` variable.

The trick to accessing a property using a property name that's stored in a variable is to treat the object like an array, and use the property name as the array index. So we can fetch the regular expression for a particular class using `FormValidation.rules[oneClass]`. This doesn't exactly make sense, but it's just one of those odd things about JavaScript that you learn from experience.

7. Because we may or may not have a regular expression that corresponds to a given class name, we need to check that we actually got a value back for the `rule` variable.

8. Finally, we can use the `test` method to check if our regular expression matches the value that the user has supplied for the field.

9. If it doesn't match, then the value is not valid, so we assign keyboard focus to the field so that the user can correct it.

10. We then display the error message corresponding to the class name, again using the trick of treating the `FormValidation.errors` object like an array.

11. Because the field's value is invalid, we cancel the default action for the `submit` event, so that the form isn't submitted.

12. And finally, we `return` immediately, so that the user is only notified of one error per submission attempt. If we left this step out, the user would be notified of each and every invalid field in the form one after another, which I think would be a little harsh if, for example, the user had simply submitted the form by accident before filling it in.

And there you have it—a reusable form validation script in just 66 lines of code. Try it out on the simple example page we've included in the code archive, then take it for a spin on some forms of your own! Finally, try adding some more regular expressions (and the corresponding error messages) of your own devising to make this library even *more* useful!

Annoyed by Alerts?

If, like many people, you find alert boxes annoying, you might like to have your validation errors displayed within the page, next to the form fields themselves, instead of in popup message boxes. This technique is demonstrated in the SitePoint book *The JavaScript Anthology* (Melbourne: SitePoint, 2006).

Custom Form Controls

Linking multiple fields together and performing client-side validation are both useful ways to enhance the rather limited form controls that are currently available in HTML, but to truly rise above the limitations of those controls, you need to create entirely new controls of your own!

Creating fully realized user interface elements using only the capabilities of HTML, CSS, and JavaScript is a tall order. The built-in form controls have a lot of subtle features that are difficult, if not impossible to achieve using JavaScript. Consider the right-click menu—not to mention all the keyboard shortcuts—that even a basic text field supports!

However, by setting realistic goals, and by taking advantage of the existing form controls where appropriate, you can produce some truly useful user interface elements. Let's look at one, shall we?

Example: Slider

A slider control can give your users a very intuitive way to select a value over a given range. The control immediately gives the user a sense of the position of the current value within the available range of values, and also allows the user to manipulate that value easily, and see the changes in real time.

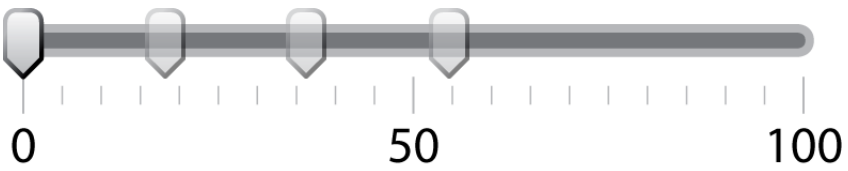


Figure 6.6. A typical slider control

A plain text field can store the same values as a slider, so in order to keep the page accessible to users who browse with JavaScript disabled, we will convert any text input field that has a class of `slider` into a slider control:

slider.html *(excerpt)*

```
<label for="percent">Percentage
  <input id="percent" name="percent" type="text" value="0"
      class="slider from0 to100 scale4" />
</label>
```

Notice the other classes that have been applied to this field: `from0`, `to100`, and `scale4`. These classes control various aspects of the slider control that will be created from this text field:

`from0` The minimum value that you will be able to select using the slider will be `0`.

`to100` The maximum value that you will be able to select using the slider will be `100`.

`scale4` For every change of 1 in the value of the field, the slider will move four pixels.

Together, these classes describe a slider that's 400 pixels wide, and allows you to select values ranging from 0 to 100.

Before we tackle the script that will make the control work, let's figure out the HTML document structures and CSS styles that will get the browser to display the control.

We'll start by creating an image for the "track" of the slider—the labelled horizontal bar that represents the range of values from which the user can choose. To match the values specified in the code, the slider's track should be exactly 400 pixels long, but the image's actual width and height are up to you. For example, the image may be wider than 400 pixels if you want to add decorative elements to either end of the track. Figure 6.7 shows the background image I created, which is 430 pixels wide and 72 pixels high.

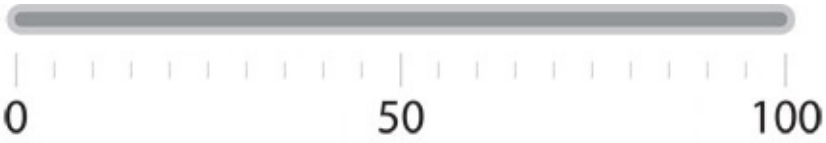


Figure 6.7. The image for the slider track

We also need a separate image for the draggable portion of the slider control—the "thumb." The one I've created is 20 pixels wide and 35 pixels high, and is shown in Figure 6.8.

Figure 6.8. The image for the slider thumb

Now, keeping in mind that we want to keep the text field visible and usable for both keyboard and screen reader users, what do we need in terms of additional DOM structure and CSS styles to produce this control? Well, I'm thinking something like this would work well:

```
<label for="percent">Percentage
  <span class="sliderControl">
    <span class="track"></span>
    <span class="thumb"></span>
    <input id="percent" name="percent" type="text"
        value="0" class="slider from0 to100 scale4" />
  </span>
<label>
```

Remember: this isn't code you'll actually add to your HTML—we'll use JavaScript to generate these new elements.

I've used `span`s because they're legal wherever an `input` is allowed in HTML. The `sliderControl span` provides a container for us to position the different elements of the slider control, the `track span` is what we'll use to display the track image, and the `thumb span` will display the thumb image for us. I've also put the existing `input` inside the `sliderControl span` because that gives us the freedom to position the field in relation to the elements of the slider control.

Here's the CSS that will style all this:

slider.css *(excerpt)*

```
span.sliderControl { 1
  display: block;
  height: 79px;
  position: relative;
}

span.sliderTrack { 2
  background: url(slider_scale.jpg);
  display: block;
  height: 72px;
  left: 0;
  position: absolute;
  top: 7px;
  width: 430px;
}

span.sliderControl span.sliderThumb { 3
  background-image: url(slider_thumb.gif);
  cursor: w-resize;
  height: 35px;
  position: absolute;
  top: 0;
  width: 20px;
}

span.sliderControl input.slider { 4
  margin-left: 430px;
}
```

This isn't a book on CSS layout, but let me quickly sum up what each of these rules does:

1. The `span` that contains all the elements that make up the slider will be displayed as a block, with a height sufficient to accommodate the slider control (specifically, the track background image, and any space you want to allow around it). Setting `position: relative` lets you position all the elements it contains relative to its top-left corner.

2. The `sliderTrack span` is really just a canvas on which to display the slider track as a background image. It's displayed as a block of the required dimen-

sions, and is positioned so that when the thumb is positioned at the left edge of the `sliderControl` container, it appears at the left end of the track.

3. The `sliderThumb span` is similarly set up to adopt the exact dimensions needed to display the thumb image as its background. Notice, however, that it doesn't have a value set for its `left` property, because its horizontal position will be controlled dynamically by our JavaScript code. It does, however, display the horizontal-resize cursor to assist the user in discovering how to use the control.

4. This final rule positions the text input field within the `sliderControl` container. In this case, I've given it sufficient left margin to place it just to the right of the slider control.

I've highlighted in bold the values in this code that are likely to change if you create a slider of your own with different dimensions and images.

Okay, the groundwork is done—let's get scripting! As usual, we start with an `init` method that searches the document for the elements that we want to enhance—in this case, elements of class `slider`:

slider.js *(excerpt)*

```
var Slider =
{
  init: function()
  {
    var sliderFields = Core.getElementsByClass("slider");

    for (var i = 0; i < sliderFields.length; i++)
    {
```

The first thing we need to do to each of these elements is extract the "from," "to," and "scale" values from its `class` attribute. This is, of course, a job for regular expressions:

slider.js *(excerpt)*

```
      var fromMatch = /(^| )from(\d+)( |$)/.exec(
          sliderFields[i].className);
      var from = parseInt(fromMatch[2], 10);
```

```
var toMatch = /(^| )to(\d+)( |$)/.exec(
    sliderFields[i].className);
var to = parseInt(toMatch[2], 10);

var scaleMatch = /(^| )scale(\d+)( |$)/.exec(
    sliderFields[i].className);
var scale = parseInt(scaleMatch[2], 10);
```

Because we're extracting values using regular expressions, we're using the more advanced `exec` method that we learned about in the section called "Example: Reusable Validation Script". In each regular expression, the number that we're interested in extracting is in the second parenthesized section of the expression, which we can grab out of the third element of the array returned by `exec`.

Now, regular expressions deal with strings, and what we actually need are numbers, so we need to convert each string value to a JavaScript number. JavaScript has a built-in function called `parseInt` that does exactly that; we've used it in the code. `parseInt` looks at the start of a string (the first argument), and returns any number it finds there. The second argument specifies the base of the number. Since we usually deal with decimal numbers, which are base `10`, you can make a habit of always passing `10` as this function's second argument.

Now it's time to begin creating the DOM structure we looked at earlier. You should be very used to this sort of thing by now:

slider.js *(excerpt)*

```
var slider = document.createElement("span");
slider.id = sliderFields[i].id + "slider";
slider.className = "sliderControl";

var track = document.createElement("span");
track.id = sliderFields[i].id + "track";
track.className = "sliderTrack";

var thumb = document.createElement("span");
thumb.id = sliderFields[i].id + "thumb";
thumb.className = "sliderThumb";
```

To each of the three `span` elements that we create, we assign both an ID and a class to facilitate styling. The class lets you apply general style properties shared by all sliders on the page, while the ID lets you apply properties specific to an individual slider control.

Since the user will actually interact with the slider thumb (by dragging it), we'll take the opportunity to store all of the values that our event listeners will need in custom properties of the thumb element:

slider.js *(excerpt)*

```
    thumb._input = sliderFields[i];
    thumb._from = from;
    thumb._to = to;
    thumb._scale = scale;
```

The user may still choose to enter values directly into the text field, however, and since we'll need to update the position of the thumb in response to such changes, we'll store a reference to the thumb in the text field element:

slider.js *(excerpt)*

```
    sliderFields[i]._thumb = thumb;
```

With all the elements created, we can now add them to the document to produce the required structure:

slider.js *(excerpt)*

```
    slider.appendChild(track);
    slider.appendChild(thumb);
    sliderFields[i].parentNode.replaceChild(
        slider, sliderFields[i]);
    slider.appendChild(sliderFields[i]);
```

The final step in building the slider is to position the thumb so that it corresponds to the current value of the text field:

slider.js *(excerpt)*

```
      var value = parseInt(sliderFields[i].value, 10);
      thumb.style.left = ((value - from) * scale) + "px";
```

The first line above obtains a number based on the string value of the text field. The second line calculates the difference between that value and the minimum value of the slider, multiplies that difference by the scale of the slider, then positions the thumb that number of pixels from the left-hand side of the slider.

That's our slider created. Now all we need are the event listeners to make it go:

slider.js *(excerpt)*

```
      Core.addEventListener(sliderFields[i], "change",
          Slider.changeListener);
      Core.addEventListener(thumb, "mousedown",
          Slider.mousedownListener);
    }
  },
```

When the user changes the value of the text field, our `change` event listener will update the slider accordingly. Conversely, when the user clicks on the slider thumb, our `mousedown` event listener will handle the dragging operation, and will update the text field's value.

Let's start with the `change` event listener, as we've dealt with this type of event before:

slider.js *(excerpt)*

```
changeListener: function(event)
{
  var thumb = this._thumb; (1)
  var value = parseInt(this.value, 10);

  if (value < thumb._from) (2)
  {
    value = thumb._from;
  }
```

```
  else if (value > thumb._to)
  {
    value = thumb._to;
  }

  thumb.style.left =
      ((value - thumb._from) * thumb._scale) + "px"; 3
  this.value = value; 4
},
```

While the code here is relatively straightforward, let me explain exactly what it's doing:

1. Since we're responding to an event targeted on the text field, we can obtain a reference to the slider thumb using the custom `_thumb` property that we created in `init`.

2. We check the value that was entered by the user against the limits that have been specified for the slider. If the value is too big, we reduce it to the maximum allowed value. If it's too small, we increase it to the minimum allowed value.

3. Once we've settled on an acceptable value, we move the thumb to the corresponding position.

4. Finally, we take the value that we settled on and write it back into the form field, so that any adjustment that occurred due to the limits of the slider are reflected in the field's value.

And now for the part you've been waiting for: the code that makes the slider thumb draggable. Every drag operation involves three kinds of events:

1. A `mousedown` event indicates that the user has pushed down a mouse button while the cursor was positioned over the draggable element.

2. A series of `mousemove` events is generated as the user moves the cursor around the page.

3. A `mouseup` event signals that the user has released the mouse button, completing the drag operation.

We've already registered a `mousedown` event listener on the slider thumb in `init`, so let's take a look at it:

slider.js *(excerpt)*

```
mousedownListener: function(event)
{
  this._valueorigin =
      parseInt(this.style.left, 10) / this._scale - this._from; (1)
  this._dragorigin = event.clientX; (2)
  document._currentThumb = this; (3)

  Core.addEventListener(
      document, "mousemove", Slider.mousemoveListener); (4)
  Core.addEventListener(
      document, "mouseup", Slider.mouseupListener); (5)
  Core.preventDefault(event); (6)
},
```

This listener is responsible for kicking off the drag operation:

(1) We begin by recording a number of values that will be needed throughout the drag operation, the first of which is the value indicated by the slider when the drag began. In subsequent steps, we'll track how far the mouse has moved from its starting position, and use that figure to determine how much the slider's value should change from this starting value.

(2) Obviously, then, we also need to record the mouse pointer's starting position. The `clientX` property of the `event` object for any event gives you the horizontal position of the pointer within the browser window when that event occurred.

(3) As we'll see in a moment, we'll also need a convenient way to access the thumb throughout the drag operation, so we'll store in a custom property of the `document` object a reference to that element.

(4) Now that a drag operation has begun, we need to respond to every `mousemove` event in the document—whether the cursor remains positioned over the thumb element or not. We therefore add a `mousemove` event listener to the `document` node. This listener will catch events that bubble up from any element in the document.

5 Similarly, we need to know when the user releases the mouse button, no matter where the cursor is located when that happens, so we add a `mouseup` event listener to the `document` object.

   Because these two event listeners are registered on the document and not the thumb, they won't be able to access the thumb element using `this`. That's why we recorded the reference to the thumb element in the document we just created.

6 Finally, we prevent the browser from taking any default action—such as beginning a text selection—in response to the mouse button being pressed.

As the user drags the cursor around the page with the mouse button held down, we'll respond to every `mousemove` event by dynamically updating the position of the thumb and the value in the text field:

slider.js *(excerpt)*

```
mousemoveListener: function(event)
{
  var thumb = document._currentThumb; 1
  var value = thumb._valueorigin +
      (event.clientX - thumb._dragorigin) / thumb._scale; 2

  if (value < thumb._from) 3
  {
    value = thumb._from;
  }
  else if (value > thumb._to)
  {
    value = thumb._to;
  }

  thumb.style.left =
      ((value - thumb._from) * thumb._scale) + "px"; 4
  thumb._input.value = value; 5

  Core.preventDefault(event); 6
},
```

This listener works a lot like `changeListener`, which we saw earlier, except that instead of fetching the updated value from a form field, it needs to calculate that value on the basis of the mouse pointer's position in relation to its starting location, and the original value when the drag operation began:

1. We're responding to a `mousemove` event that has bubbled up to the document node, but all the information we need is locked away in the thumb element. Thankfully, we stored a reference to that element in the custom `_currentThumb` property of the `document` object.

2. Here, we calculate the slider value corresponding to the current mouse position. We start with the value of the slider before the drag started (`thumb._valueorigin`), then add to it the difference between the current mouse position (`event.clientX`) and the starting mouse position (`thumb._dragorigin`) divided by the scale of the slider (`thumb._scale`).

3. As with `changeListener`, we limit the value that we just calculated so that it falls within the range allowed by the slider.

4. To produce the illusion that the user is actually dragging the slider thumb, we adjust its position based on the value that we have just calculated. Again, this code works just like the corresponding code in `changeListener`.

5. We also update the value of the text field on the fly.

6. Finally, we prevent the browser from doing anything it might normally do in response to the movement of the mouse.

As the French might say, *le tour est joué!*—we've pulled it off! The user can now click and drag the slider, and see the field's value update in real time. All that's left is to complete the drag operation when the user releases the mouse button:

slider.js *(excerpt)*

```
mouseupListener: function(event)
{
  document._currentThumb = null;
  Core.removeEventListener(document, "mousemove",
      Slider.mousemoveListener);
```

```
  Core.removeEventListener(document, "mouseup",
      Slider.mouseupListener);
}
```

This code is pretty self-explanatory. It removes the reference to the thumb that we stored in the `document` object, and it removes the `mousemove` and `mouseup` listeners, leaving the `mousedown` listener in place to start the next drag.

As you can probably tell, creating draggable elements on the Web is a bit of a black art, and things get even more complicated when you want to allow the user to *drop* the draggable element on one or more target elements.[4]

For now, however, we've managed to make dragging an element seem natural with an intricate mesh of listeners that finely control how the browser responds to the relevant mouse events. This demonstrates just how much you can achieve with JavaScript if you're willing to take matters into your own hands. Figure 6.9 shows what the finished slider control should look like.

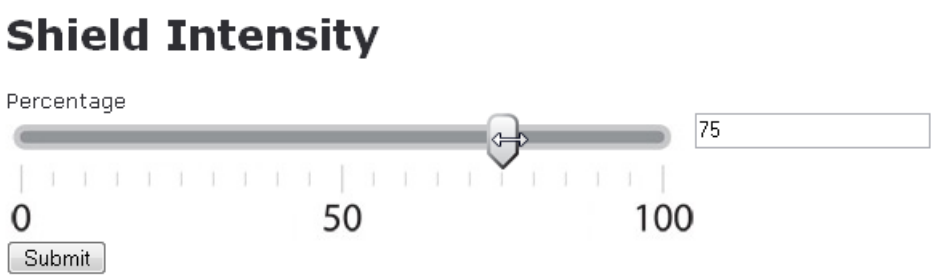


Figure 6.9. The finished slider control

Here's the complete JavaScript code for the slider control. Considering what it can add to the usability of a form, I'd say it's refreshingly brief!

[4] This is covered at great length in the SitePoint book *The JavaScript Anthology* (Melbourne: SitePoint, 2006).

slider.js

```
var Slider =
{
  init: function()
  {
    var sliderFields = Core.getElementsByClass("slider");

    for (var i = 0; i < sliderFields.length; i++)
    {
      var fromMatch = /(^| )from(\d+)( |$)/.exec(
          sliderFields[i].className);
      var from = parseInt(fromMatch[2], 10);

      var toMatch = /(^| )to(\d+)( |$)/.exec(
          sliderFields[i].className);
      var to = parseInt(toMatch[2], 10);

      var scaleMatch = /(^| )scale(\d+)( |$)/.exec(
          sliderFields[i].className);
      var scale = parseInt(scaleMatch[2], 10);

      var slider = document.createElement("span");
      slider.id = sliderFields[i].id + "slider";
      slider.className = "sliderControl";

      var track = document.createElement("span");
      track.id = sliderFields[i].id + "track";
      track.className = "sliderTrack";

      var thumb = document.createElement("span");
      thumb.id = sliderFields[i].id + "thumb";
      thumb.className = "sliderThumb";
      thumb._input = sliderFields[i];
      thumb._from = from;
      thumb._to = to;
      thumb._scale = scale;

      sliderFields[i]._thumb = thumb;

      slider.appendChild(track);
      slider.appendChild(thumb);
      sliderFields[i].parentNode.replaceChild(
          slider, sliderFields[i]);
```

```
    slider.appendChild(sliderFields[i]);

    var value = parseInt(sliderFields[i].value, 10);
    thumb.style.left = ((value - from) * scale) + "px";

    Core.addEventListener(
        sliderFields[i], "change", Slider.changeListener);
    Core.addEventListener(
        thumb, "mousedown", Slider.mousedownListener);
  }
},

changeListener: function(event)
{
  var thumb = this._thumb;
  var value = parseInt(this.value, 10);

  if (value < thumb._from)
  {
    value = thumb._from;
  }
  else if (value > thumb._to)
  {
    value = thumb._to;
  }

  thumb.style.left =
      ((value - thumb._from) * thumb._scale) + "px";
  this.value = value;
},

mousedownListener: function(event)
{
  this._valueorigin =
      parseInt(this.style.left, 10) / this._scale - this._from;
  this._dragorigin = event.clientX;
  document._currentThumb = this;

  Core.addEventListener(
      document, "mousemove", Slider.mousemoveListener);
  Core.addEventListener(
      document, "mouseup", Slider.mouseupListener);
  Core.preventDefault(event);
},
```

```
  mousemoveListener: function(event)
  {
    var thumb = document._currentThumb;
    var value = thumb._valueorigin +
        (event.clientX - thumb._dragorigin) / thumb._scale;

    if (value < thumb._from)
    {
      value = thumb._from;
    }
    else if (value > thumb._to)
    {
      value = thumb._to;
    }

    thumb.style.left =
        ((value - thumb._from) * thumb._scale) + "px";
    thumb._input.value = value;

    Core.preventDefault(event);
  },

  mouseupListener: function(event)
  {
    document._currentThumb = null;
    Core.removeEventListener(
        document, "mousemove", Slider.mousemoveListener);
    Core.removeEventListener(
        document, "mouseup", Slider.mouseupListener);
  }
};

Core.start(Slider);
```

Exploring Libraries

A little surprisingly, none of the major JavaScript libraries currently does the kinds of simple form enhancements that we looked at earlier in this chapter—dependent fields, cascading menus, and so on. However, many of these libraries *do* have features that can make building these sorts of enhancements yourself a little easier.

That said, the two other types of form enhancements that we've looked at in this chapter—client-side validation and custom controls—are well served by one library or another.

Form Validation

One of the Dojo library's main strengths is its library of widgets, many of which are enhanced versions of the basic HTML form controls. Some of these widgets provide built-in validation features. For example, with Dojo loaded, you can create a text field that only accepts whole numbers (integers) like this:

```
<input type="text" name="name" class="dojo-IntegerTextbox" />
```

This code invokes Dojo's `IntegerTextbox` widget, which automatically displays an error message when the user types anything that isn't a whole number into the field.

As of this writing, however, Dojo's validation widgets aren't well documented, and the requirement that all the fields of a form are valid before the user is allowed to submit the form still necessitates your writing some JavaScript code of your own. In short, figuring out just how to use Dojo to automate your form validation can take a lot longer than writing your own validation library, as we did in this chapter.

That said, if you're willing to look beyond the major JavaScript libraries, you'll find a couple of well-developed mini-libraries that do client-side validation very well indeed.

wForms is an actively-developed library that works very much like the reusable form validation script that we developed in this chapter.[5] Like our library, it applies reusable validation rules to your form fields based on the `class` attribute. Here's a list of the class names that it recognizes, and the corresponding checks that it carries out:

`required` — The field cannot be left empty. By setting this class name on an element that contains a number of form fields, you can require the user to fill in *at least one* of those fields.

[5] http://www.formassembly.com/wForms/

`validate-alpha`	This class allows only alphabetic characters in the value of the field.
`validate-alphanum`	This class allows only numbers and alphabetic characters.
`validate-date`	This class allows a date to be entered in any format that's recognized by the browser.
`validate-email`	This class allows an email address, or several email addresses separated by commas, spaces, or semicolons.
`validate-integer`	This class allows only a whole number.
`validate-float`	This class allows only decimal numbers.
`validate-custom /`*`regex`*`/`	This class allows any value that matches the specified regular expression.
`allrequired`	By setting this class name on an element that contains a number of form fields, you can require that *all* of the fields be filled in.

Forms also offers extensive control over the error messages that are displayed when validation fails. Error messages can appear in an alert box, within the page, or both. This library is full of all sorts of other useful features, including support for dependent fields that's similar to what we developed at the start of this chapter. wForms calls them "conditional sections."

Another library that's worth checking out is Really Easy Field Validation, which was developed by a fellow named Andrew Tetlaw. It works very much like wForms, except that it's based on the Prototype JavaScript library (so you need to load Prototype, and then this library, on your page). You can read all about—and download—Really Easy Field Validation at the developer's web site.[6]

[6] http://tetlaw.id.au/view/javascript/really-easy-field-validation

Custom Controls

As I mentioned earlier, one of the main areas of focus for the Dojo library has been the creation of rich widgets to supplement the user interface elements that plain HTML has to offer. Among these is an extensive collection of Form Widgets that work in the same way as the slider control that we developed in this chapter, providing a rich, JavaScript-powered user interface that sits atop a standard HTML form field.

Here's a quick list of some of Dojo's Form Widgets:

Button	an advanced version of the HTML `button` element
Checkbox	like a standard HTML checkbox, but uses customizable images for the checkbox
ColorPalette	displays a grid of color swatches from which the user can choose
ComboBox	like a standard text field, but pops up a list of suggested values in response to user input
DatePicker	displays an interactive calendar for selecting a date
TimePicker	displays a rich interface for selecting a time of day
Editor2 (RichText)	a WYSIWYG HTML editor that lets the user input and format rich text that's submitted as HTML
HslColorPicker	displays a rich interface for selecting a color value
Select	an advanced version of the HTML `select` element
Slider	a graphical slider control, much like the one we built in this chapter
Spinner	an input field that lets you select a number by clicking up and down arrows to adjust the value

Again, as of this writing, Dojo's widgets are sparsely documented, so trying to use them without advanced JavaScript experience can be frustrating.

The Yahoo! UI Library (YUI) offers a less extensive collection of controls,[7] but those that it does provide are richly documented with plenty of beginner-friendly examples. Here's a list of the form-related YUI Controls as of this writing:

AutoComplete a text field that can pop up a list of suggested values in response to user input

Calendar a rich interface for selecting a date

Slider a graphical slider control, much like the one we built in this chapter

By all means, take some time to play with these widget libraries, and pay special attention to how well (or how badly, as is more often the case) they handle issues like keyboard navigation, screen reader accessibility, and semantically meaningful HTML code. While it can be tempting to trust these widget libraries implicitly, I believe you'll find there is definitely still a lot of value in the do-it-yourself option.

Whatever you decide, it's important to go in with your eyes open, and by understanding—at least in principle—how to build custom form controls yourself, you'll be better equipped to evaluate prepackaged options like these.

Summary

Back when JavaScript support was first added to web browsers, practically the only thing anyone could think to use it for was to enhance HTML forms. The tasks we've seen in this chapter—interlinking form fields, performing client-side validation of form submissions, and creating new types of form controls—are the things JavaScript does best, if only because it has been doing them for such a long time.

Although these form enhancements may not be as shiny and new as some of the other examples in this book, the techniques we've used to implement them—progressive enhancement, DOM manipulation, and unobtrusive scripting—certainly are. If the JavaScript pioneers that first set out to enhance HTML forms had had these techniques at their disposal, the JavaScript-enhanced forms that appear all over the Web today would work a lot better than they do.

[7] http://developer.yahoo.com/yui/#elements

Chapter 7

Errors and Debugging

A truth that many JavaScript books won't tell you is that JavaScript is a tough language to get right the first time.

By now you should have a fairly solid feel for JavaScript as a language, and how to use things like event listeners and the DOM API to enrich the web sites you build. But if you've actually tried to write an original script of your own, chances are that you came away feeling humbled, and maybe even a little angry.

That frustration's probably due, in part at least, to the fact that JavaScript, like all languages that run in the browser, is designed to fail silently by default. When things go wrong in the code you write, there's no point in shouting about it to your hapless visitors, so browsers just quietly set aside broken scripts and ignore them. The instructions in this chapter will show you how to get the browser to speak up, so you can find out about JavaScript errors as they happen.

However, even once you can *see* the error messages, you'll likely find that most JavaScript errors aren't all that helpful—especially if you're new to the language. Most of them are written in "programmer-ese," or complain about a perfectly good part of the code when the problem is actually elsewhere. So we'll spend some time

in this chapter deciphering the most common error messages you're likely to encounter, and what they really mean.

With those tools tucked in your belt, you should be able track down problems that the browser can detect for you. But you might still have occasion to wonder why on earth your carefully scripted code (which is perfectly fine, as far as the browser is concerned) is behaving the way it is. With the right tools, you can track a problem in your code to its source, even stepping through your JavaScript code a line at a time if necessary.

Nothing Happened!

It isn't very encouraging to spend two hours piecing together the ultimate script full of whizz-bang effects only to fire up your browser and have nothing happen; but when you're first writing a new script, that's usually your first hint that something has gone wrong—nothing happens.

The good news is that, if you've done your job right, the fact that the JavaScript fails silently means that your plain HTML/CSS can work on its own, and a user need never know that your code isn't working right. See? The browser's just looking after your reputation! This isn't much help when you're trying to find out *why* the script isn't working, though.

When you're working on your JavaScript code, then, you should configure your browser of choice to let you know about JavaScript errors when they happen. Depending on which browser you're dealing with (and of course, you will eventually need to test as many as possible), the procedure is different.

Firefox has a very nice error console that you can access by selecting **Tools** > **Error Console**. This opens the window shown in Figure 7.1. The console displays not only JavaScript errors, but errors in your CSS code, and even so-called **chrome errors** generated internally by the browser (usually after you've installed a buggy browser extension).

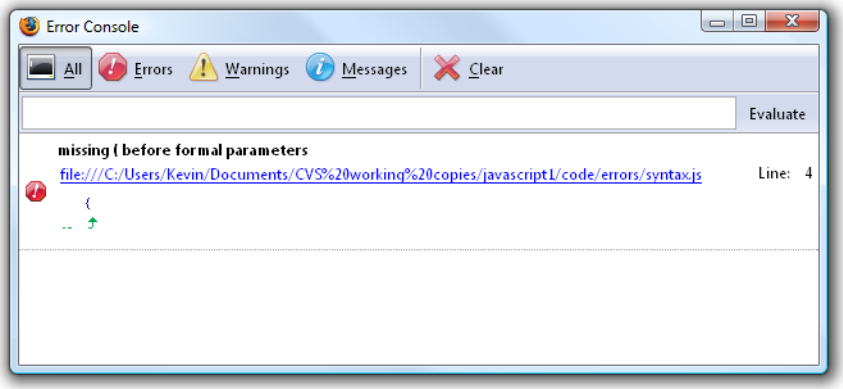


Figure 7.1. The Firefox Error Console

As you can see, the Firefox error console displays more than just errors:

Errors problems with your code that prevented the browser from continuing to run the script

Warnings problems with your code that the browser was able to work around, but which may indicate that the script isn't doing what you expect it to

Messages notes from your code that tell you what it's doing, usually only used as a debugging tool by browser extension developers—you won't see many of these

For each entry in the console, the specific file and line number that generated the notification will be displayed. Depending on the nature of the entry, the error console may even show you the line in question, with a little arrow pointing to the exact code that generated the entry.

When you're trying to track down a problem in your code, you'll usually start by opening the error console and clicking the **Clear** button to empty out the backlog of entries that may be displayed. Then you can reload the page in your main browser window and note the notifications that appear in the error console. If there are a lot of notifications, you might start by clicking the **Errors** button to see only the most severe errors, and concentrate on fixing them first, before returning to the **All** view to fix the less serious issues.

Opera's error console, shown in Figure 7.2, works very much like Firefox's. You can get to it by clicking **Tools** > **Advanced** > **Error Console**.

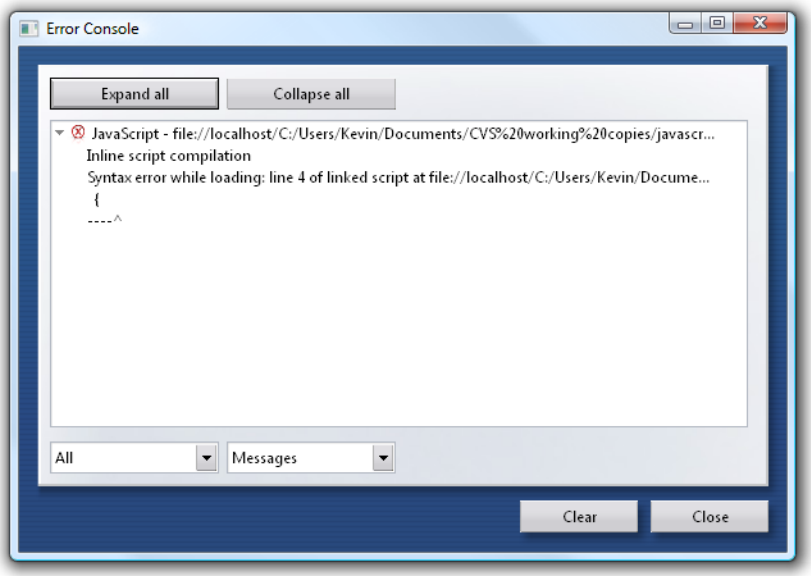


Figure 7.2. The Opera Error Console

The two drop-down menus at the bottom of this window are the key to seeing just the notifications that interest you. The first controls the source of the notifications that will be displayed, and you can choose **JavaScript** from this menu if you want to filter out things like HTML and CSS errors while working on a script. The second drop-down works much like the notification filtering buttons in Firefox's error console, enabling you to set the severity of the entries that are displayed in the console. The default selection, **Messages**, will allow messages, warnings, and errors to be displayed.

Useful error messages are harder to find in Internet Explorer. To see them, you need to open the Internet Options window (**Tools** > **Internet Options**), then, on the **Advanced** tab, look for the **Display a notification about every script error** option, under **Browsing**. Make sure it's checked, as shown in Figure 7.3.

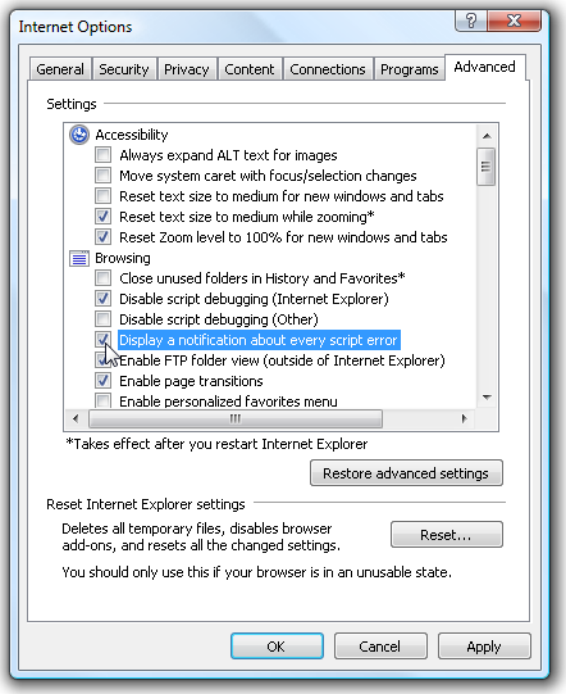


Figure 7.3. Enabling JavaScript errors in Internet Explorer

Once this option is set, you'll be notified the moment a JavaScript error has occurred, with a remarkably unhelpful message box similar to that shown in Figure 7.4.

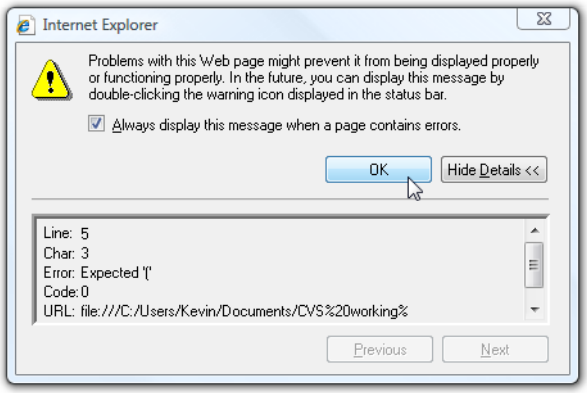


Figure 7.4. A JavaScript error notification in Internet Explorer

The only part of this window that you should really pay attention to is the line that begins with **Error:**. All the other information in this dialog (including the line and character numbers) is usually wrong. Heck, as of Internet Explorer 7, even the message at the top of this window was wrong! If you uncheck the **Always display this message when a page contains errors** checkbox, the warning icon in the status bar that it mentions will not actually be displayed.

As you can see, finding and fixing JavaScript errors in Internet Explorer is not easy. There are tools that can make it a little easier,[1] but because most JavaScript problems will affect *all* browsers, you're usually better off doing your JavaScript development in a different browser, and using Internet Explorer's JavaScript error reporting as a last resort for problems specific to that browser.

To access Safari's error console, you need to enable a hidden feature of that browser. Open a Terminal window and type:

```
defaults write com.apple.Safari IncludeDebugMenu 1
```

Press **Enter**, then quit the Terminal application. When you next launch Safari, you'll see a new **Debug** menu next to the **Help** menu. Make sure the **Log JavaScript Exceptions** option is checked in the menu, as shown in Figure 7.5, then click **Show JavaScript Console** to bring up the error console shown in Figure 7.6.

You'll note that, of the four browsers we've discussed in this section, Safari provides the tersest and least helpful error messages. It generally gets the file name and line number right, at least.

Now that you can *see* JavaScript error messages, you need to learn how to interpret them.

Common Errors

Every browser has its own particular dialect for JavaScript error messages. Almost invariably, Firefox produces the most sensible and helpful messages of current browsers, so your best bet when faced with a confusing message is to open your page in Firefox to see what its error console says.

[1] http://blogs.msdn.com/ie/archive/2004/10/26/247912.aspx

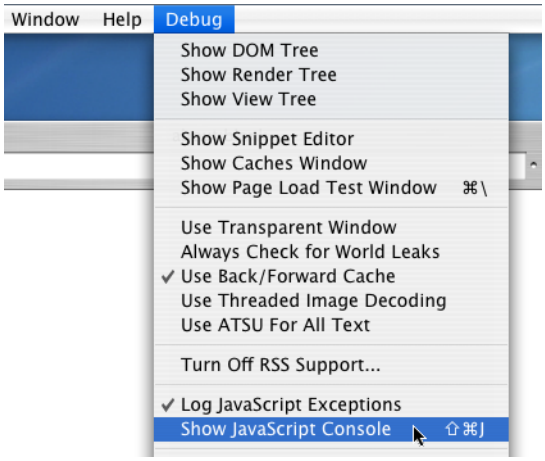


Figure 7.5. Safari's hidden **Debug** menu

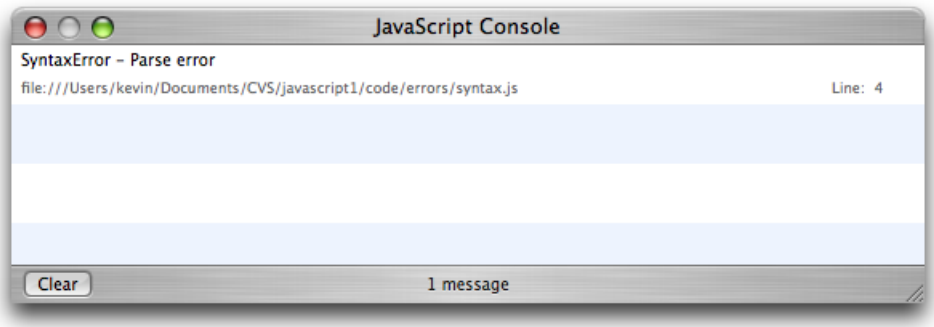


Figure 7.6. Safari's Error Console

Three kinds of errors can occur in JavaScript:

- syntax errors
- runtime errors
- logic errors

Only the first two produce error messages.

Syntax Errors

A **syntax error** is caused when your code violates the fundamental rules (or syntax) of the JavaScript language. It's the browser's way of saying, "Whatchoo talkin' 'bout,

Willis?"[2] Of the errors that the browser will tell you about, syntax errors are the easiest to fix, but the hardest to spot in your code.

Here's a simple script that contains a number of syntax errors:

syntax.js

```
// This script contains four syntax errors
var MyScript = {
  init: function
  {
    MyScript.doSomething();
  }
  doSomething: function()
  {
    alert("Hold onto your "hat"!");
    \\ something happens
  }
};

Core.start(MyScript);
```

If you load a page that links to this script (like **syntax.html** in the code archive) in Firefox, the error console will display the error message shown in Figure 7.7.

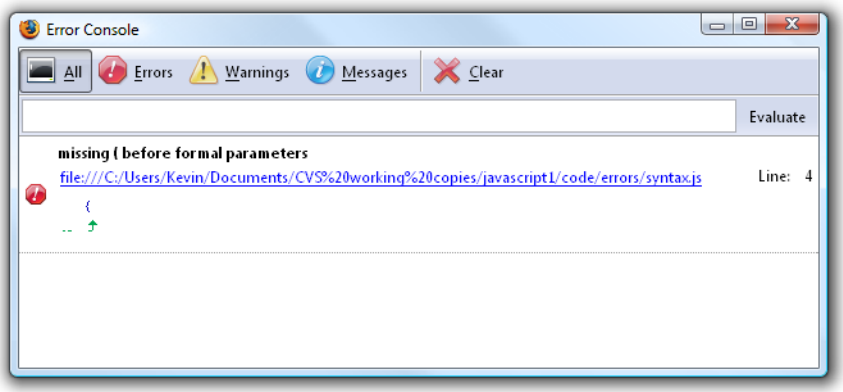


Figure 7.7. The first syntax error

[2] http://www.imdb.com/title/tt0077003/quotes

Because the browser gives up on trying to make sense of your script when it runs into a syntax error, you'll only ever be notified about the first syntax error in a given script. As you can see, the error message in this case is "missing (before formal parameters," and the browser ran into this error when it hit the opening brace ({) on line four of **syntax.js**.

Looking back at the script, it may not be immediately obvious to you what the problem is. The brace is certainly in the right place, and is required to mark the start of the `init` method's body, so why is the browser complaining about it? And what are "formal parameters" anyway?

The real problem here occurs on the previous line: the parentheses that must follow the `function` keyword were left out! These parentheses enclose the list of arguments for the function (also known as **formal parameters**), and are required. Here's what the corrected code looks like:

```
3   init: function()
4   {
5     MyScript.doSomething();
```

It turns out that most syntax errors occur when the browser was expecting one thing, but ran into something else instead. In this case, it was expecting an opening parenthesis (`(`), and encountered an opening brace (`{`) instead. That's why the error message points to the innocent-looking brace.

Usually, the error message will tell you what the browser expected to find instead (in this case, it complains about the "missing `(`"), but if the message doesn't make sense to you, a good tactic is to look at what immediately precedes the place in your code where the error occurred, and try to identify what you might've left out.

If you fix this error and reload the page, you'll have an excellent opportunity to try out this technique when you see the error message shown in Figure 7.8.

Can you find the problem? Again, the browser is complaining about a missing character—in this case a closing brace (`}`)—but if you look at the line before the error, there's a closing brace there already!

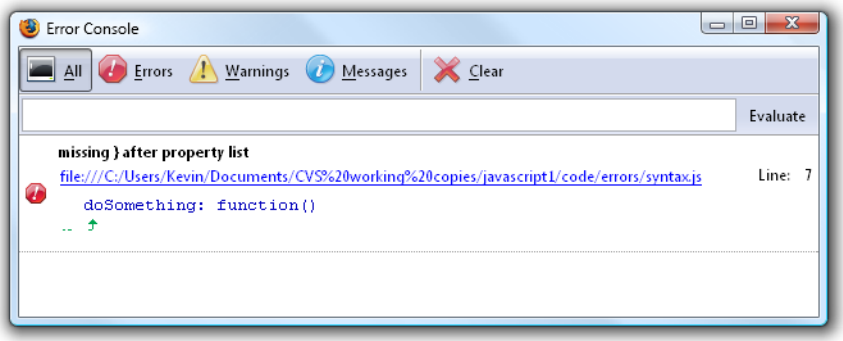


Figure 7.8. Another syntax error

syntax.js *(excerpt)*

```
6   }
7   doSomething: function()
```

Why is the browser complaining about a missing closing brace? Well, because it thinks you need *another* closing brace. Now why would it think something like that?

The error message says "missing } after property list" because what we're doing at this point in our code is building an object (`MyScript`) by providing a list of its properties and methods. At the end of that list, we'd normally end the object declaration with a closing brace, but judging by the error message, JavaScript thinks we meant the object to end right here, on line seven.

Looking at the code in this light, you can probably spot what's wrong: because the first method (`init`) isn't followed by a comma (`,`), the browser doesn't know that we want to declare a second, and falsely assumes that this is the end of the object declaration. To fix this error, we must add the missing comma:

```
6   },
7   doSomething: function()
```

This case highlights the fact that error messages are often just the browser's best guess at what you meant to say with your code. In an ideal world, the message would have said something like "missing } after property list, *unless you're declaring*

another property, in which case you're missing a comma," but unfortunately browsers just aren't that smart (yet). It's up to you to notice when an error message is based on a false assumption on the browser's part, and act accordingly.

If you fix this error and reload the page, you'll see another example error, as depicted in Figure 7.9. This is an easy one, so see if you can figure it out before reading on.

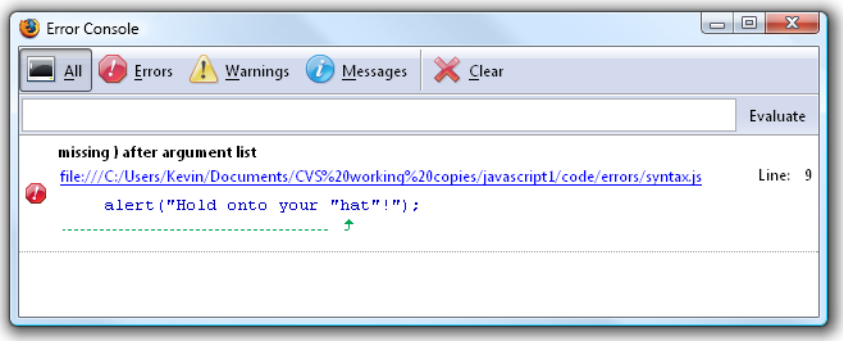


Figure 7.9. Yet another syntax error

From the error message, when the browser reached the "h" in "hat," it actually expected a closing parenthesis (`)`), because it thought it had reached the end of the argument list for the `alert` function call. Why would it think that in the middle of an argument?

Again, look at the code that immediately precedes the error for an explanation. Just before the "h" is a double quote (`"`), which the browser interprets as the end of the string `"Hold onto your "`—the first argument in your `alert` function call. After this argument, it expects either a comma (`,`) followed by the next argument, or a closing parenthesis to complete the list of arguments. The error message assumes you meant to choose the latter option.

Of course, in this case, you meant neither—you don't actually want that double quote to signal the end of the first argument! To fix this, escape the double quotes in the string with backslashes:

```
9     alert("Hold onto your \"hat\"!");
```

Figure 7.10 shows the final error that the browser will trip over in this script.

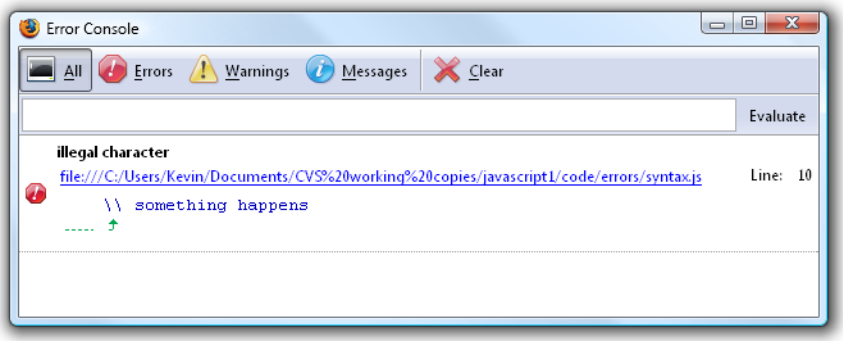


Figure 7.10. Guess what! (Yes—it's another syntax error)

Okay, I threw you a bone with this one. Just to show you that it *does* happen now and then, this error message says exactly what it means. The backslash at the start of the line is termed an **illegal character**, which is a fancy way of saying "Get a grip—you're not even *allowed* to type a backslash here!"

The problem, of course, is that comments must be preceded by a double slash, not a double backslash:

```
10     // something happens
```

As you can see, once you've figured out where you went wrong, syntax errors are easy to fix, but the error messages that the browser displays can often be misleading. Usually it comes down to a forgotten character or two, or accidentally using one type of bracket when you meant to use another.

The good news is that the more JavaScript you write, the more familiar you'll become with the nitty-gritty details of the language, and the fewer syntax errors you'll be likely to run into.

Runtime Errors

Runtime errors occur when a perfectly valid piece of JavaScript code tries to do something that it's not allowed to do, or that is flat out impossible. These errors are so named because they occur while the script is actually running. Unlike syntax error messages, the messages produced by runtime errors tend to be right on the money. The trick is figuring out why the error occurred.

This script contains a number of runtime errors:

runtime.js

```
1  // This script contains three runtime errors
2  var MyScript = {
3    init: function()
4    {
5      var example = document.getElementsById("example");
6
7      for (var i = 0; i < example.length; i++)
8      {
9        Core.addEventListener(example[i], "click", doSomething);
10     }
11
12     var links = documents.getElementsByTagName("a");
13     var firstLink = links[0];
14
15     if (firstLink && firstLink.className == "")
16     {
17       alert("The first link has no class assigned to it!");
18     }
19   },
20   doSomething: function(event)
21   {
22     alert("Hold onto your \"hat\"!");
23   }
24 };
25
26 Core.start(MyScript);
```

Once again, fire up Firefox and load the corresponding HTML file (**runtime.html**) to see the first error produced by this script—it's shown in Figure 7.11.

As you can see, runtime errors look just like syntax errors, except that the error console doesn't show the line of code that caused a runtime error.

An "is not a function" error usually indicates that you've misspelled the name of the function or method that you're trying to call, or that the function or method that you're trying to call simply doesn't exist.

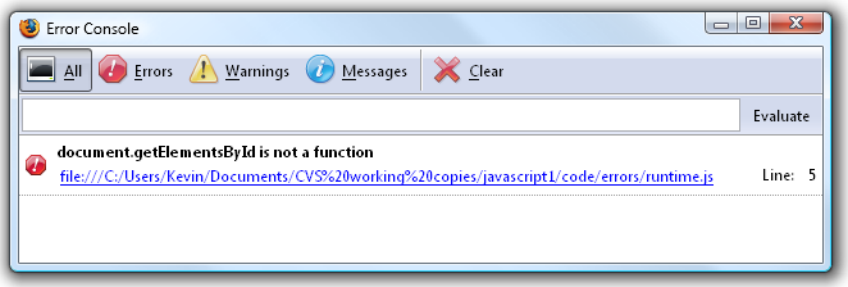


Figure 7.11. A runtime error ... how novel!

As a newcomer to JavaScript, you'll probably see this message a lot—especially if you aren't used to the case-sensitive nature of JavaScript. Attempting to call `Alert` instead of `alert`, for example, will produce this error message. But it can happen in more subtle cases too, like calling a string method like `toLowerCase` on a text node (which isn't a string), rather than the `nodeValue` of the text node (which is).

In this example, the cause is far simpler: the developer has tried to call a method called `getElementsById` when there is no such method. The method the developer was probably thinking of is `getElementById` (without the "s"). Since you can only have one element with a given ID in a document, it wouldn't make sense for there to be a method called `getElementsById`.

The fix in this case is trivial:

```
5     var example = document.getElementById("example");
```

But it turns out that fixing this problem actually causes another runtime error, shown in Figure 7.12.

A "has no properties" error means that you're trying to treat something that isn't an object as if it were an object, by trying to access a property or method on something that has neither.

The most common cause of this error is a method that normally returns an object (like `getElementById`) returning `null`, JavaScript's special "no object" value. If your code assumes that an object would be returned and treats the returned value as such, you'll end up with an error like this.

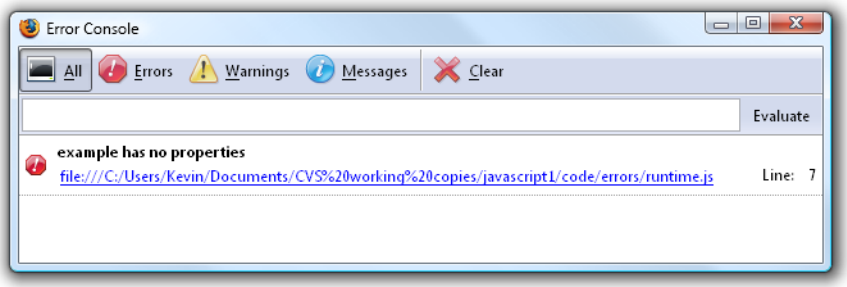


Figure 7.12. A runtime error that's slightly less clear

In this case, the error is being caused by the following line of code:

runtime.js *(excerpt)*

```
7     for (var i = 0; i < example.length; i++)
```

If you check **runtime.html**, you'll see that there isn't actually an element with ID `example`, so the `getElementById` call we just fixed above returns `null`, and when this `for` loop tries to read the `length` property of that `null` value, it produces the error message we're looking at now.

Apparently, whoever wrote this script assumed that the fictitious `getElementsById` method would return an empty array if it didn't find any elements with the specified ID. Since `getElementById` returns either a single element node or `null`, we need to replace the `for` loop with an `if` statement:

```
3   init: function()
4   {
5     var example = document.getElementsById("example");
6
7     if (example)
8     {
9       Core.addEventListener(example, "click", doSomething);
10    }
```

With that fixed, Figure 7.13 shows the last runtime error in this script.

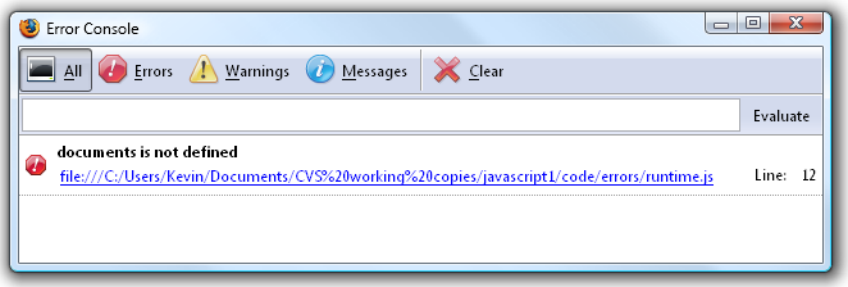


Figure 7.13. A runtime error that's easy to fix

An "is not defined" error is about as easy an error to fix as there is. Like the "is not a function" error we saw earlier, it usually results from a simple typing mistake, but rather than a misspelled function or method name, "is not defined" indicates you've tried to use a variable or property that doesn't exist.

In this case, the error is really straightforward:

runtime.js *(excerpt)*

```
12     var links = documents.getElementsByTagName("a");
```

As you can see, the developer has simply misspelled `document` as `documents`.

Fix this error, and the script will successfully display the alert message, "The first link has no class assigned to it!"

Logic Errors

Logic errors aren't so much errors as they are bugs in your script. The code runs fine as it was written—it just doesn't behave the way you expected it to when you wrote it. These kinds of errors can be devilishly difficult to find because as far as the browser's concerned the script is working just fine, so you never see an error message.

Since every logic error is different, each one presents a new challenge. Just in case you're not entirely clear what a logic error *is*, here's a script with a few logic errors in it:

logic.js

```
// This script contains three logic errors
var MyScript = {
  init: function()
  {
    var links = document.getElementsByTagName("a");
    var exampleLinks = [];
    for (var i = 0; i < links.length; i++)
    {
      if (links[i].className = "Example")
      {
        Core.addEventListener(
            links[i], "click", MyScript.doSomething);
        exampleLinks[exampleLinks.length] = links[i];
        i--;
      }
    }
  },
  doSomething: function(event)
  {
    alert("Hold onto your \"hat\"!");
  }
};

Core.start(MyScript);
```

This script is *supposed* to find and build an array that contains all links in the document that have a `class` attribute of `example`, and assign to each one the `doSomething` method as a `click` event listener.

Due to what could be described as a perfect storm of logic errors, this script actually attempts to *set* the `class` attribute of *every* link in the document to `Example`, but only gets as far as setting the first one in the document before going into an infinite loop that hangs the browser. Nice, huh?

If it's any consolation, you'd have to be pretty unlucky to innocently produce a script with so serious a combination of logic errors as this. However, taken in isolation, each of the logic errors in this script is a reasonably common bug of the type you may run into when writing your own scripts.

The most serious problem in the script is the infinite loop, which is just about as bad a problem as you'll ever create with a logic error. Older browsers would become completely unresponsive, and would have to be forcibly terminated by the user in the event of an infinite loop. These days, browsers will detect when a script has been running for a long time, and will display a message like the one shown in Figure 7.14, which offers to stop the script and return control to the user.

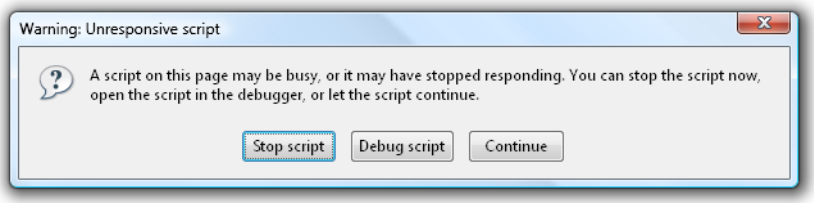


Figure 7.14. The prompt an infinite loop will produce (eventually)

Let's start by fixing this problem. Here's the code responsible for the infinite loop:

logic.js *(excerpt)*

```
12          exampleLinks[exampleLinks.length] = links[i];
13          i--;
```

When the script has discovered a link that it wants to add to the `exampleLinks` array, it does so with the first of these two statements. Unfortunately, the developer seems to have suffered a brain fart, and assumed that adding a reference to the link to the end of the `exampleLinks` array would remove it from the node list stored in `links`. In an effort to compensate for this imagined loss of an item from `links`, the developer has decremented the counter variable with the second statement (`i--;`).

In fact, the link remains in the `links` node list, and all the decrementing `i` achieves is to cause the `for` loop to process the same link over and over again—our infinite loop.

We can avoid the infinite loop simply by removing the second statement:

```
 7      for (var i = 0; i != links.length; i++)
 8      {
 9        if (links[i].className = "Example")
```

```
10        {
11          Core.addEventListener(
12              links[i], "click", MyScript.doSomething);
13          exampleLinks[exampleLinks.length] = links[i];
14        }
15      }
```

Run this corrected code, and the next thing you'll discover is that the `click` event listener is assigned to *every* link in the document. Behind the scenes, the `exampleLinks` array is also filled with all the links in the page. What's causing this issue?

The source of this bug is an extremely common mistake made by JavaScript beginners, who will often confuse the assignment operator (=) with the equality operator (==). See it now?

logic.js *(excerpt)*

```
9       if (links[i].className = "Example")
```

The condition in this `if` statement is supposed to be checking if the `className` property has a value *equal* to `"Example"`, but the developer has mistakenly used the assignment operator (=) here, causing this code to *set* the value of `className` to `"Example"`. An assignment statement used as a condition like this evaluates to the assigned value (`"Example"`), and since any non-empty string is considered "true," the `if` statement will execute for every link it processes. The solution, of course, is to use the correct operator:

```
9       if (links[i].className == "Example")
```

Run this modified code, however, and you'll suddenly find that the event listener isn't assigned to *any* of the links in the page—not even the first one in this paragraph:

logic.html *(excerpt)*

```
<p>This is an <a href="http://www.example.com/"
      class="example">example</a>, but this
  <a href="http://www.sitepoint.com/">is not</a>.</p>
```

This one's pretty obvious. The `if` statement is looking for a `className` of `"Example"` (capital "E"), but the `class` attribute of the link is `example` (lowercase "e"). Class names are case-sensitive, so we need to make sure the script matches the actual `class` value in the document:

```
9       if (links[i].className == "example")
```

Now be honest: how many of those errors could you have spotted without my help, based only on the behavior of the browser? And if you thought spotting logic errors in someone *else's* code was difficult, just wait till you're sitting in front of your *own* code armed only with the absolute conviction that your code is perfect in every way!

In the absence of error messages, you need a specialized tool to help you track down logic errors like the ones we've just seen.

Debugging with Firebug

In the past, the most common approach to resolving logic errors in JavaScript code was liberal use of the `alert` function. If you're expecting a `for` loop to run five times, you can stick an `alert` on the first line inside the loop and see if the browser displays five alert boxes when you load the page. If it does, you move the `alert` call somewhere else, to test another theory about why your script might be misbehaving.

Sound tedious? It is.

At the time of writing, a much more powerful (not to mention sane) approach is to use a JavaScript debugger, and by far the best debugger around is Firebug. Firebug is a free extension for Firefox that adds to the browser a panel containing a rich set of tools for diagnosing problems with your HTML, CSS, and JavaScript. You can download and install it from the Firebug web site,[3] shown in Figure 7.15.

[3] http://www.getfirebug.com/

Figure 7.15. Getting Firebug

Let me show you how to use Firebug to track down the infinite loop that we fixed in the previous section:

1. Open the page in your browser, and wait for the "Unresponsive Script" warning. Click **Stop script**.

2. Hit **F12**, or click the new Firebug status icon at the bottom of the browser window to open the Firebug panel, shown in Figure 7.16, at the bottom of your browser.

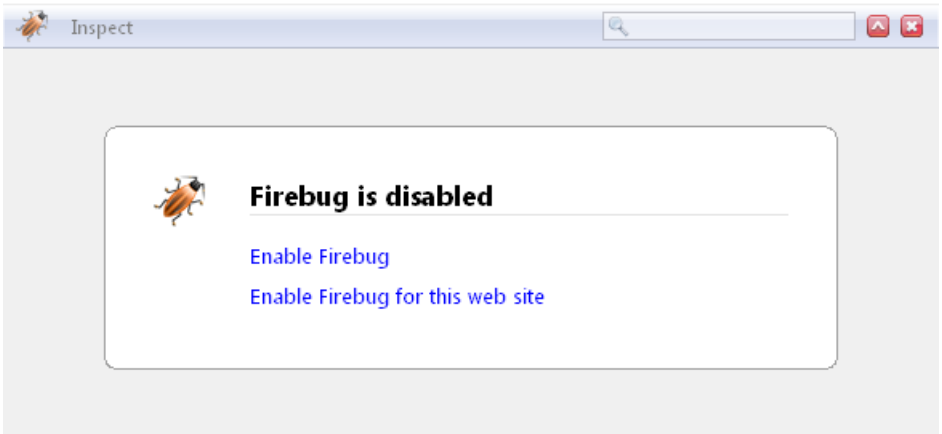


Figure 7.16. The Firebug panel

3. Since Firebug slows your browser's performance, it's disabled by default. Click either of the **Enable Firebug** links to enable Firebug. The first thing you'll see is the **Console** tab pictured in Figure 7.17.

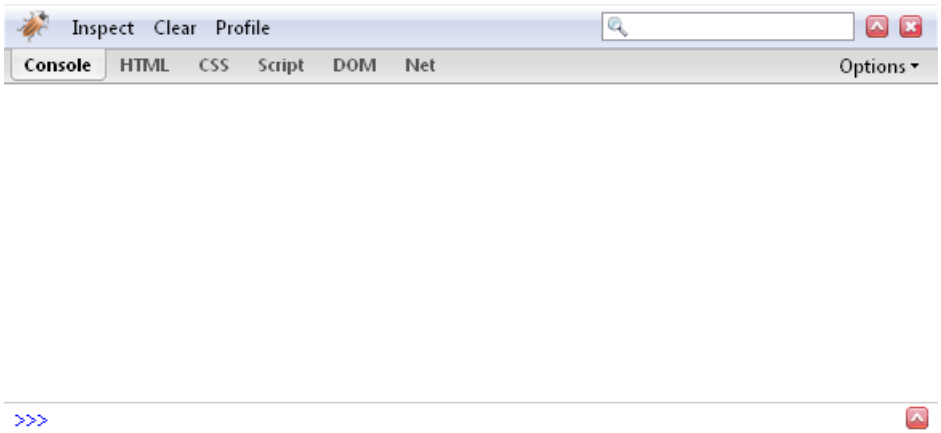


Figure 7.17. The **Console** tab

Like Firefox's error console, Firebug's **Console** tab will display JavaScript and CSS errors. It will also display useful information about Ajax requests, which you'll learn about in the next chapter, if you select **Show XMLHttpRequests** on the **Options** menu. You can type any JavaScript statement or expression into

the command line at the bottom of the tab, and Firebug will execute it in the currently displayed page and display the result in the console.

4. To fix the infinite loop in this page, however, we need something more powerful than the **Console** tab. Click the **Script** tab to see Firebug's JavaScript debugger, which is depicted in Figure 7.18.

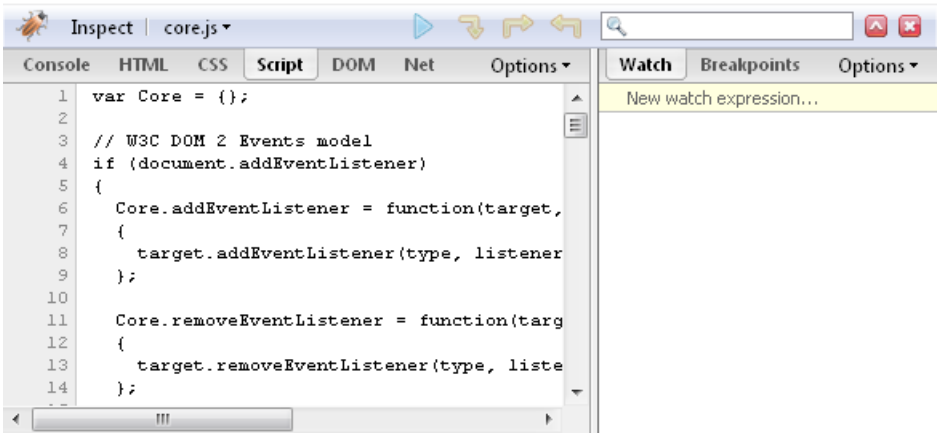


Figure 7.18. The **Script** tab

5. The debugger lets you pause the execution of your scripts and step through them one line at a time, observing the values of variables and the structure of the DOM as you go. Since the problem is likely to be in the **logic.js** file, start by selecting it from the drop-down menu at the top of the Firebug tab, as shown in Figure 7.19.

6. Firebug now displays the code of the **logic.js** file. We can tell by the way the browser is hanging that we're dealing with an infinite loop, and the only loop in the code is the `for` loop on line 7, so click in the gutter next to that line number, as shown in Figure 7.20. This sets a **breakpoint**, represented by a red circle, which tells the debugger to pause the execution of your script when it reaches that line.

7. With your breakpoint in place, reload the page. As Figure 7.21 indicates, a yellow arrow will appear on top of the breakpoint to indicate that execution of the script has been paused at that line. The **Watch** tab in the debugger's right-

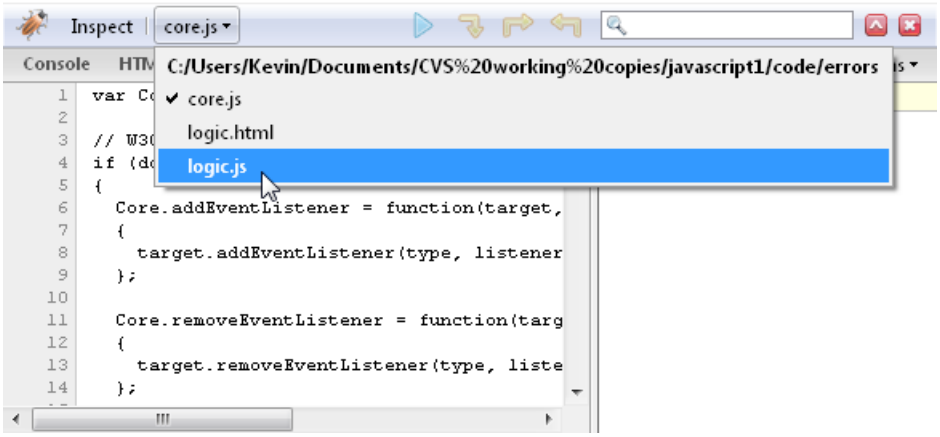


Figure 7.19. Selecting the file to debug

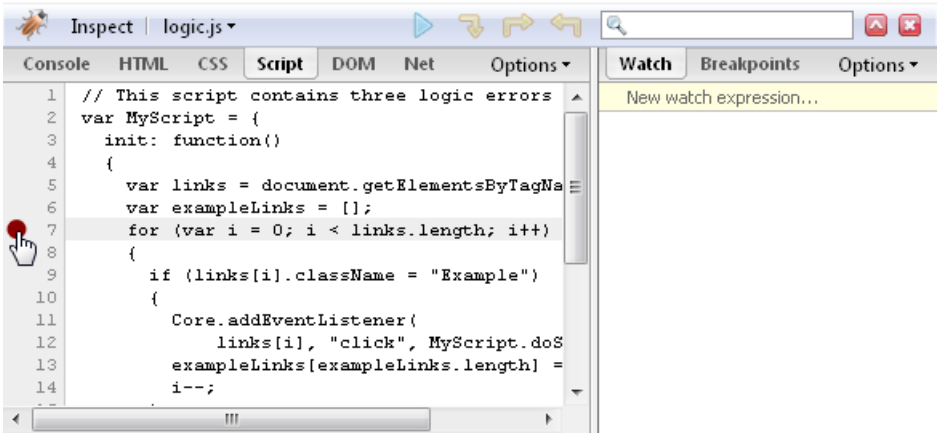


Figure 7.20. Setting a breakpoint

hand pane will also show the local variables that exist on that line, and their current values.

While the debugger is paused, you can click one of the four buttons at the top of the Firebug panel to control the execution of your script. The blue arrow resumes normal execution of the script, which will continue until it reaches the next breakpoint, if any. The three gold arrows let you step through your script one statement at a time. The first, **Step Over**, simply executes the current statement and pauses execution again on the next line of the current code listing. The second, **Step Into** works just like **Step Over**, except when the current state-

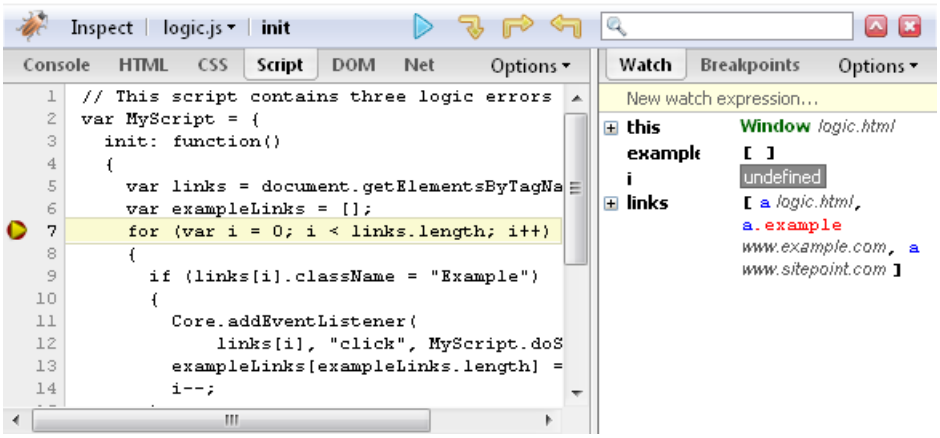


Figure 7.21. Pausing execution

ment contains a function or method call. In such cases, the debugger will step *into* the call, pausing execution on the first line inside the function/method. The third arrow, **Step Out**, allows the script to finish executing the current function, and pauses execution on the next line of the code that called it.

8. Click **Step Over** once to move to the first line inside the `for` loop.

9. Now, it would be nice to know about the current link being processed by the `for` loop, `links[i]`. You can eyeball it by looking at the value of `i` in the **Watch** pane, then expanding the `links` variable in that pane to find the corresponding element of the array. Alternatively, you can click the area labeled **New watch expression...**, type **`links[i]`**, and press **Enter** to add the expression to the list of local variables in the **Watch** pane, as shown in Figure 7.22.

   So we can tell that the first time through the loop, `links[i]` is pointing to the hyperlink to `logic.html`.

10. Click the yellow arrow in the gutter to set another breakpoint, this time on line 9.

11. Click the blue arrow to resume execution of the script. The `for` loop finishes its first iteration and starts its second. Execution pauses at the new breakpoint on line 9. Already, as shown in Figure 7.23, you can spot a number of clues about what's going wrong:

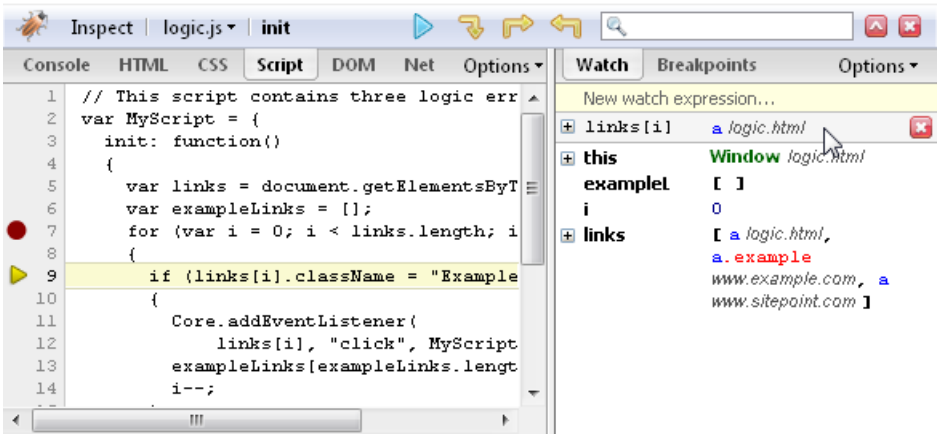


Figure 7.22. Adding a custom watch expression

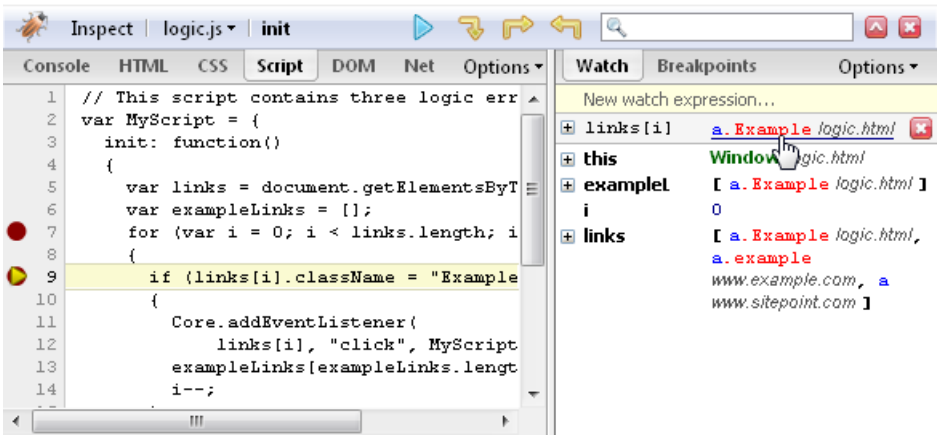


Figure 7.23. Examining the clues

- `i` still has a value of `0`, even though we're in the second iteration of the loop.
- `links[i]` still refers to the link to `logic.html`, except that it now has a class of `Example`.
- The `links` node list still contains all three of the links in our document.

At this point, a perceptive developer would be looking very hard at that `if` statement and the `i--;` inside it. If you needed more to go on, you could step through the body of the `for` loop line by line to see exactly what's going on.

Firebug has *tons* of other cool stuff in it. Spend some time reading the Firebug web site to learn about the other features that it offers to aid you in your quest for the answer to the eternal question, "Why is the browser *doing* that?" And if you like what you see, think about donating a few bucks to the development of this incredible tool—I have.

Summary

That's it! You can go out and brag to your friends that you know JavaScript now. From here on in, we'll look at extra browser features and other software that can make JavaScript do more.

In the next chapter, we'll delve into the mysteries of Ajax, whose sheer buzzword power may well be the reason you *bought* this book. If so, you'll be pleased to know that the long wait is over. Turn the page, and bask in the buzz.

Chapter 8

Ajax

It's probably not an understatement to say that Ajax has revitalized the Web. It's certainly one of the reasons for a resurgent interest in JavaScript and it might even be the reason that you're reading this book.

Irrespective of the hyperbole surrounding Ajax, the technology has dramatically affected the way in which people can interact with a web page. The ability to update individual parts of a page with information from a remote server mightn't sound like a revolutionary change, but it has facilitated a seamless type of interaction that has been missing from HTML documents since their inception.

This ability to create a fluidly updating web page has captured the imaginations of developers and interaction designers alike; they've flocked to Ajax in droves, using it to create the next generation of web applications, as well as annoy the heck out of the average user. As with any new technology, there's a temptation to overuse Ajax; but when it's used sensibly, it can definitely create more helpful, more responsive, and more enjoyable interfaces for users to explore.

Although quite a few of the JavaScript libraries out there will offer you a complete "Ajax experience" in a box, there is really no substitute for the freedom that comes with knowing how it works from the ground up. So let's dive in!

XMLHttpRequest: Chewing Bite-sized Chunks of Content

The main concept of Ajax is that you're instructing the browser to fetch small pieces of content instead of big ones; instead of a page you might request a single paragraph.

Although cross-browser Ajax-type functionality was hacked together previously with `iframes`, the current Ajax movement was sparked when `XMLHttpRequest` became available in more than just Internet Explorer.

`XMLHttpRequest` is a browser feature that allows JavaScript to make a call to a server without going through the normal browser page-request mechanism. This means that JavaScript can make additional server requests behind the scenes while a page is being viewed. In effect, this allows us to pull down extra data from the server, then manipulate the page using the DOM—replacing sections, adding sections, or deleting sections depending on the data we receive. The distinction between normal and Ajax requests is illustrated in Figure 8.1.

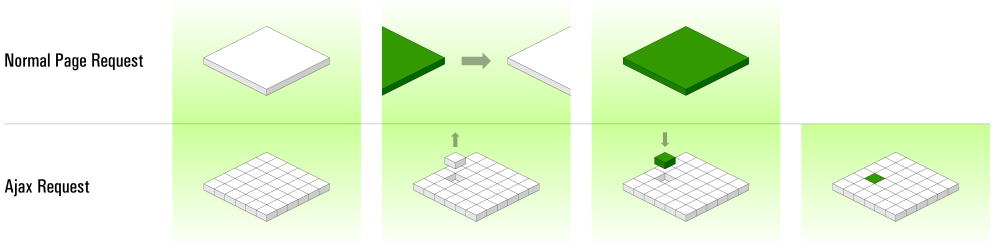


Figure 8.1. Comparing a normal page request (replacing the whole page) with an Ajax request (replacing part of the page)

Communications with the server that don't use the page-request mechanism are called **asynchronous requests**, because they can be made without interrupting the user's page interaction. A normal page request is synchronous, in that the browser waits for a response from the server before any more interaction is allowed.

`XMLHttpRequest` is really the only aspect of Ajax that's truly new. Every other part of an Ajax interaction—the event listener that triggers it, the DOM manipulation that updates the page, and so on—has been covered in previous chapters of this book already. So, once you know how to make an asynchronous request, you're ready to go.

Creating an `XMLHttpRequest` Object

Internet Explorer 5 and 6 were the first browsers to implement `XMLHttpRequest`, and they did so using an ActiveX object:[1]

```
var requester = new ActiveXObject("Microsoft.XMLHTTP");
```

Every other browser that supports `XMLHttpRequest` (including Internet Explorer 7) does so without using ActiveX. A request object for these browsers looks like this:

```
var requester = new XMLHttpRequest();
```

ActiveX is Unreliable

The way that `XMLHttpRequest` is implemented in Internet Explorer 6 and earlier means that if a user has disabled *trusted* ActiveX controls, `XMLHttpRequest` will be unavailable to you even if JavaScript is enabled. Many people disable *untrusted* ActiveX controls, but disabling trusted ActiveX controls is less common.

We can easily reconcile the differences between the two methods of object creation using a `try-catch` statement, which will automatically detect the correct way to create an `XMLHttpRequest` object:

try-catch_test.js *(excerpt)*

```
try
{
  var requester = new XMLHttpRequest(); (1)
}
```

[1] An ActiveX object is Microsoft's term for a reusable software component that provides encapsulated, reusable functionality. In Internet Explorer, such objects normally give client-side scripting access to operating system facilities like the file system, or in the case of `XMLHttpRequest`, the network layer.

```
catch (error) 2
{
  try
  {
    var requester = new ActiveXObject("Microsoft.XMLHTTP"); 3
  }
  catch (error)
  {
    var requester = null;
  }
}
```

A `try` statement allows you to try out a block of code, but if anything inside that block causes an error, the program won't stop execution entirely; instead, it moves onto the `catch` statement and runs the code inside that. As you can see in Figure 8.2, the whole structure is like an `if-else` statement, except the branch taken is conditional on any errors occurring.

We need to use a `try-catch` statement to create ActiveX objects because an object detection test will indicate that ActiveX controls are still available even if a user has disabled them (though your script will throw an error when you actually try to create an ActiveX object).

The Damage Done

If an error occurs inside a `try` statement, the program will not revert to the state it had before the `try` statement was executed—instead, it will switch immediately to the `catch` statement. Thus, any variables that were created *before* the error occurred will still exist. However, if an error occurs while a variable is being assigned, that variable will not be created at all.

Here's what happens in the code above:

1. We try to create an `XMLHttpRequest` object using the cross-browser method. If our attempt is successful, the variable `requester` will be a new `XMLHttpRequest` object. But if `XMLHttpRequest` is unavailable, the code will cause an error. We can try out a different method inside the `catch` statement.

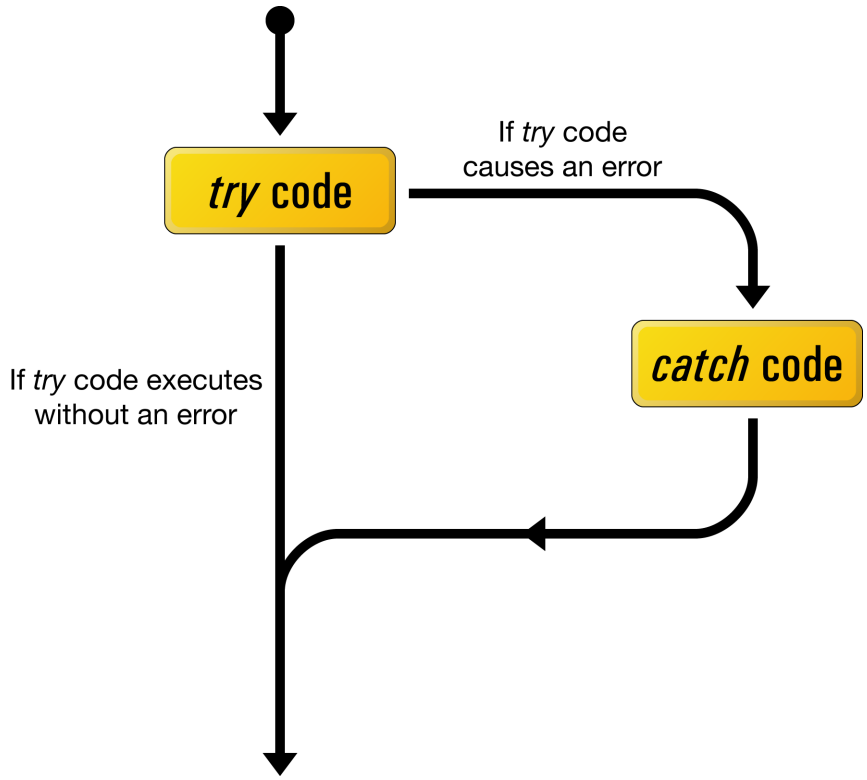


Figure 8.2. The logical structure or a `try-catch` statement

2. `catch` statements "catch" the exception that caused the `try` statement to fail. This exception is identified by the variable name that appears in brackets after `catch`, and it's mandatory to give that exception a name (even if we're not going to use it). You can give the exception any name you like, but I think `error` is nicely descriptive.

3. Inside the `catch`, we try to create an `XMLHttpRequest` object via ActiveX. If that attempt is successful, `requester` will be a valid `XMLHttpRequest` object, but if we still can't create the object, the second `catch` statement sets `requester` to `null`. This makes it easy to test whether the current user agent supports `XMLHttpRequest`, and to fork to some non-Ajax fallback code (such as that which submits a form normally):

```
if (requester == null)
{
  code for non-Ajax clients
}
else
{
  code for Ajax-enabled clients
}
```

Thankfully, the significant differences between browser implementations of `XMLHttpRequest` end with its creation. All of the basic data communication methods can be called using the same syntax, irrespective of the browser in which they're running.

Calling a Server

Once we've created an `XMLHttpRequest` object, we must call two separate methods—`open` and `send`—in order to get it to retrieve data from a server.

`open` initializes the connection and takes two required arguments, with several optionals. The first argument is the type of HTTP request you want to send (GET, POST, DELETE, etc.); the second is the location from which you want to request data. For instance, if we wanted to use a GET request to access **feed.xml** in the root directory of a web site, we'd initialize the `XMLHttpRequest` object like this:

```
requester.open("GET", "/feed.xml", true);
```

The URL can be either relative or absolute, but due to cross-domain security concerns the target must reside on the same domain as the page that's requesting it.

HTTP Only

Quite a few browsers will only allow `XMLHttpRequest` calls via `http://` and `https://` URLs, so if you're viewing your site locally via a URL beginning with `file://`, your `XMLHttpRequest` call may not be allowed.

The third argument of `open` is a Boolean that specifies whether the request is made asynchronously (`true`) or synchronously (`false`). A synchronous request will freeze

the browser until the request has completed, disallowing user interaction in the interim. An asynchronous request occurs in the application's background, allowing other scripts to run and the user to access the browser at the same time. I recommend you use asynchronous requests; otherwise, you run the risk of users' browsers locking up while they wait for a request that has gone awry. `open` also has optional fourth and fifth arguments that specify the user's name and password for authentication purposes when a password-protected URL is requested.

Once `open` has been used to initialize a connection, the `send` method activates that connection and makes the request. `send` takes one argument that allows you to send encoded data along with a POST request, in the same format as a form submission:

```
requester.setRequestHeader("Content-Type",
    "application/x-www-form-urlencoded");
requester.open("POST", "/query.php", true);
requester.send("name=Clark&email=superman@justiceleague.xmp");
```

`Content-Type` Required

Opera requires you to set the `Content-Type` header of a POST request using the `setRequestHeader` method. Other browsers don't require it, but it's the safest approach to take to allow for all browsers.

To simulate a form submission using a GET request, you need to hard-code the names and values into the `open` URL, then execute `send` with a `null` value:

```
requester.open("GET",
    "query.php?name=Clark&email=superman@justiceleague.xmp", true);
requester.send(null);
```

Internet Explorer doesn't require you to pass any value to `send`, but Mozilla browsers will return an error if no value is passed; that's why `null` is included in the above code.

Once you've called `send`, `XMLHttpRequest` will contact the server and retrieve the data that you requested. In the case of an asynchronous request, the function that created the connection will likely finish executing while the retrieval takes place.

In terms of program flow, making an `XMLHttpRequest` call is a lot like `setTimeout`, which you'll remember from Chapter 5.

We use an event handler to notify us that the server has returned a response. In this particular case, we'll need to handle changes in the value of the `XMLHttpRequest` object's `readyState` property, which specifies the status of the object's connection, and can take any of these values:

0 uninitialized

1 loading

2 loaded

3 interactive

4 complete

We can monitor changes in the `readyState` property by handling `readystatechange` events, which are triggered each time the property's value changes:

```
requester.onreadystatechange = readystatechangeHandler;

function readystatechangeHandler()
{
  code to handle changes in XMLHttpRequest readystate
}
```

`readyState` increments from `0` to `4`, and the `readystatechange` event is triggered for each increment. However, we only really want to know when the connection has completed (that is, `readyState` equals `4`), so our handling function needs to check for this value.

Upon the connection's completion, we also have to check whether the `XMLHttpRequest` object successfully retrieved the data, or was given an HTTP error code such as `404` (page not found). You can determine this from the request object's `status` property, which contains an integer value. A value of `200` is a fulfilled request; you should check for it—along with `304` (not modified)—as these values indicate successfully retrieved data. However, `status` can take as a value *any* of the HTTP codes that servers are able to return, so you may want to write some conditions

that will handle some other codes. In general, however, you'll need to specify a course of action for your program to take if the request is not successful:

```
requester.onreadystatechange = readystatechangeHandler;

function readystatechangeHandler()
{
  if (requester.readyState == 4)
  {
    if (requester.status == 200 || requester.status == 304)
    {
      code to handle successful request
    }
    else
    {
      code to handle failed request
    }
  }
}
```

Instead of assigning a function that's defined elsewhere as the `readystatechange` event handler, you can declare a new, anonymous (unnamed) function inline:

```
requester.onreadystatechange = function()
{
  if (requester.readyState == 4)
  {
    if (requester.status == 200 || requester.status == 304)
    {
      code to handle successful request
    }
    else
    {
      code to handle failed request
    }
  }
}
```

The advantage of specifying the `readystatechange` callback function inline like this is that the `requester` object will be available inside that function via a closure. If the `readystatechange` handler function is declared separately, you'll need to

jump through hoops to obtain a reference to the `requester` object inside the handling function.

XMLHttpRequest is Non-recyclable

Even though an `XMLHttpRequest` object allows you to call the `open` method multiple times, each object can effectively only be used for one call, as the `readystatechange` event refuses to fire again once `readyState` changes to `4` (in Mozilla browsers). Therefore, you will have to create a new `XMLHttpRequest` object every time you want to retrieve new data from the server.

Dealing with Data

If you've made a successful request, the next logical step is to read the server's response. Two properties of the `XMLHttpRequest` object can be used for this purpose:

`responseXML` This property stores a DOM tree representing the retrieved data, but only if the server indicated a `content-type` of `text/xml` for the response. This DOM tree can be explored and modified using the standard JavaScript DOM access methods and properties we explored in Chapter 3, such as `getElementsByTagName`, `childNodes`, and `parentNode`.

`responseText` This property stores the response data as a single string. If the `content-type` of the data supplied by the server was `text/plain` or `text/html`, this is the only property that will contain data. In the case of a `text/xml` response, this property will also contain the XML code as a text string, providing an alternative to `responseXML`.

In simple cases, plain text works perfectly well as a means of transmitting and handling the response, so the `XMLHttpRequest` object doesn't exactly live up to its name. When we're dealing with more complex data structures, however, XML can provide a convenient way to express those structures:

```
<?xml version="1.0" ?>
<user>
  <name>Doctor Who</name>
```

```
  <email>thedoctor@tardis.biz</email>
</user>
<user>
  <name>The Master</name>
  <email>themaster@gallifrey.org</email>
</user>
```

`responseXML` allows us to access different parts of the data using all the DOM features with which we're familiar from our dealings with HTML documents. Remember that data contained between tags is considered to be a text node inside the element in question. With that in mind, extracting a single value from a `responseXML` structure is reasonably easy:

```
var nameNode =
    requester.responseXML.getElementsByTagName("name")[0];
var nameTextNode = nameNode.childNodes[0];
var name = nameTextNode.nodeValue;
```

The `name` variable will now take as its value the first user's name: `"Doctor Who"`.

Whitespace Generates Text Nodes

As in HTML documents, be aware that whitespace between tags in XML will often be interpreted as a text node. If in doubt, remember that you can check if a given node is an element by looking at its `nodeType` property, as described in Chapter 4.

We can use the data contained in the XML to modify or create new HTML content, updating the interface on the fly in the manner that has become synonymous with Ajax.

The main downside of using XML with JavaScript is that a fair amount of work can be involved in parsing XML structures and accessing the information we want. However, an alternative way to use the `XMLHttpRequest` object is to remove this data processing layer and allow the server to return HTML code, all ready to insert into your page. This approach is taken by libraries such as Prototype, in which HTML is delivered with a MIME type of `text/html` and the value of `responseText` is automatically inserted into the document using `innerHTML`, overwriting the contents of an existing element.

As with any use of `innerHTML`, this technique suffers from the disadvantages we discussed in Chapter 3, but it can certainly be a viable option if your circumstances require it.

A Word on Screen Readers

Until now, we've always taken time to make sure our JavaScript enhancements do not prevent users of assistive technologies—like screen readers—from using our sites. Unfortunately, when you add Ajax to the equation, this goal becomes extremely difficult, if not impossible to achieve for screen reader users in particular.

Most, if not all of the current screen readers are unable to handle in a sensible (let alone useful) way the on-the-fly page updates that typify Ajax development. Screen readers either will not pick up those changes at all, or they'll pick them up at the most untimely of moments.

In some very specific cases, developers have begun to produce experimental solutions that start to address these issues, but we're a long way from having reliable, best-practice techniques in hand. The prevailing wisdom suggests that any real solution will have to be developed at least in part by the screen reader vendors themselves.

This leaves web developers like you and me with a tough decision: do we abandon Ajax and the amazing usability enhancements that it makes possible, or do we shut out screen reader users and take full advantage of Ajax? Of course, if you can justify asking screen reader users to disable JavaScript when visiting your site, you can offer these users the same fallback experience as other non-JavaScript users, which can work just fine. But you'll need to make sure these users can find out enough about your site to decide if it's worth disabling JavaScript to proceed. And consider making it even easier—a link that said "Disable user interface features on this site that are not compatible with screen readers," for example, would not be out of the question.

Putting Ajax into Action

Now you know the basics of Ajax—how to create and use an `XMLHttpRequest` object. But it's probably easier to understand how Ajax fits into a JavaScript program if you try out a simple example.

In this example, we'll retrieve information that's relevant to the selections users make from the widget shown in Figure 8.3. This tool allows users to choose any of three cities; each selection will update the widget's display with the weather for that location.

Figure 8.3. In our weather widget, the user is asked to select a particular city

The HTML for the widget looks like this:

weather_widget.html *(excerpt)*

```
<div id="weatherWidget">
  <h2>Weather</h2>
  <p>Please select a city:</p>
  <ul>
    <li>
      <a href="/weather/london/">London</a>
    </li>
    <li>
      <a href="/weather/new_york/">New York</a>
    </li>
    <li>
      <a href="/weather/melbourne/">Melbourne</a>
    </li>
  </ul>
</div>
```

We'll override those anchors with our Ajax code, but it's important to note that the `href` attribute of each anchor points to a valid location. This means that users who have JavaScript or `XMLHttpRequest` turned off will still be able to get the information; they just won't be gobsmacked by our cool use of Ajax to retrieve it.

When creating Ajax functionality, you can generally follow this pattern:

1. Initialize event listeners.
2. Handle event triggers.
3. Create an `XMLHttpRequest` connection.
4. Parse data.
5. Modify the page.

To handle the behavior of this weather widget, we'll create a `WeatherWidget` object. Its initialization function will start by adding event listeners to those anchor tags, which will capture any clicks that the user makes:

weather_widget.js *(excerpt)*

```
var WeatherWidget =
{
  init: function()
  {
    var weatherWidget = document.getElementById("weatherWidget");
    var anchors = weatherWidget.getElementsByTagName("a");

    for (var i = 0; i < anchors.length; i++)
    {
      Core.addEventListener(anchors[i], "click",
          WeatherWidget.clickListener);
    }
  },
  ⋮
};
```

Each of those anchors now has a `click` event listener, but what happens when the event is fired? Let's fill out the listener method, `clickListener`:

```
clickListener: function(event)
{
  try
  {
    var requester = new XMLHttpRequest(); (1)
  }
  catch (error)
  {
    try
    {
      var requester = new ActiveXObject("Microsoft.XMLHTTP");
    }
    catch (error)
    {
      var requester = null;
    }
  }

  if (requester != null) (2)
  {
    var widgetLink = this;
    widgetLink._timer = setTimeout(function() (3)
        {
          requester.abort();

          WeatherWidget.writeError(
              "The server timed out while making your request.");
        }, 10000);

    var city = this.firstChild.nodeValue; (4)

    requester.open("GET", "ajax_weather.php?city=" +
        encodeURIComponent(city), true); (5)
    requester.onreadystatechange = function() (6)
    {
      if (requester.readyState == 4)
      {
        clearTimeout(widgetLink._timer);

        if (requester.status == 200 || requester.status == 304)
        {
          WeatherWidget.writeUpdate(requester.responseXML);
        }
        else
```

```
        {
          WeatherWidget.writeError(
              "The server was unable to be contacted.");
        }
      }
    };
    requester.send(null); 7

    Core.preventDefault(event); 8
  }
}
```

1 The start of this function is occupied by the standard `XMLHttpRequest` creation code that we looked at earlier in this chapter.

2 Once that has been executed, the real logic of `clickListener` is contained inside the conditional statement `if (requester != null)`. By using this condition, we ensure that the Ajax code is only executed if the `XMLHttpRequest` object is available. Otherwise, the event handling function will exit normally, allowing the browser to navigate to the `href` location, just as it would if JavaScript wasn't enabled. This approach provides an accessible alternative for users without Ajax capability.

3 This is a real-world Ajax application that's subject to the unreliability of network traffic, so before the actual Ajax call is made, it's a good idea to place a time limit on the transaction to ensure that the user won't be sitting around waiting forever if the server fails to respond. To establish this limit, we assign a `setTimeout` call as a custom property of the link that was clicked (`widgetLink`), with a ten-second delay. If the Ajax request takes longer than ten seconds, the function supplied to `setTimeout` will be called, canceling the request via the `abort` method, and supplying the user with a sensible error message. Just like the `readystatechange` listener, the timeout function is specified inline, so `requester` will be available via a closure. We'll have to remember to stop this timeout later, if and when the Ajax request is actually completed.

4 In the first line of code after the timeout function, we determine which city was selected. To do so, we get the value of the text from the anchor that was clicked. We assume that the link contains a text node—so that will be the first child node of the anchor—and the city name itself will be the `nodeValue` of

that text node. We can pass this value to our server-side script in order to access the weather data for that particular city.

5. Once the requested city has been identified, we begin our Ajax connection. For this example we're using a GET request, and the server-side script we're trying to access is at `ajax_weather.php`. Since we're using GET, the request variables have to be encoded directly into the URL. To do so, we append a question mark (`?`) to the script location, followed by the variables specified as *`name=value`* pairs. If you want to pass multiple variables (we don't in this case), each pair must be separated by an ampersand (`&`).

   In this example, we're passing the city as a request variable called `city`; its value is the city name we extracted from the link. The actual value is generated by the built-in `escapeURIComponent` function, which will encode values so that they don't cause errors when being used as part of a URL. You should encode any string values that you attach to a URL this way.

6. Once the `requester` object has been initialized with `requester.open`, we set up our `readystatechange` event handler as an anonymous inline function. The code inside this handler is almost identical to the template that we outlined earlier in this chapter, except that we have filled in the actions that are to be taken for successful and unsuccessful requests. You should also note that once a response has been received from the server, a `clearTimeout` call is executed to cancel the `setTimeout` call we made earlier. This ensures that the user won't receive an error message about the server timing out when it hasn't actually done so.

7. Directly after the `onreadystatechange` function declaration, we fire off the requester's `send` method. The Ajax call is now in action.

8. We don't want any clicks on our Ajax links to take the user to a new page (that would defeat the links' purposes!), so the last line of `clickListener` stops the browser from performing the link's default action.

What actually occurs on the server after we make our Ajax request isn't the concern of our JavaScript. In our code, we've referred to a PHP script that will return information on the basis of the value of the city we passed it, but we could equally refer

to a JSP script, a Ruby on Rails action, or a .NET controller. Whatever technology is used, we simply need it to return some correctly formatted XML.

If a successful request occurred, we call our `writeUpdate` method, and pass it the `responseXML` data returned by the server. If there was an *unsuccessful* request, we call `writeError` and give it a suitable error message.

When `writeUpdate` is called, we know that we've got some XML data waiting to be parsed and added to our HTML. In order to use it, we need to extract particular data points from the XML, then insert them into appropriate elements in our page.

When you're liaising with a custom server-side script, you'll have to agree on a format for the XML, so that the server-side script can write it correctly and the JavaScript can read it correctly. For this example, we're going to assume that the XML has a form like this:

melbourne.xml

```
<?xml version="1.0" ?>
<city>
  <name>Melbourne</name>
  <temperature>18</temperature>
  <description>Fine, partly cloudy</description>
  <description_class>partlyCloudy</description_class>
</city>
```

Knowing this structure makes it easy for us to write `writeUpdate` to extract the pertinent data:

weather_widget.js *(excerpt)*

```
writeUpdate: function(responseXML)
{
  var nameNode = responseXML.getElementsByTagName("name")[0]; (1)
  var nameTextNode = nameNode.firstChild;
  var name = nameTextNode.nodeValue;

  var temperatureNode =
      responseXML.getElementsByTagName("temperature")[0];
  var temperatureTextNode = temperatureNode.firstChild;
  var temperature = temperatureTextNode.nodeValue;
```

```
  var descriptionNode =
      responseXML.getElementsByTagName("description")[0];
  var descriptionTextNode = descriptionNode.firstChild;
  var description = descriptionTextNode.nodeValue;

  var descriptionClassNode =
      responseXML.getElementsByTagName("description_class")[0];
  var descriptionClassTextNode = descriptionClassNode.firstChild;
  var descriptionClass = descriptionClassTextNode.nodeValue;

  var weatherWidget = document.getElementById("weatherWidget"); (2)
  while (weatherWidget.hasChildNodes())
  {
    weatherWidget.removeChild(weatherWidget.firstChild);
  }

  var h2 = document.createElement("h2"); (3)
  h2.appendChild(document.createTextNode(name + " Weather"));
  weatherWidget.appendChild(h2);

  var div = document.createElement("div");
  div.setAttribute("id", "forecast");
  div.className = descriptionClass;
  weatherWidget.appendChild(div);

  var paragraph = document.createElement("p");
  paragraph.setAttribute("id", "temperature");
  paragraph.appendChild(
      document.createTextNode(temperature + "\u00B0C")); (4)
  div.appendChild(paragraph);

  var paragraph2 = document.createElement("p");
  paragraph2.appendChild(document.createTextNode(description));
  div.appendChild(paragraph2);
}
```

(1) The first four paragraphs of code inside `writeUpdate` parse the XML to get a particular tag value. As you can see, parsing XML can be quite tedious, but because we know the syntax of the data, we can go directly to the elements we need and extract their values fairly easily.

2. Right after we've finished parsing the data, we get a reference to the widget container and clean out its contents by removing all of its child nodes. This gives us an empty element into which we can insert our new data.

3. Using this clean slate, we can go about creating the new HTML elements that are going to represent the weather report. We'll use the data from the XML to create the relevant content.

4. If you have a sharp eye, you'll have noticed the peculiar text that we specified for the contents of `paragraph`. What the heck does `"\u00B0C"` mean, anyway?

   In fact, that will be displayed as °C (as in "It's a lovely 20°C outside"). The code `\u00B0` is a JavaScript character code for Unicode character number 00B0, which is the degree symbol (°).

   In an HTML document, you could just type the ° character verbatim, assuming you're hip to the whole Unicode thing and your HTML is written in UTF-8, or you could use the HTML character entity `&deg;`. JavaScript strings, on the other hand, only support Latin-1 (ISO-8859-1) characters in older browsers, and they don't support HTML character entities at all.

   So whenever you need to include in a JavaScript string a character that you can't easily type on an English keyboard (which means it may not be within the Latin-1 character set), your best bet is to look up its Unicode character number (using a tool like Character Map, which is built into Windows, or Mac OS X's Character Palette) and replace it with a `\uXXXX` code.

After we finish manipulating the DOM, the HTML of the weather widget will look roughly like this:

```
<div id="weatherWidget">
  <h1>
    Melbourne Weather
  </h1>
  <div id="forecast" class="partlyCloudy">
    <p id="temperature">
      18°C
    </p>
    <p>
      Fine, partly cloudy
```

```
    </p>
  </div>
</div>
```

The `class` on the `forecast` element enables us to style the weather forecast with a little icon, producing an updated widget that looks like Figure 8.4.

Figure 8.4. The weather widget with its updated content

Our little Ajax program is almost finished. All that's left is to handle the error that will be returned if our server doesn't return proper data:

weather_widget.js *(excerpt)*

```
  writeError: function(errorMsg)
  {
    alert(errorMsg);
  }
};
```

That one's really simple: the error message supplied to `writeError` is popped up in an alert box, letting the user know that something went wrong.

With the methods written, all we have to do is throw them together into one object, and initialize it with `Core.start`:

weather_widget.js

```
var WeatherWidget =
{
  init: function()
  {
    var weatherWidget = document.getElementById("weatherWidget");
    var anchors = weatherWidget.getElementsByTagName("a");

    for (var i = 0; i < anchors.length; i++)
    {
      Core.addEventListener(anchors[i], "click",
          WeatherWidget.clickListener);
    }
  },
  clickListener: function(event)
  {
    try
    {
      var requester = new XMLHttpRequest();
    }
    catch (error)
    {
      try
      {
        var requester = new ActiveXObject("Microsoft.XMLHTTP");
      }
      catch (error)
      {
        var requester = null;
      }
    }

    if (requester != null)
    {
      var widgetLink = this;
      widgetLink._timer = setTimeout(function()
          {
            requester.abort();

            WeatherWidget.writeError(
                "The server timed out while making your request.");
          }, 10000);
```

```
    var city = this.firstChild.nodeValue;

    requester.open("GET", "ajax_weather.php?city=" +
        encodeURIComponent(city), true);
    requester.onreadystatechange = function()
    {
      if (requester.readyState == 4)
      {
        clearTimeout(widgetLink._timer);

        if (requester.status == 200 || requester.status == 304)
        {
          WeatherWidget.writeUpdate(requester.responseXML);
        }
        else
        {
          WeatherWidget.writeError(
              "The server was unable to be contacted.");
        }
      }
    };
    requester.send(null);

    Core.preventDefault(event);
  }
},
writeUpdate: function(responseXML)
{
  var nameNode = responseXML.getElementsByTagName("name")[0];
  var nameTextNode = nameNode.firstChild;
  var name = nameTextNode.nodeValue;

  var temperatureNode =
      responseXML.getElementsByTagName("temperature")[0];
  var temperatureTextNode = temperatureNode.firstChild;
  var temperature = temperatureTextNode.nodeValue;

  var descriptionNode =
      responseXML.getElementsByTagName("description")[0];
  var descriptionTextNode = descriptionNode.firstChild;
  var description = descriptionTextNode.nodeValue;

  var descriptionClassNode =
      responseXML.getElementsByTagName("description_class")[0];
```

```
    var descriptionClassTextNode = descriptionClassNode.firstChild;
    var descriptionClass = descriptionClassTextNode.nodeValue;

    var weatherWidget = document.getElementById("weatherWidget");
    while (weatherWidget.hasChildNodes())
    {
      weatherWidget.removeChild(weatherWidget.firstChild);
    }

    var h2 = document.createElement("h2");
    h2.appendChild(document.createTextNode(name + " Weather"));
    weatherWidget.appendChild(h2);

    var div = document.createElement("div");
    div.setAttribute("id", "forecast");
    div.className = descriptionClass;
    weatherWidget.appendChild(div);

    var paragraph = document.createElement("p");
    paragraph.setAttribute("id", "temperature");
    paragraph.appendChild(
        document.createTextNode(temperature + "\u00B0C"));
    div.appendChild(paragraph);

    var paragraph2 = document.createElement("p");
    paragraph2.appendChild(document.createTextNode(description));
    div.appendChild(paragraph2);
  },
  writeError: function(errorMsg)
  {
    alert(errorMsg);
  }
};

Core.start(WeatherWidget);
```

And there you have it: a little Ajax weather widget that you could pop into the sidebar of one of your sites to let users instantly check the weather without leaving your page. Ajax offers endless possibilities; all you have to do is remember the pattern I described at the start of this example, and you'll have even the most complex interactions within your reach.

Seamless Form Submission with Ajax

As we saw in Chapter 6, forms are integral to the user experience provided by most web sites. One of the things Ajax allows us to do is to streamline the form submission process by transmitting the contents of a form to the server without having to load an entirely new page into the browser.

It's fairly simple to extend the Ajax code that we used in the previous example so that it can submit a form. Consider the contact form pictured in Figure 8.5, which uses this code:

contact_form.html *(excerpt)*

```
<form id="contactForm" action="form_mailer.php" method="POST">
  <fieldset>
    <legend>
      Contact Form
    </legend>
    <label for="contactName">
      Name
    </label>
    <input id="contactName" name="contactName" type="text" />
    <label for="contactEmail">
      Email Address
    </label>
    <input id="contactEmail" name="contactEmail" type="text" />
    <label for="contactType">
      Message Type
    </label>
    <select id="contactType" name="contactType">
      <option value="1">Enquiry</option>
      <option value="2">Spam</option>
      <option value="3">Wedding proposal</option>
    </select>
    <label for="contactMessage">
      Message
    </label>
    <textarea id="contactMessage" name="contactMessage"></textarea>
    <input id="contactNewsletter" name="contactNewsletter"
        type="checkbox" value="1" />
    <label for="contactNewsletter">
      I'd like to receive your hourly newsletter
    </label>
```

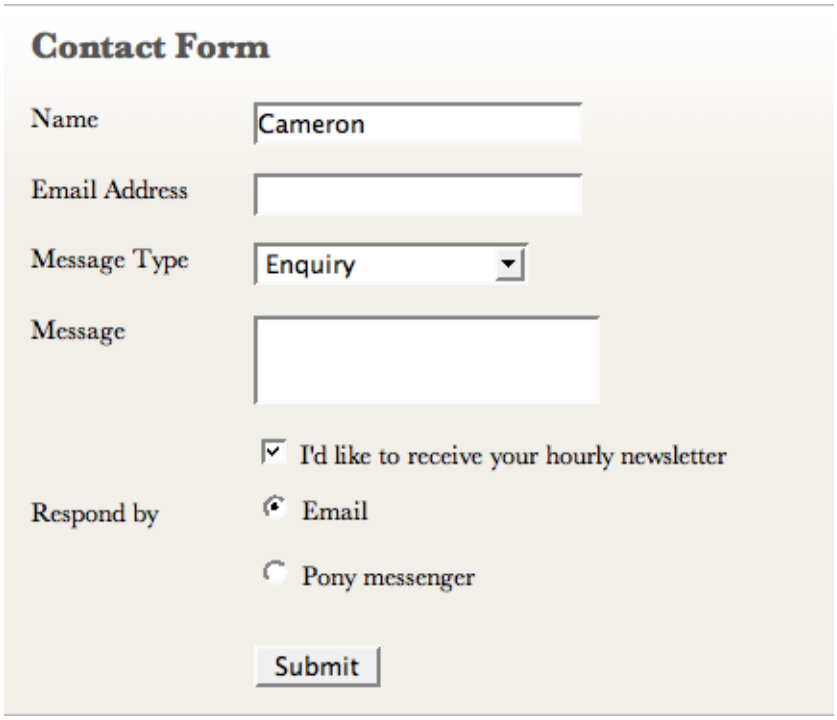


Figure 8.5. The example form to which we'll apply Ajax submission techniques

```
    <fieldset>
      <legend>
        Reply by
      </legend>
      <input id="contactMethodA" name="contactMethod" type="radio"
          value="1" />
      <label for="contactMethodA">
        Email
      </label>
      <input id="contactMethodB" name="contactMethod" type="radio"
          value="2" />
      <label for="contactMethodB">
        Pony messenger
      </label>
    </fieldset>
    <input type="hidden" name="id" value="SS56789" />
    <input type="submit" value="submit" />
  </fieldset>
</form>
```

In order to submit the form's contents using Ajax, we need to do a couple of things:

1. Override the default form submission behavior.
2. Get the form data.
3. Submit the form data to the server.
4. Check for the success or failure of submission.

We'll create this functionality inside an object called `ContactForm`. The only thing we need to do when we initialize the object is to override the form's default submission action. This can easily be done by intercepting the form's `submit` event with an event listener:

contact_form.js *(excerpt)*

```
var ContactForm =
{
  init: function()
  {
    var contactForm = document.getElementById("contactForm");
    Core.addEventListener(contactForm, "submit",
        ContactForm.submitListener);
  },
```

Ajax and Form Validation

Adding multiple event listeners to a given element for a given event can be risky business, because you have no control over the order in which they will be invoked. Thus, it isn't safe to assign an Ajax form submitter and a client-side form validator separately. If you do so, the submitter may be executed before the validator, and you might end up sending invalid data to the server. Link your validator and your submitter together to ensure that validation takes place before the form is submitted.

Now, before the form is submitted, the `submitListener` method will be run. It's inside this function that we collect the form data, send off an Ajax request, and cancel the normal form submission:

contact_form.js *(excerpt)*

```
submitListener: function(event)
{
  var form = this;

  try
  {
    var requester = new XMLHttpRequest();
  }
  catch (error)
  {
    try
    {
      var requester = new ActiveXObject("Microsoft.XMLHTTP");
    }
    catch (error)
    {
      var requester = null;
    }
  }

  if (requester != null)
  {
    form._timer = setTimeout(function()
        {
          requester.abort();

          ContactForm.writeError(
              "The server timed out while making your request.");
        }, 10000);

    var parameters = "submitby=ajax"; (1)
    var formElements = []; (2)

    var textareas = form.getElementsByTagName("textarea");

    for (var i = 0; i < textareas.length; i++)
    {
      formElements[formElements.length] = textareas[i]; (3)
    }

    var selects = form.getElementsByTagName("select");
```

```
for (var i = 0; i < selects.length; i++)
{
  formElements[formElements.length] = selects[i];
}

var inputs = form.getElementsByTagName("input");

for (var i = 0; i < inputs.length; i++)
{
  var inputType = inputs[i].getAttribute("type");

  if (inputType == null || inputType == "text" ||
      inputType == "hidden" ||
      (typeof inputs[i].checked != "undefined" &&
      inputs[i].checked == true)) 4
  {
    formElements[formElements.length] = inputs[i];
  }
}

for (var i = 0; i < formElements.length; i++) 5
{
  var elementName = formElements[i].getAttribute("name");

  if (elementName != null && elementName != "") 6
  {
    parameters += "&" + elementName + "=" +
        encodeURIComponent(formElements[i].value); 7
  }
}

requester.setRequestHeader("Content-Type",
    "application/x-www-form-urlencoded"); 8
requester.open("POST", form.getAttribute("action"), true); 9
requester.onreadystatechange = function()
{
  clearTimeout(form._timer);

  if (requester.readyState == 4)
  {
    if (requester.status == 200 || requester.status == 304)
    {
      ContactForm.writeSuccess(form); 10
    }
```

```
        else
        {
          ContactForm.writeError(
              "The server was unable to be contacted."); (11)
        }
      }
    };
    requester.send(parameters); (12)

    Core.preventDefault(event); (13)
  }
},
```

(1) One of the most important parts of the `submitForm` method is the variable `parameters`. This variable is used to store the **serialized** contents of the form—all of the field names and values combined into one long string that's suitable for sending in a POST request. We start this string off with the value `"submitby=ajax"`, which effectively adds to the form a variable named `submitby` with a value of `ajax`. The server can use this variable to identify form submissions that are transmitted via Ajax—as opposed to a standard form submission—and respond differently to them (for example, by sending the response in XML format rather than as a full HTML page).

(2) A form can contain a number of different elements, and we have to deal with all of them in order to produce a properly serialized version of the form's contents. There are three essential element types—`input`, `select`, and `textarea`—but `input` elements can have different types with different behaviors, so we have to cater for those as well. In order to minimize code repetition, we use the DOM to get node lists of each of the three element types, then we combine them into one big array—`formElements`—through which we can iterate in one fell swoop.

(3) All of the `textarea`s and `select`s can be added to `formElements` straight away, because those elements are always of the same type.

(4) When it comes to `input` elements, we have to distinguish between text `input`s, hidden `input`s, checkboxes, and radio buttons. Text `input`s and hidden `input`s can always be submitted with the form, because they don't have an on/off

toggle. However, we need to test whether checkboxes and radio buttons are `checked` or not before we add them to the list of submitted elements. We don't want to submit a value for a checkbox that wasn't checked, or for the wrong radio button in a group.

Inside the `for` loop that iterates over the `inputs` node list, we use a combination of each `input`'s `type` and its `checked` property to determine whether it should be added to `formElements`. The `if` statement uses a number of OR conditions to perform this check, and the logic reads like this:

1. IF the `type` of the `input` is `null` (it will be a text `input` by default)
2. OR the `type` of the input is `text`
3. OR the `type` of the input is `hidden`
4. OR the `input`'s `checked` property exists AND it is `true`
5. THEN add the `input` to `formElements`

The fourth point above catches both checkboxes *and* radio buttons. Any checkbox that's checked should have its value submitted, and only one radio button in a radio button group will ever have `checked` set to `true`, so it's safe to submit that one as well.

5. Once all the valid elements have been added to `formElements`, we have to write out their name/value pairs in a serialized fashion. This process is identical for all form element types, which is why we can minimize code repetition by building the `formElements` array in advance.

6. As we add each form element to the serialized string, we have to check whether a `name` is assigned to it.

7. If it *does* have a name, we'll want to send its value to the server, so we take the `name`, followed by `"="`, followed by the `value`, and add the whole thing to the end of `parameters`. Each of the name/value pairs in `parameters` is separated by an ampersand (`"&"`).

You'll notice that, again, we encode the names and values of our form elements using `encodeURIComponent`, to make sure that the request data remains valid no matter which special characters the user types into the form.

We're now ready to submit our serialized form data string to the server. The `XMLHttpRequest` code should be pretty familiar to you by now:

8. Remember that we have to set the `content-type` in the header of a POST request so that the request will work in Opera.

9. This time, when we `open` our `XMLHttpRequest` object, we'll use the `"POST"` method. The URL for the server request is pulled directly from the `action` of the form itself, which increases the reusability of this script.

10. In the `readystatechange` handler, all we're doing is waiting for a success code from the server. We don't actually care what data it gives us, so long as we know that it received the contact information. Once we've received that confirmation (in the form of a successful status code), we can execute `ContactForm.writeSuccess`, which will let the user know that the action was successful:

contact_form.js *(excerpt)*

```
writeSuccess: function(form)
{
  var newP = document.createElement("p");
  newP.setAttribute("id", "success");
  newP.appendChild(document.createTextNode(
      "Your message was submitted successfully."));
  form.parentNode.replaceChild(newP, form);
},
```

For this example, the success handler replaces the contact form with a paragraph that reads "Your message was submitted successfully," as shown in Figure 8.6.

11. Similarly, if the server does not successfully receive the contact information, `ContactForm.writeError` can handle the error in any manner it chooses—with a simple alert telling the user to try again, an error message, or even by shaking the browser window violently from side to side—whatever suits your applica-

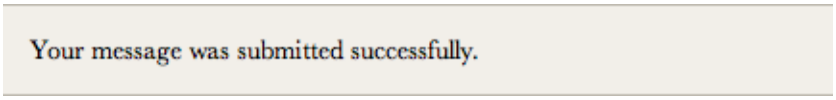


Figure 8.6. The message that appears when the form has been submitted successfully

tion best. For reasons of simplicity, and because it's highly visible to the user, this example uses an alert box:

contact_form.js *(excerpt)*

```
writeError: function(errorMsg)
{
  alert(errorMsg);
}
```

⓬ With that last method in place, the `readystatechange` handler is ready to do its work, and `submitListener` is free to fire off the Ajax request. Since this is a POST request, we pass `parameters` to the `send` method, rather than including them in the URL itself (as we would for a GET request).

⓭ The very last command in `submitListener` is a `Core.preventDefault` call, which stops the browser from submitting the form (as we just did it ourselves via Ajax). And that's why we call this program "seamless form submission."

We've generalized the code that handles form elements, which means that it's really easy to take this code and use it in other applications you might work on. Just modify the `init` method to reference the correct form, and away you go!

Exploring Libraries

Obviously, in this age of buzzwords, every JavaScript library must support Ajax in its own special way. As a result, almost every library out there has its own abstraction of the `XMLHttpRequest` object, which saves you from writing your own `try-catch` statements, supplying request variables in the right way depending on the type of request, and wiring up functions to handle different success and error conditions. Some of them even have handy shortcuts for common Ajax interactions, which can save you from writing code.

For each of the Prototype, Dojo, jQuery, YUI, and MooTools libraries, we'll translate this low-level Ajax code into the equivalent library syntax:

```
try
{
  var requester = new XMLHttpRequest();
}
catch (error)
{
  try
  {
    var requester = new ActiveXObject("Microsoft.XMLHTTP");
  }
  catch (error)
  {
    var requester = null;
  }
}

requester.open("GET", "library.php?dewey=005", true);
requester.onreadystatechange = readystatchangeHandler;
requester.send(null);

function readystatchangeHandler()
{
  if (requester.readyState == 4)
  {
    if (requester.status == 200 || requester.status == 304)
    {
      writeUpdate(requester);
    }
  }
}

function writeUpdate(requestObject)
{
  document.getElementById("container").childNodes[0].nodeValue =
      requestObject.reponseText;
}
```

That code steps through a fairly standard Ajax program:

1. Create a new `XMLHttpRequest` object.

2. Set up a GET connection to a server-side script.
3. Attach a request variable to the server call.
4. Send the data to the server.
5. Monitor the request for completion.
6. Insert the returned data into an HTML element (by setting the `nodeValue` of the text node it contains).

This program's standard functionality should give you a fair indication of the way that each library handles Ajax connections and interactions.

Prototype

Prototype's approach to handling Ajax calls basically represents the archetype for other libraries. It gives you access to an `Ajax` object, which offers a couple of methods with which to make requests.

We start a basic Ajax request by calling `new Ajax.Request` and passing it a number of parameters in the form of an object literal. When you specify an `onComplete` function, it will automatically be called once the server call has successfully completed:

```
var requester = new Ajax.Request("library.php",
    {
      method: "get",
      parameters: "dewey=005",
      onComplete: writeUpdate;
    });

function writeUpdate(requestObject)
{
  document.getElementById("container").childNodes[0].nodeValue =
      requestObject.reponseText;
}
```

That code can be further shortened by replacing `Ajax.Request` with `Ajax.Updater`. The second method assumes that you will place the contents of `responseText` directly inside an HTML element (the most common Ajax operation), and allows you

to specify the ID of that element. Thus, it circumvents the need for you to create your own callback function:

```
var requester = new Ajax.Updater("container", "library.php",
    {
      method: "get",
      parameters: "dewey=005",
    });
```

That's very succinct!

Dojo

To use Dojo's Ajax handler, we just call `dojo.io.bind` with an object literal that contains the appropriate parameters:

```
dojo.io.bind(
    {
      url: "library.php?dewey=005",
      load: writeUpdate,
      mimetype: "text/plain"
    });

function writeUpdate(type, data, event)
{
  document.getElementById("container").childNodes[0].nodeValue =
      data;
}
```

There are a couple of tricks with the Dojo Ajax implementation. Specifying `mimetype` inside the object literal determines what type of data Dojo will pass to your `load` function when the request is completed (text or XML). The `load` function receives three arguments:

`type` a superfluous variable that always has a value of `"load"`

`data` the only variable you'll actually use, as it contains the data from the server's response

event contains a reference to the low-level transport object that was used to perform the server communication (For the moment, it will inevitably be the `XMLHttpRequest` object.)

jQuery

As with everything in jQuery, Ajax functionality is available as part of the `$` object. `$.ajax` lets you specify an all-too-familiar object literal with the particular configuration you require for the call:

```
$.ajax(
    {
      type: "GET",
      url: "library.php",
      data: "dewey=005",
      success: writeUpdate
    });

function writeUpdate(data)
{
  document.getElementById("container").childNodes[0].nodeValue =
      data;
}
```

Again, with jQuery, as with Dojo, either `responseXML` or `responseText` will be passed directly to the `success` function—the property that's passed will depend upon the MIME type of the data returned from the server.

YUI

In true Yahoo! UI style, the name that Yahoo! has given to its Ajax object (which it calls a Connection Manager) is rather verbose, but in most other respects it's similar to what we've seen so far:

```
var handlers = {
  success: writeUpdate
}

YAHOO.util.Connect.asyncRequest(
    "GET",
    "library.php",
```

```
    handlers,
    "dewey=005"
);

function writeUpdate(requestObject)
{
  document.getElementById("container").childNodes[0].nodeValue =
      requestObject.reponseText;
}
```

The `handlers` variable allows you to specify both the handler function for a successful Ajax request, and the function to be called when an error occurs. These functions are passed a full `XMLHttpRequest` object, rather than just the data.

MooTools

MooTools based its Ajax handler on the one that comes with Prototype, so both handlers have similar syntax. MooTools' handler even has the shortcut for placing the returned data directly into an element without specifying a callback function:

```
var requester = new Ajax("library.php?dewey=005",
    {
      method: "get",
      onComplete: writeUpdate;
    });

requester.request();

function writeUpdate(requestObject)
{
  document.getElementById("container").childNodes[0].nodeValue =
      requestObject.reponseText;
}
```

The one difference between these two libraries is that the MooTools object doesn't automatically send the request once it has been initialized; you have to call `request` when you want to send it.

If you wish to use the element insertion shortcut, the code looks like this:

```
var requester = new Ajax("library.php?dewey=005",
    {
      method: "get",
      update: "container";
    });

requester.request();
```

The `update` property takes the ID of the element whose contents you wish to update.

Summary

Ajax is definitely here to stay, and as users and developers become accustomed to its behavior, the number of ways in which it will be used to enhance web interfaces will only increase.

As you've seen in this chapter, the actual communication mechanism of Ajax is relatively straightforward. The pattern of *initialize-retrieve-modify* draws heavily on all the techniques that you've picked up as you've worked your way through the preceding seven chapters, so Ajax is the perfect finishing point for all the practical work in this book. But read on to find out where the future of JavaScript might lead you…

Chapter 9

Looking Forward

When you first picked up this book, you were undoubtedly aware of JavaScript's meteoric rise in (or return to) popularity over the past couple of years. Interest in JavaScript—and its usage on the Web—is now at its highest point ever, and it's only going to keep rising.

However, for all its popularity, JavaScript is still very immature—a truth that applies to many aspects of the Web. New things are being learned about JavaScript each day, and new ideas are springing up from every corner of the globe. JavaScript is a language that's yet to reach its full potential.

It wasn't so long ago that developers could forget about being unobtrusive! *Everything* was obtrusive: we used inline event handlers in our HTML, content was being inserted with `document.write`, and we had to provide a different version of our code for every browser. How quickly we've moved on, created new rules, and almost invented a new language. The JavaScript you've learned in this book looks very different from the JavaScript that was written only a few years ago.

Some people might see this immaturity as a bad thing, but I embrace it. Why? Because it's exciting. JavaScript is on the edge. With JavaScript, you can do things

that people have never done before. There are countless new ideas to be explored, and new pages to be forged.

Hopefully, all the reading you've done so far has made you as excited as I am, and you've got a million and one ideas buzzing around inside your head. You should be thinking of all the ways you can use JavaScript on your pages to create a usable and fun experience for your users.

But if, by some strange eventuality, your JavaScript juices aren't quite flowing yet, this peek at the future is sure to get them going like the Niagara.

Bringing Richness to the Web

Maybe you thought Flash brought richness to the Web. It certainly did its bit, changing the Internet from a static, page-based paradigm to a morphing, transitioning, all-singing, all-dancing "experience." But it tends to throw out the benefits of HTML, while introducing some of its own disadvantages.

JavaScript—with HTML as its foundation—manages to strike a happy medium between the interactivity of Flash and the accessibility of HTML, especially if you follow the tenets of progressive enhancement that we've been advocating in this book. If you think of JavaScript as an add-on to HTML, it makes the transition from a page-based development model to a more interactive model much less turbulent.

Although a lot of the recent focus of interaction design has been placed on Ajax, it's not the only way in which you can create a more usable interface. Sometimes, you just need to think about how you can represent your interface in a different manner.

Easy Exploration

Take, for example, the Travelocity site.[1] Its creators recognize that when you're looking for accommodation, there are normally one or two variables that you want to use to narrow down your options. To let you get feedback quickly on the effect that your choices have made, they implemented the variables as JavaScript sliders,

[1] http://www.travelocity.com/

as shown in Figure 9.1. These sliders allow you very easily to specify your accepted range for the variables, and immediately see how many hotels meet your criteria.

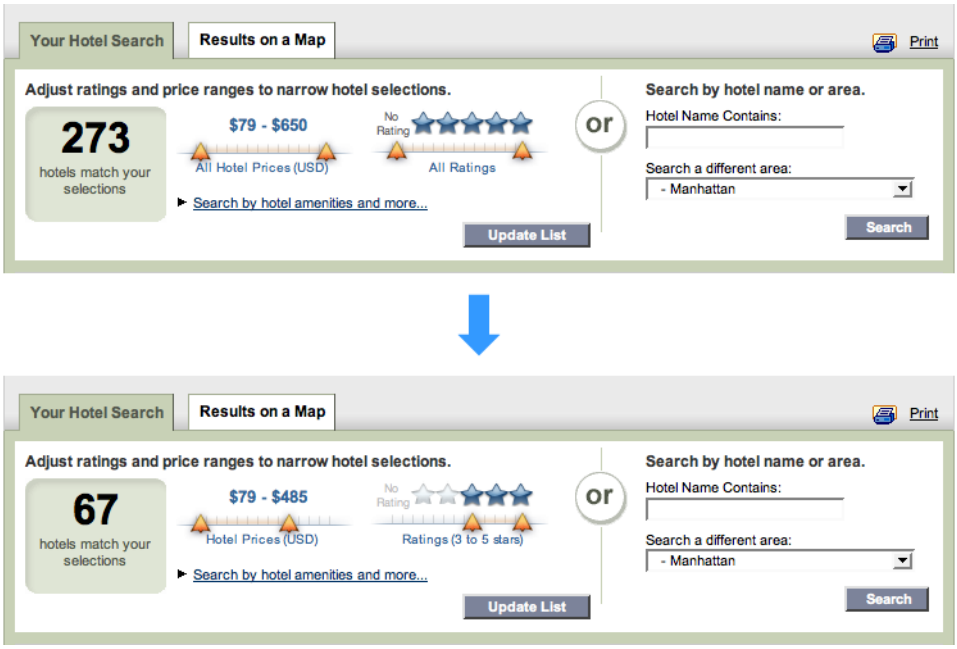


Figure 9.1. The JavaScript sliders on Travelocity providing immediate feedback

If you wanted to implement this sort of feature using nothing but server-side calculation, obtaining the correct data would take an impractical number of form submissions. The site's users would probably either give up before discovering exactly what they wanted, or settle for accommodation that wasn't really what they wanted.

By having this easy-to-use interface element, the site actually encourages users to explore all of the options, making them much happier throughout the experience, and giving them a greater chance of finding ideal accommodation matches.

Easy Visualization

Among all of the Ajax voodoo used on Flickr,[2] one of my favorite pieces of functionality is the inline editing capability shown in Figure 9.2. This type of interaction tool serves two purposes. Firstly, the use of an Ajax call to update information

[2] http://www.flickr.com/

means that users' changes are applied to pages very quickly. Helping this impression of speed, only small amounts of data are being sent (which is important on Flickr's sometimes slow-loading pages), and the Gallery view makes it easy to make bulk changes to multiple photos.

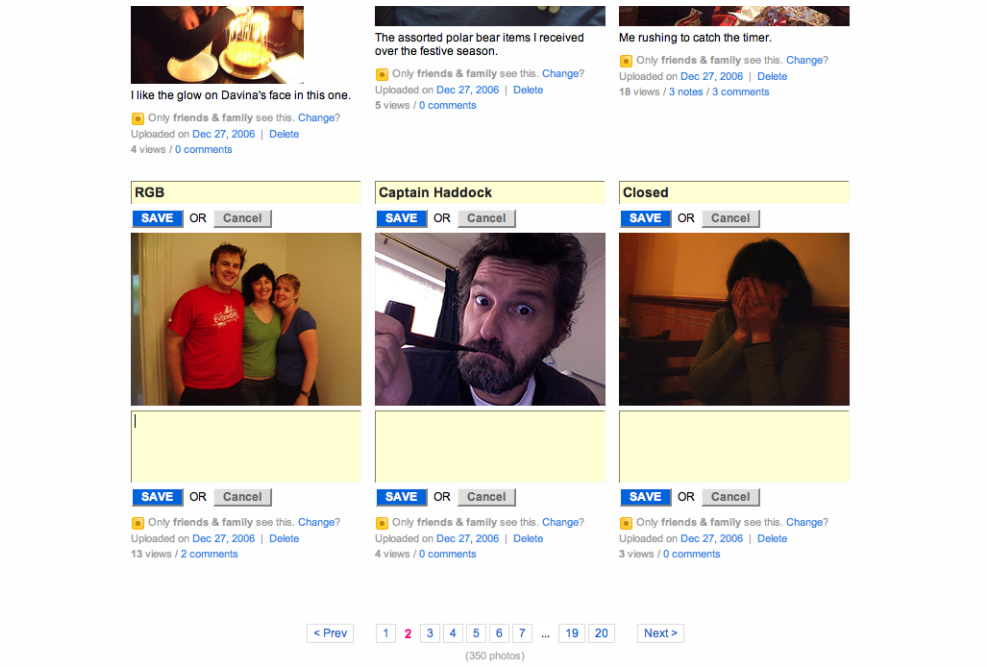


Figure 9.2. Inline editing areas eliminating full page loads on Flickr

The other great thing about inline editing is that you can be browsing your photo gallery pages, spot something that you want to change (like the typo in Figure 9.3), and change it right there and then. You get to see the data exactly as it appears to users, rather than in a form that's not at all connected with the way the data will be finally presented. Thus the usual disconnect between administration and presentation vanishes.

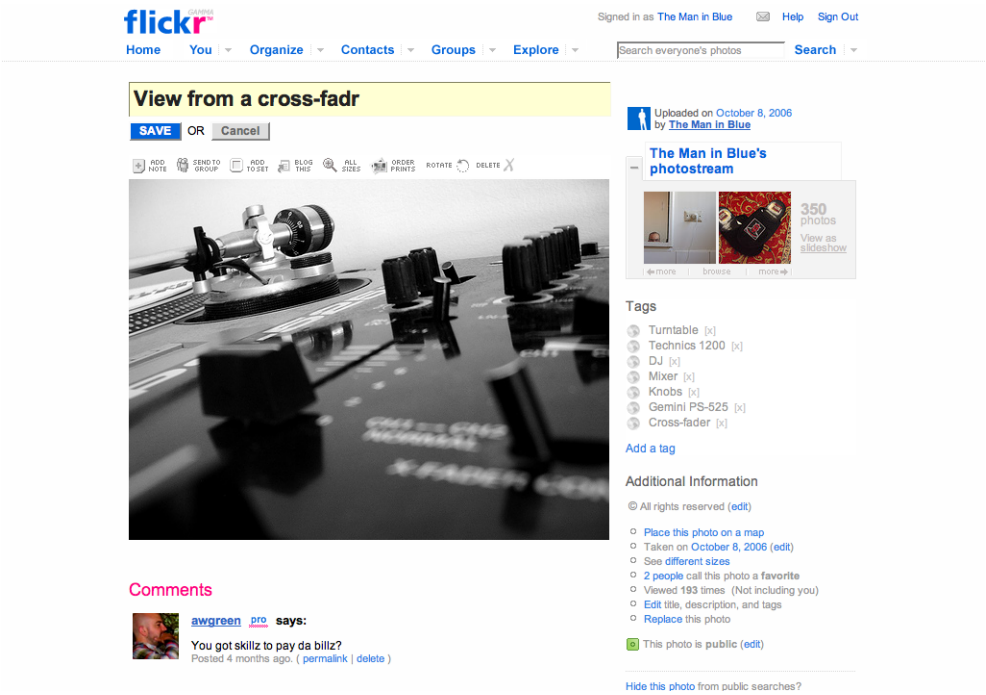


Figure 9.3. Editing data inline, just as it appears on your pages

Unique Interaction

You can perform a million and one usability tweaks with JavaScript, but they still essentially function like static HTML—with a bit less waiting. By far the most forward-thinking—and, let's face it, *sexier*—examples are those that use the interactive capabilities of JavaScript to create something that's not possible using traditional client-server techniques.

One of my favorite examples of this is Meebo, which is shown in Figure 9.4.[3] It (and its web-based IM brethren) take the originally desktop-based instant messaging applications and move them onto a freely available web-based platform, thereby multiplying their usefulness by a factor of approximately 15.[4]

With IM available through your browser, maintaining contact with your friends becomes as easy as blinking. Having an entirely browser-based chat client means

[3] http://www.meebo.com/

[4] Note: author's personal estimation only. Real results may differ from those shown on the pack. —Ed.

you don't have to install any software (an impossibility on many corporate systems), and you can switch workstations and still use the service—without having to transfer your IM account details. These services are also pretty good at beating firewalls that are set up to block IM ports. (I'm not sure whether this is counted as a good thing for employers, though!) Without JavaScript, this type of complex interface would be impossible—it would simply take too long using a normal page request architecture.

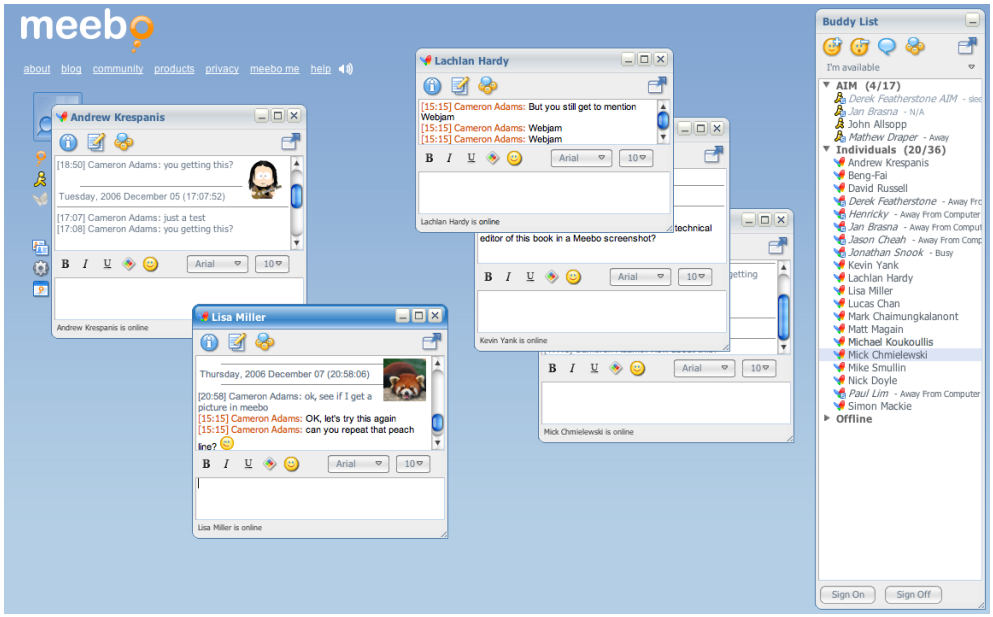


Figure 9.4. Meebo's JavaScript- and Ajax-based, in-browser instant messaging client

If you want to take a bigger step into the future, it's possible to combine JavaScript with emerging vector-rendering standards that allow you to create you own *really* customized interfaces. Brand new at the time this book was written was the Yahoo! Pipes service shown in Figure 9.5.[5] It uses the `canvas` element (which is available in quite a few browsers) to create a unique interface that's best suited to its own particular functionality: wiring together an unlimited number of data modules.

[5] http://www.pipes.yahoo.com

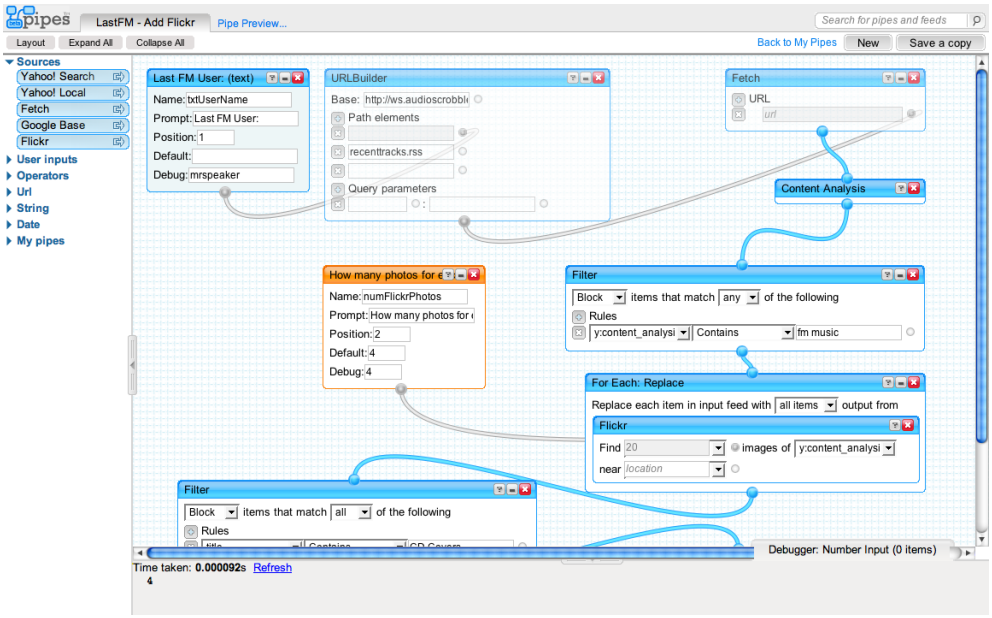


Figure 9.5. Yahoo! Pipes's intuitive and unique interface

canvas and its cousin SVG give designers the ability to create lightweight, highly adaptable vector graphics for use in web interfaces. Combined with the interactivity of JavaScript, these technologies are finally giving web applications the power that their desktop cousins have enjoyed for so many years, thereby removing one of the biggest barriers to the proliferation of a totally web-based application system.

And of course it's not all business suits and briefcases on the side of JavaScript; there's the potential to let your mind wander and produce some frivolous but fun entertainment of the kind for which Flash has earned a reputation—my Bunny Hunt game is a case in point.[6] The sky's the limit!

[6] http://www.themaninblue.com/experiment/BunnyHunt/

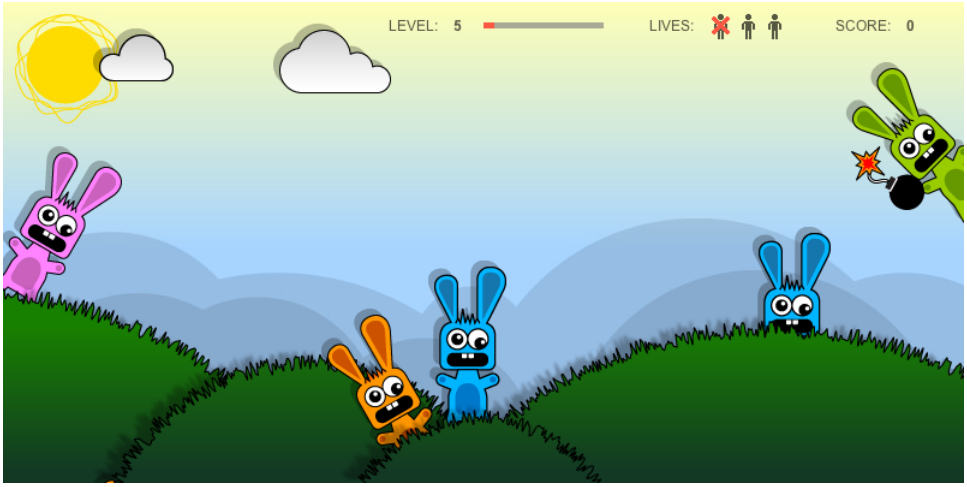


Figure 9.6. Bunny Hunt, a game whose style you'd normally associate with Flash

Rich Internet Applications

As you might have noticed as you read the previous section, the introduction of JavaScript and other new technologies is promoting a shift in the Web's focus. Previously, web sites were largely restricted to storing information, and offered very little interactivity, apart from the ubiquitous form fields.

As JavaScript makes more and more desktop-style interactions possible, we're beginning to see the Web transformed into an extension of the desktop. And eventually, who knows? We may lose the desktop altogether.

These next-generation web applications are distinguished from their static ancestors by the name **Rich Internet Applications** (RIAs). The important distinction here is made by the term *rich*, which refers to the styles of interaction that are available through these applications: expanding/collapsing, seamlessly updating, auto-completion, drag-and-drop, customized gestures—the list goes on.

In RIAs, most user interface actions are driven from the client side, which makes them much more responsive and flexible, because no time-consuming communication has to be performed with the server to complete these types of operations. Examples of RIAs include most of Google's recent applications[7] (Maps, Mail, Calen-

[7] http://www.google.com/intl/en/options/

dar—shown in Figure 9.7—and Docs & Spreadsheets), Meebo,[8] Netvibes,[9] and Zimbra.[10]

Many of these applications focus on taking functionality that currently exists on your desktop and moving it onto the Web, literally making your data available to you wherever you may be in the world. But as RIA development matures, we'll begin to see more applications that have no desktop counterpart—and never will—such as Yahoo! Pipes.

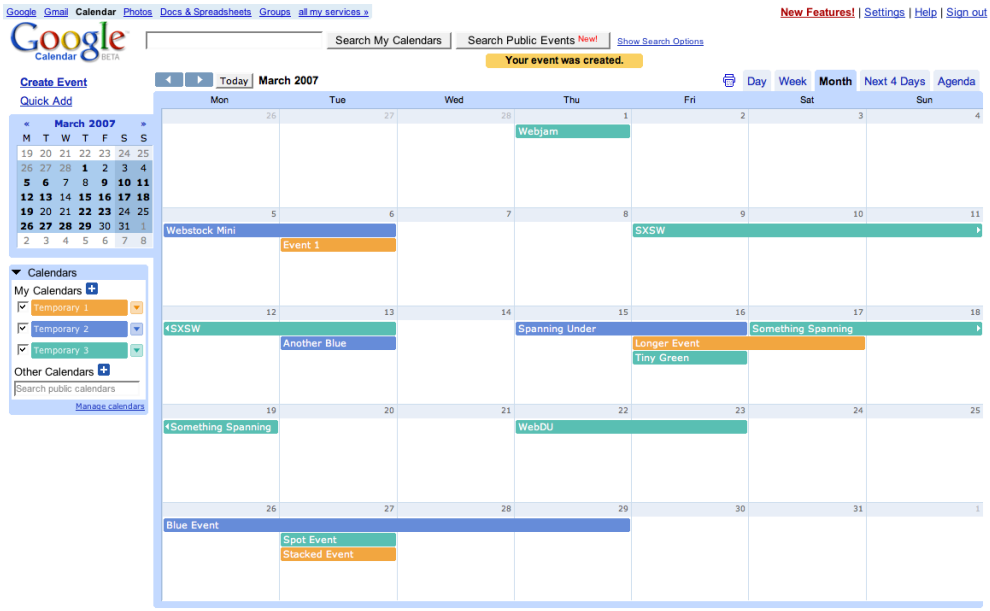


Figure 9.7. The Google Calendar interface

The main problem with RIAs is that they're extremely complex to develop—probably even more complex than their desktop equivalents. This complexity is the result of three factors, which have a lot to do with the nature of the Web itself:

- The interface is subject to display in a browser, which entails all the browser quirks and incompatibilities that are normally associated with web development.

[8] http://www.meebo.com/
[9] http://www.netvibes.com/
[10] http://www.zimbra.com/

- The behaviors required to perform much of the interaction don't natively exist on the Web or in a browser; they have to be created from the ground up using JavaScript. So if you want drag-and-drop capabilities, you'll have to include a script that creates this functionality.

- The Web was designed to be stateless, yet this clashes with the way a lot of applications are designed to work. Applications often rely upon a certain sequence of events occurring in a given order, but this order can't be guaranteed on the Web—a place where users can jump in or out at any point they choose, and can modify data without the server knowing.

One fundamental issue underlies all of these problems: the Web wasn't designed to support applications.

This fact is highlighted by the way that current accessibility technology handles RIAs: badly. Since assistive technology has been designed to work with a page-based model, the new micro-updates that lie at the heart of an RIA throw tools like screen readers into confusion—they don't know when the page has been updated, and even if they did, they wouldn't know how best to alert a user to those updates.

The blame can't be laid solely on the screen reader manufacturers, though. The structure that they're trying to translate has some inherent accessibility flaws when it comes to describing applications—flaws that can't just be patched. The thing is, browsers that render HTML have become so ubiquitous that it makes sense to create applications that take advantage of them. We just have to hack our way around all the challenges somehow. At least, that's one point of view. The other is to remodel or even recreate HTML with more application-oriented functionality.

`canvas` is one of the first steps in this direction. Introduced by the Web Hypertext Application Technology Working Group (WhatWG)—a community organized towards the development of a more coherent web application standard—`canvas` is just a small part of a greater plan that aims to create a developer-friendly, user-advantageous system of application delivery. However, its full potential is yet to be realized, and even after all the standards have been locked in, the standard's roll-out to a majority of browsers will take time.

For the moment, it looks like we're stuck using the sometimes clunky, always complex combination of HTML, CSS, and JavaScript for web application develop-

ment. It's with this fact in mind that quite a few people are trying to find the most viable way of easing the pressure.

If JavaScript follows the path of most other programming languages—which, judging from the rise in libraries, it most likely will—it seems inevitable that popular application functionality will become invested in a few frameworks that will help ease the burden of the first two points outlined above. It remains to be seen whether or not the third problem can be solved.

Widgets

A number of common themes run through the design of a lot of applications. On the desktop, these functional commonalities include tasks like file handling and window control, and tools such as dialog boxes, toolbars, and menus. You can use a range of these standard components to build desktop applications, so you don't have to reinvent the wheel every time you create a new application. By using these components, you take advantage of established conventions that aid usability—users already know what a menu looks like, how a dialog box works, and what they should do with a button.

Parallels are starting to be drawn in application design on the Web. A few interactions are now becoming sufficiently common to warrant the creation of reusable components, or **widgets**, that, when dropped into a page, *just work*. Whereas previously these widgets might have been offered piecemeal across a dozen sites spread over the Web, now they're beginning to be aggregated in large libraries, to become what you might call **component frameworks**.

These frameworks don't provide just a bit of abstracting code that addresses specific browser differences; they offer an entirely new mode of development. Instead of writing all the HTML, all the styles, and all the JavaScript for a particular piece of functionality, you can just include a widget—maybe just one line of code—and your application will automatically boast the functionality that you once had to write hundreds of lines of code to achieve.

Frameworks don't just make coding easier, however; they also have an effect upon the language itself. If you start substituting one line of Dojo for a hundred lines of JavaScript, are you writing JavaScript, or are you writing Dojo?

JavaScript Off the Web

As the popularity of web development techniques continues to surge, JavaScript has become quite a powerful force in software development. The separation of structure, style, and behavior that is so strictly expressed in web standards is actually an ideal platform for the development of applications and widgets both on the Web, and off it.

With the introduction of Dashboard Widgets, a sampling of which is shown in Figure 9.8, Apple's Mac OS X brought the combination of HTML, CSS, and JavaScript onto the desktop, making it a much easier proposition for people to create their own native applications. The masses of people who had cut their development teeth on the pages of the Web were now able to port those skills directly to the desktop. And the growth in the number of widgets has revealed the ease with which this skills transfer can be achieved.

Figure 9.8. Creating mini-applications using HTML, CSS, and JavaScript with MacOS X widgets

Alongside MacOS X, Yahoo! released its own desktop widget tool; they've just been joined by the latest incarnation of Windows—Vista—which supports similar items, called "gadgets."

Aside from widgets, it's possible to create entire applications using JavaScript. One of the browsers with the fastest growing market share—Firefox—was built entirely with JavaScript; an application development language called XUL was used to provide the user interface elements. Again, the separation of structure, style, and behavior here allows the application to easily be skinned via CSS, and invites the addition of new functionality—contained in Firefox extensions—via JavaScript.

Exploring Libraries

The number of JavaScript libraries available on the Web is gradually being whittled down predominantly on the basis of how much they're used, but their endurance is also affected by factors such as functionality, ease of integration, and available documentation.

To my mind, a library can go in two directions—small and light, or big and heavy. There are some advantages to keeping a library small—especially on the Web, where download time is still an issue, along with client-side processing power and code complexity. Libraries such as jQuery[11] and moo.fx[12] fill that niche neatly, providing an efficient pocket knife of JavaScript functions that will help you get the job done.

At the other end of the scale, the big libraries are becoming even bigger. More and more functionality is being absorbed into their bellies, in an effort to encompass any and all types of interaction that can be imagined to take place in a browser. These libraries are becoming less like libraries from which you can pick and choose the elements you want, and more like frameworks, where the library becomes the language, and your programs become distilled into minimalistic lines of API code.

The various styles of these large frameworks differ. Some try to maintain the separation of layers that has characterized web development for some time, while others attempt to abstract the web layer entirely, by creating a translation between server-side logic and browser implementation. To cover the specifics of each of these

[11] http://jquery.com/
[12] http://moofx.mad4milk.net/

frameworks would take many more pages than we have available, but I'll try to give you a tiny taste of how they function here.

Dojo

We've taken a decent look at Dojo throughout this book, but only as a library, not a framework.[13] Dojo offers more than just helper functions to shortcut common scripting actions—it actually provides an entire system for developing interfaces using common widgets.

The widgets currently available in Dojo number well over a hundred and include such items as accordion menus, combo boxes, date pickers, fish-eye lists, progress bars, sliders, and tree menus. The framework also has its own layout system, which allows you to create panes with related dimensions (such as equal heights) and adjustable dimensions—something that's quite common in desktop application design.

We can implement widgets in Dojo in two ways: structurally and programmatically. If you use the structural method, you must modify the HTML structure of your page in order to include the Dojo widgets you want to use. To achieve this, you tack non-standard attributes onto particular HTML elements—a big no-no if web standards are an important consideration to you.

The programmatic creation of Dojo widgets is most useful when you want to create new interface elements on the fly, or you don't want to dirty your HTML with attributes that aren't meant to be there. You create the new widget inside your JavaScript, then include it on the page via the DOM.

Following the tenets of progressive enhancement, a lot of Dojo widgets try to use HTML elements that provide the same functionality, then transform them into more complex objects. For example, a `select` box can be turned into a combo box (a box that allows you to type and receive suggestions) with the addition of some extra attributes:

```
<select dojoType="ComboBox" dataUrl="example.php" name="state">
</select>
```

[13] http://dojotoolkit.org/

Next, include that element on an HTML page along with the Dojo library and `ComboBox` module:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en-US">
  <head>
    <title>ComboBox</title>
    <meta http-equiv="Content-Type"
        content="text/html; charset=utf-8" />
    <script type="text/javascript" src="dojo.js"></script>
    <script type="text/javascript">
      dojo.require("dojo.widget.ComboBox");
    </script>
    ⋮
```

The result is a custom form element that provides users with suggestions as they type, like the one shown in Figure 9.9.

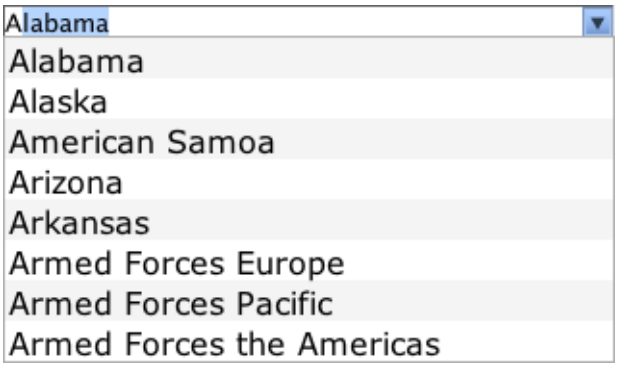


Figure 9.9. Using the Dojo `ComboBox` widget to transform a select element into a field that auto-completes as users type

When the page is parsed by Dojo, the `dojoType` attribute is used to determine which elements are to be transformed into widgets. Then, the Dojo engine will go about changing the page to include the specified widgets.

If you wanted to include the `ComboBox` programmatically, you could write a script that creates a `ComboBox` widget, then adds it to the page:

```
var comboBox = dojo.widget.createWidget("ComboBox",
    {
      dataUrl: "../../tests/widget/comboBoxData.js"
    });
document.getElementsByTagName("body")[0].appendChild(
    comboBox.domNode);
```

New widgets are created using the `dojo.widget.createWidget` method, which takes the name of the widget, followed by a set of options expressed as an object literal. In this case, the only option we're supplying to the `ComboBox` widget is the `dataUrl` location.

Once that widget has been created, we'll have a reference to the widget as a JavaScript object, but in order to include it on the page, we'll have to use its `domNode` property, which acts as the HTML interface for the actual object. And once that's done, we'll have a working `ComboBox` widget in our interface.

In the example above, the `ComboBox` is actually Ajax-driven—it will use the URL specified by the `dataUrl` property to retrieve listings that match what the user types. However, this explanation highlights a flaw in the way that this widget uses progressive enhancement.

When using a `select` box, users without JavaScript would expect to click on the box, and receive a number of `options` from which they can choose. The only way that this can be done is if all the `options` are included in the HTML. One of the aims of the `ComboBox` is to *reduce* the amount of data transferred between client and server by only retrieving the data that matches what the user is typing. However, leaving the `select` box empty initially renders it useless to users who don't have JavaScript enabled. This is not progressive enhancement. Then again, if we included all the `options` for the `select` box in the HTML, we'd be removing one of the main advantages of the `ComboBox`.

Maybe a `ComboBox` could be thought of more like a plain text field (although Dojo's `ComboBox` widget doesn't work with text fields, so this is a purely academic argument). Non-JavaScript users would be able to type whatever value they wanted into the field, but they wouldn't get a list of possible values.

These questions highlight some of the problems that are being caused by Rich Internet Applications. JavaScript and HTML give us a great deal of flexibility to create whatever we want, but sometimes that flexibility comes at the cost of accessibility. Is there a better answer? Only time will tell.

Google Web Toolkit

The Google Web Toolkit (GWT) uses Java as the basis for its engine.[14] You never have to write JavaScript, HTML, or CSS if you don't want to. Using GWT's API, you can create entire applications in Java. The GWT compiler translates all of the Java into browser code (HTML, CSS, and JavaScript) for inclusion in your web site.

This style of framework makes web application development similar to traditional application development—all the code is written by the developer, then compiled for execution. In GWT's case, compilation includes the creation of HTML and JavaScript files that will help run your application in a browser.

This framework is radically different from anything we've looked at in this book, but as an example, this Java class is used to create a page with a button that launches an alert box saying, "Hello, JavaScript:"

```
public class Hello implements EntryPoint {

  public void onModuleLoad() {
    Button b = new Button("Click me", new ClickListener() {
      public void onClick(Widget sender) {
        Window.alert("Hello, JavaScript");
      }
    });

    RootPanel.get().add(b);
  }

}
```

As well as being a very quick way for Java developers to create web applications, GWT is also extremely inaccessible. No static HTML is provided as the structure for the application, and every interface element is created via JavaScript. Therefore,

[14] http://code.google.com/webtoolkit/

there is essentially no document for the browser to fall back on if JavaScript is not available. The unlucky user gets zip, nada, nothing. If this isn't a worry for you, you might want to give GWT a try. But, personally, it scares the pants off me.

Summary

The future is bright for JavaScript. Its dominance as the client-side language of choice on the Web has been unchallenged for well over ten years, and as long as HTML (or something like it) exists, JavaScript will be there to partner it.

Yet the question remains, what will the future hold for JavaScript? It's flexible enough to adapt to a range of environments—as we've discussed in this chapter—and its ultimate form will largely be dependent upon which technology makes its ascendance. But no matter which one rises to the top of the heap, I'm pretty certain that JavaScript will be there right alongside it, making it do all the *really* cool stuff.

Appendix A: The Core JavaScript Library

Most of the examples in this book have relied on a small library of useful methods contained in a file named **core.js**. This library enabled us to set aside the messy details of some of the common tasks you need to perform in most scripts, and focus on the more unique aspects of each example.

In some cases, this library's methods were a little too complicated to explain in detail at the moment we first needed them. This appendix, therefore, contains a complete description of the Core library, and how each of its methods works.

The Object

Like most of the scripts in this book, the Core library encloses all of its functions within a JavaScript object, making them methods of that object. In the past, we have always enclosed functions within a JavaScript object using an object literal:

```
var Core = {
  method1: function(…)
  {
    ⋮
  },
  method2: function(…)
  {
    ⋮
  },
  ⋮
};
```

While the Core library has the same basic structure as the code shown above, the code that we've used to define the object and its methods uses different syntax:

```
var Core = {};

MyObject.method1 = function(…)
{
  ⋮
};
MyObject.method2 = function(…)
```

```
{
  ⋮
};
```

The object that's produced here is identical to the first; the syntax is just a little more verbose.

The reason we use this alternative style for the Core library is that it gives us the freedom to define different versions of each method based on the browser in which the script is running—essential for compatibility mine-fields like event listeners:

core.js *(excerpt)*

```
var Core = {};

// W3C DOM 2 Events model
if (document.addEventListener)
{
  Core.addEventListener = function(target, type, listener)
  {
    ⋮ W3C DOM 2 Events implementation
  };
  ⋮
}
// Internet Explorer Events model
else if (document.attachEvent)
{
  Core.addEventListener = function(target, type, listener)
  {
    ⋮ Internet Explorer Events implementation
  };
  ⋮
}
```

Event Listener Methods

As we saw in Chapter 4, there are two very different models for implementing event listeners in current browsers: the Internet Explorer Events model, supported by IE browsers up to and including version 7, and the W3C DOM 2 Events model, suppor-

ted by every other browser out there. The Core library bridges the gap by providing a set of methods that will work in both of these models:

`Core.addEventListener(target, type, listener)`
: For the object *target* (usually a DOM element node), this method assigns the function/method *listener* as an event listener for events of type *type*. For example:

```
Core.addEventListener(theLink, "click", MyScript.clickListener);
```

`Core.removeEventListener(target, type, listener)`
: For the object *target* (usually a DOM element node), this method removes the function/method *listener*, previously added as an event listener for events of type *type*. Here's an example:

```
Core.addEventListener(theLink, "click", MyScript.clickListener);
```

`Core.preventDefault(event)`
: This method prevents the browser from performing the default action associated with the event represented by *event*, an event object passed as an argument to an event listener.

`Core.stopPropagation(event)`
: This method prevent event listeners attached to elements higher up in the DOM tree from being triggered in response to the event represented by *event*, an event object passed as an argument to an event listener.

Thanks to the code structure discussed in the previous section, we are able to implement these event listener methods separately for each of the two event models.

The W3C DOM 2 versions of the event listeners are trivial wrappers around the standard methods:

core.js *(excerpt)*

```
// W3C DOM 2 Events model
if (document.addEventListener)
{
  Core.addEventListener = function(target, type, listener)
  {
```

```
    target.addEventListener(type, listener, false);
  };

  Core.removeEventListener = function(target, type, listener)
  {
    target.removeEventListener(type, listener, false);
  };

  Core.preventDefault = function(event)
  {
    event.preventDefault();
  };

  Core.stopPropagation = function(event)
  {
    event.stopPropagation();
  };
}
```

The Internet Explorer versions are much more complicated, so we'll look at them one at a time. Fundamentally, however, these functions make the following enhancements to Internet Explorer's built-in event listener functionality:

- Prevent the same function/method from being assigned as a listener for the same event on the same element more than once.
- Pass Internet Explorer's global `event` object as an argument to the event listener.
- Call event listeners in such a way that, within the listener, `this` represents the object to which the listener was assigned.
- Clean up all registered event listeners when the document is unloaded, so as to prevent memory leaks in Internet Explorer.

The code for these functions is based upon a script presented in the book *JavaScript: The Definitive Guide, 5th Edition*.[1]

`addEventListener` is the most complex of the methods, so let's take it one step at a time:

[1] David Flanagan, *JavaScript: The Definitive Guide, 5th Edition* (Sebastopol: O'Reilly, 2006).

core.js *(excerpt)*

```
Core.addEventListener = function(target, type, listener)
{
  // prevent adding the same listener twice, since DOM 2 Events
  // ignores duplicates like this
  if (Core._findListener(target, type, listener) != -1) return;
```

We start by checking if the specified combination of target object, event type, and listener function has already been registered in this document. To do this, we use a private method named `_findListener`, the implementation of which we'll look at shortly. If the listener has already been registered, we simply do nothing, and return from this method immediately.

Now, we can't go using the listener function that was passed to `addEventListener` as is. If we did, Internet Explorer wouldn't pass it the event object as an argument, and `this` would refer to the global context within the function, rather than referring to `target`. To resolve these issues, we can wrap the supplied listener inside a function that makes the necessary changes:

core.js *(excerpt)*

```
  // listener2 calls listener as a method of target in one of
  // two ways, depending on what this version of IE supports,
  // and passes it the global event object as an argument
  var listener2 = function()
  {
    var event = window.event; (1)

    if (Function.prototype.call)
    {
      listener.call(target, event); (2)
    }
    else
    {
      target._currentListener = listener; (3)
      target._currentListener(event)
      target._currentListener = null;
    }
  };
```

1. As you can see, `listener2` is a function that starts by retrieving the global `event` object. It then calls `listener` in such a way as to make `target` the value of `this`, and passes the event object to it as an argument. This is done in one of two ways, depending on the version of Internet Explorer that's running the script.

2. In Internet Explorer 5.5 or later, all functions support a method named `call`, which allows the function to be called as if it were a method of a specified object. If this function is available (which we can test by checking for the presence of `Function.prototype.call`), we can use it to call `listener` as a method of `target`, and with `event` as an argument.

3. In previous versions of Internet Explorer, the `call` method is not available, so instead we store `listener` in a temporary property of `target` named `_currentListener`, which enables us to call `listener` as a method of `target`. Again, we pass `event` as an argument. Once that's done, we set `_currentListener` to `null`.

Now that we've got our pimped-out `listener2`, we can use Internet Explorer's `attachEvent` method to register it as an event listener:

core.js *(excerpt)*

```
// add listener2 using IE's attachEvent method
target.attachEvent("on" + type, listener2);
```

Next, we need to do a little bookkeeping to ensure that the `_findListener` method we called at the top of `addEventListener` can tell that this listener has been registered. We'll create an object that contains all the pertinent information about the listener that we have just added:

core.js *(excerpt)*

```
// create an object describing this listener so we can clean
// it up later
var listenerRecord =
{
  target: target,
  type: type,
```

```
    listener: listener,
    listener2: listener2
  };
```

For `_findListener` to do its job, we could just add this object to an array stored as a property of `target`. But as long as we're keeping records of our event listeners, let's set things up so that it's easy to clean up *all* the listeners in the document when the page is unloaded. To this end, we'll store our `listenerRecord` in an object that's shared by *all* the listeners in the document:

core.js *(excerpt)*

```
// get a reference to the window object containing target
var targetDocument = target.document || target; (1)
var targetWindow = targetDocument.parentWindow;

// create a unique ID for this listener
var listenerId = "l" + Core._listenerCounter++; (2)

// store a record of this listener in the window object
if (!targetWindow._allListeners)
  targetWindow._allListeners = {}; (3)
targetWindow._allListeners[listenerId] = listenerRecord; (4)
```

(1) Since we want to have one object containing all the listeners in the window, we need to get a reference to that window. To do that reliably, we need to start by getting a reference to the current document.

Now, `target` may be either an element in the document or the document itself, so if `target.document` exists we'll use that as our document; otherwise, we can just assume that `target` is the document, and use that. We can then get a reference to the window using the document's `parentWindow` property.

(2) Since our listener records will be stored in an object, not an array, we need a unique property name for each listener record. This unique listener ID will be the letter "l" followed by a counter that is incremented each time we register a new listener. This counter's initial value is declared after all the other event listener methods in this script:

core.js *(excerpt)*

```
Core._listenerCounter = 0;
```

3. The object that will contain records of all the listeners in this window will be stored in a property of the window, named `_allListeners`. If the property doesn't yet exist, we create it here as an empty object.

4. Lastly, we store our listener record in the `_allListeners` object. As we learned in Chapter 6, when we have a property name stored in a variable, we need to use array-like syntax to access it, even though we're dealing with an object.

Since all our listeners are stored in a single object, it will be really easy to clean them up when the document is unloaded. However, it might take a long time for our `_findListener` method to determine if a particular listener had been assigned to a particular event on a particular element—the method will need to search through this potentially massive object. To make this search more efficient, we'll store the unique ID of the listener in an array attached to `target`:

core.js *(excerpt)*

```
    // store this listener's ID in target
    if (!target._listeners) target._listeners = [];
    target._listeners[target._listeners.length] = listenerId;
```

Lastly, we need to make sure that our script is notified when the document is unloaded, so that we can clean up all the listeners:

core.js *(excerpt)*

```
    // set up Core._removeAllListeners to clean up all listeners
    // on unload
    if (!targetWindow._unloadListenerAdded)
    {
      targetWindow._unloadListenerAdded = true;
      targetWindow.attachEvent(
          "onunload", Core._removeAllListeners);
    }
  };
```

And that's `addEventListener` taken care of! We'll get to the method that actually removes all the listeners, `_removeAllListeners`, in a moment.

Thanks to a lot of the bookkeeping we did in `addEventListener`, `removeEventListener` is relatively straightforward:

core.js *(excerpt)*

```
Core.removeEventListener = function(target, type, listener)
{
  // find out if the listener was actually added to target
  var listenerIndex =
      Core._findListener(target, type, listener); (1)
  if (listenerIndex == -1) return;

  // get a reference to the window object containing target
  var targetDocument = target.document || target; (2)
  var targetWindow = targetDocument.parentWindow;

  // obtain the record of the listener from the window object
  var listenerId = target._listeners[listenerIndex]; (3)
  var listenerRecord = targetWindow._allListeners[listenerId];

  // remove the listener, and remove its ID from target
  target.detachEvent("on" + type, listenerRecord.listener2); (4)
  target._listeners.splice(listenerIndex, 1); (5)

  // remove the record of the listener from the window object
  delete targetWindow._allListeners[listenerId]; (6)
};
```

(1) Our first task is to find out if the specified combination of `target`, `type`, and `listener` actually corresponds to a registered event listener. We can use the same `_findListener` method that we used at the top of `addEventListener` to do this. This method returns the index of the listener in `target`'s `_listeners` array, or `-1` if no such listener has been registered.

(2) Having confirmed that the specified listener was registered, we need to get at the "listener record" object that we stored in the global `_allListeners` object for that listener. As in `addEventListener`, we start by getting a reference to the window.

3. We get the unique ID of the listener out of `target`'s `_listeners` array (using the `listenerIndex` that we got from `_findListener`), then use it to access the listener record in `_allListeners`.

4. With the listener record in hand, we can use Internet Explorer's `detachEvent` method to unregister the listener. Remember that, since we enhanced `listener` by wrapping it in a new function, the listener that we need to pass to `detachEvent` is the `listener2` property of the `listenerRecord`, not `listener` itself.

5. With the listener unregistered, we now need to remove our own records of its existence. First, we remove the relevant item from `target`'s `_listeners` array. To do so, we use a method that's supported by all JavaScript arrays: `splice`. We pass to this method the index of the first element that we want to remove from the array, and the number of elements to remove (in this case, `1`).

6. Finally, we remove the property from the global `_allListeners` object that contains the listener record. To do so, we make use of the rarely seen JavaScript `delete` statement, which deletes properties from objects.

Did you get all that? Don't worry too much if you're not able to follow all the code that goes into `addEventListener` and `removeEventListener`—it's fairly advanced, and when it comes right down to it, you really shouldn't have to understand JavaScript of this level of complexity to do something as simple as adding and removing event listeners. That's why we decided to hide this code at the back of this book, and why we badmouth the Internet Explorer Events model whenever we have the chance.

By comparison, the Internet Explorer versions of `preventDefault` and `stopPropagation` are simple:

core.js *(excerpt)*

```
Core.preventDefault = function(event)
{
  event.returnValue = false;
};

Core.stopPropagation = function(event)
```

```
{
  event.cancelBubble = true;
};
```

In order to prevent the default action associated with an event in IE, we simply set the event object's `returnValue` property to `false`. Similarly, in order to stop the propagation of an event in IE, we set its `cancelBubble` property to `true`. And that's all these methods need to do.

That takes care of the Internet Explorer versions of our four methods, but we still need to look at the two helper methods that they rely on—`_findListener` and `_removeAllListeners`:

core.js *(excerpt)*

```
Core._findListener = function(target, type, listener)
{
  // get the array of listener IDs added to target
  var listeners = target._listeners;
  if (!listeners) return -1;

  // get a reference to the window object containing target
  var targetDocument = target.document || target;
  var targetWindow = targetDocument.parentWindow;

  // searching backward (to speed up onunload processing),
  // find the listener
  for (var i = listeners.length - 1; i >= 0; i--)
  {
    // get the listener's ID from target
    var listenerId = listeners[i];

    // get the record of the listener from the window object
    var listenerRecord = targetWindow._allListeners[listenerId];

    // compare type and listener with the retrieved record
    if (listenerRecord.type == type &&
        listenerRecord.listener == listener)
    {
      return i;
    }
```

```
    }
    return -1;
  };
```

_findListener looks through the specified target's _listeners array, retrieving for of the each listener IDs it contains the corresponding listener record from the global _allListeners object. If it finds a listener record that matches the target, type, and listener specified, it returns the index of that listener's ID in target's _listeners array. If it doesn't find a matching record, it returns -1.

core.js *(excerpt)*

```
  Core._removeAllListeners = function()
  {
    var targetWindow = this;

    for (id in targetWindow._allListeners)
    {
      var listenerRecord = targetWindow._allListeners[id];
      listenerRecord.target.detachEvent(
          "on" + listenerRecord.type, listenerRecord.listener2);
      delete targetWindow._allListeners[id];
    }
  };
```

_removeAllListeners does something we haven't actually had to do anywhere else in this book—step through the properties of an object one at a time. Specifically, it steps through the listener records stored as properties in the _allListeners object. To step through the properties of an object, we use a for-in loop of this form:

```
for (propertyName in object)
{
  ⋮ access each property as object[propertyName]
}
```

Aside from that extra spoonful of syntactic sugar, _removeAllListeners is self-explanatory. Within the for-in loop, this method retrieves each of the listener record objects, uses Internet Explorer's detachEvent method to remove the corresponding listener, and then deletes the property from _allListeners.

Script Bootstrapping

From almost the very first script we saw in this book, we've used the Core library's `start` method to run a script's `init` method as soon as the document has finished loading.

`Core.start(`*`runnable`*`)`

This method runs the `init` method of the given script object (*runnable*) as soon as the document has finished loading.

As we've seen many times in this book, here's how we use `Core.start` to initialize an object once the page has loaded:

```
var MyScript =
{
  init: function()
  {
    ⋮
  },
  ⋮
};

Core.start(MyScript);
```

As we saw in Chapter 4, this method simply calls the `addEventListener` method to register the script's `init` method as a `load` event listener for the window:

core.js *(excerpt)*

```
Core.start = function(runnable)
{
  Core.addEventListener(window, "load", runnable.init);
};
```

This is a nice, simple approach to starting up a script (a process known as **bootstrapping**), but it has one important drawback: the `load` event that it depends on is not triggered until the page *and all linked resources* (such as images) have loaded. The practical consequence of this is that the page will sit there in its static form, with no added JavaScript functionality, while all the images load up. Finally, once all

the static content is in play, your JavaScript will be triggered and the dynamic functionality will snap into existence. If the user started to read the page while the images were loaded, this sudden (and potentially dramatic) change can be disorienting—and annoying.

Ideally, JavaScript code should start running the moment the element(s) of the document that it requires have been loaded, and are available for scripting. Exactly how element availability can be detected, and just what JavaScript is allowed to do at the various stages of the page loading process, is another swirling morass of vague standards and cross-browser incompatibility.

In the absence of clear standards for early bootstrapping, we have elected to stick with the standard, reliable `load` event throughout this book.

That said, there *is* an approach that works reliably in Firefox and Opera (version 9 or later) browsers: the `DOMContentLoaded` event. This is a nonstandard event that these browsers generate as soon as the HTML content of the document has finished loading (that is, before the images and other external resources). If this event were supported by all browsers, we could re-implement the Core library's `start` method as follows:

```
Core.start = function(runnable)
{
  Core.addEventListener(document, "DOMContentLoaded",
      runnable.init);
};
```

Normally, we'd shy away from a nonstandard solution like this, but the benefit to the user is so significant that it's worth taking some time to think about how this solution could be made to work cross-browser.

It would be nice to detect if the browser supported the `DOMContentLoaded` event, and have the `start` method use it if it's available, but fall back on the `load` event if it's not. Unfortunately, there is no reliable and future-proof way to detect events like this.

Instead, what we can do is register the listener for *both* events, and do a little extra work to make sure that browsers that *do* support `DOMContentLoaded` don't end up calling the script's `init` method twice:

```
Core.start = function(runnable)
{
  var initOnce = function()
  {
    if (arguments.callee.done) return;
    arguments.callee.done = true;
    runnable.init();
  };

  Core.addEventListener(document, "DOMContentLoaded", initOnce);
  Core.addEventListener(window, "load", initOnce);
};
```

This version of the method starts by creating a new function called `initOnce`. The first time this function is called, it will call the `init` method of the given script. If it's called again, however, it will do nothing. How does this work? The function sets a property named `done` on itself (a function's code can access the function itself as `arguments.callee`) the first time it's called, and checks for that property to stop execution and immediately return to the code that called the method on subsequent calls.

We can now register this `initOnce` function as a listener for both the document's `DOMContentLoaded` event, and the window's `load` event. In Firefox and Opera, `initOnce` will be called for both events, but will only run the script's `init` method on the first event. Other browsers will only call `initOnce` once—when the `load` event occurs.

Despite its nonstandard status, this approach is reliable enough to be recommended for production use. In the spirit of progressive enhancement, it starts with an approach that works just fine in standards-compliant browsers (the `load` event), then adds to it an enhancement (the `DOMContentLoaded` event) that will improve the user experience where it is supported. If you'd like your scripts to start up sooner in Firefox and Opera browsers, feel free to replace the Core library's `start` method with the final version above.

Can anything be done for other browsers—like, say, Internet Explorer? Well, some high-flying members of the JavaScript community have met with some success by toying with IE's support for the `defer` attribute of the `<script>` tag. Others have tried things like repeatedly checking for the presence of a required DOM element

(an approach known as **polling**) and launching the script as soon as it appears. A number of the major JavaScript libraries, such as jQuery, MooTools, and the Yahoo! UI Library, have even adopted different combinations of these solutions.

The short answer, however, is that—especially since the release of Internet Explorer 7, which apparently introduces some new bugs related to these solutions—there is no single solution that is 100% reliable. For this reason, we recommend sticking with the simple approach given above.

If you're interested in the full story on the JavaScript community's attempts to solve the early bootstrapping problem, the most complete and up-to-date article on the subject at the time of this writing is Peter Michaux's *The window.onload problem (still).*[2]

CSS Class Management Methods

The next four methods in the Core library have to do with manipulating the CSS classes that are applied to elements in your HTML documents:

`Core.addClass(target, theClass)`
: This method adds to the `className` property of *target* the specified CSS class (*theClass*), without removing any classes that may already have been applied to *target*.

`Core.getElementsByClass(theClass)`
: This method returns an array of all elements in the document that have the specified CSS class (*theClass*) applied to them.

`Core.hasClass(target, theClass)`
: This method returns `true` if *target* has the specified CSS class (*theClass*) applied to it, and `false` if not.

`Core.removeClass(target, theClass)`
: This method removes from the `className` property of *target* the specified CSS class (*theClass*), without removing any of the other classes that may also have been applied to *target*.

[2] http://peter.michaux.ca/article/553

These methods are fully described in Chapter 3. For completeness, you can find the code for these methods reproduced along with the rest of the library in the section called "The Complete Library" below.

Retrieving Computed Styles

The final method in our core library is `getComputedStyle`:

`Core.getComputedStyle(`*`element`*`, `*`styleProperty`*`)`

This method retrieves the effective value of the CSS property indicated by *styleProperty* once all the various sources of CSS styles (linked style sheets, embedded styles, inline styles, and dynamically-applied styles) have all been applied to *element*.

This method is fully described in Chapter 5; however, the complete code for this method is also reproduced along with the rest of the library in the following section.

The Complete Library

Here it is—the complete code for the Core JavaScript library.

core.js

```
var Core = {};

// W3C DOM 2 Events model
if (document.addEventListener)
{
  Core.addEventListener = function(target, type, listener)
  {
    target.addEventListener(type, listener, false);
  };

  Core.removeEventListener = function(target, type, listener)
  {
    target.removeEventListener(type, listener, false);
  };

  Core.preventDefault = function(event)
  {
    event.preventDefault();
```

```
  };

  Core.stopPropagation = function(event)
  {
    event.stopPropagation();
  };
}
// Internet Explorer Events model
else if (document.attachEvent)
{
  Core.addEventListener = function(target, type, listener)
  {
    // prevent adding the same listener twice, since DOM 2
    // Events ignores duplicates like this
    if (Core._findListener(target, type, listener) != -1)
      return;

    // listener2 calls listener as a method of target in one of
    // two ways, depending on what this version of IE supports,
    // and passes it the global event object as an argument
    var listener2 = function()
    {
      var event = window.event;

      if (Function.prototype.call)
      {
        listener.call(target, event);
      }
      else
      {
        target._currentListener = listener;
        target._currentListener(event)
        target._currentListener = null;
      }
    };

    // add listener2 using IE's attachEvent method
    target.attachEvent("on" + type, listener2);

    // create an object describing this listener so we can
    // clean it up later
    var listenerRecord =
    {
      target: target,
```

```
    type: type,
    listener: listener,
    listener2: listener2
  };

  // get a reference to the window object containing target
  var targetDocument = target.document || target;
  var targetWindow = targetDocument.parentWindow;

  // create a unique ID for this listener
  var listenerId = "l" + Core._listenerCounter++;

  // store a record of this listener in the window object
  if (!targetWindow._allListeners)
    targetWindow._allListeners = {};
  targetWindow._allListeners[listenerId] = listenerRecord;

  // store this listener's ID in target
  if (!target._listeners) target._listeners = [];
  target._listeners[target._listeners.length] = listenerId;

  // set up Core._removeAllListeners to clean up all
  // listeners on unload
  if (!targetWindow._unloadListenerAdded)
  {
    targetWindow._unloadListenerAdded = true;
    targetWindow.attachEvent(
        "onunload", Core._removeAllListeners);
  }
};

Core.removeEventListener = function(target, type, listener)
{
  // find out if the listener was actually added to target
  var listenerIndex = Core._findListener(
      target, type, listener);
  if (listenerIndex == -1) return;

  // get a reference to the window object containing target
  var targetDocument = target.document || target;
  var targetWindow = targetDocument.parentWindow;

  // obtain the record of the listener from the window object
  var listenerId = target._listeners[listenerIndex];
```

```
    var listenerRecord =
        targetWindow._allListeners[listenerId];

    // remove the listener, and remove its ID from target
    target.detachEvent("on" + type, listenerRecord.listener2);
    target._listeners.splice(listenerIndex, 1);

    // remove the record of the listener from the window object
    delete targetWindow._allListeners[listenerId];
};

Core.preventDefault = function(event)
{
    event.returnValue = false;
};

Core.stopPropagation = function(event)
{
    event.cancelBubble = true;
};

Core._findListener = function(target, type, listener)
{
    // get the array of listener IDs added to target
    var listeners = target._listeners;
    if (!listeners) return -1;

    // get a reference to the window object containing target
    var targetDocument = target.document || target;
    var targetWindow = targetDocument.parentWindow;

    // searching backward (to speed up onunload processing),
    // find the listener
    for (var i = listeners.length - 1; i >= 0; i--)
    {
        // get the listener's ID from target
        var listenerId = listeners[i];

        // get the record of the listener from the window object
        var listenerRecord =
            targetWindow._allListeners[listenerId];

        // compare type and listener with the retrieved record
        if (listenerRecord.type == type &&
```

```
          listenerRecord.listener == listener)
      {
        return i;
      }
    }
    return -1;
  };

  Core._removeAllListeners = function()
  {
    var targetWindow = this;

    for (id in targetWindow._allListeners)
    {
      var listenerRecord = targetWindow._allListeners[id];
      listenerRecord.target.detachEvent(
          "on" + listenerRecord.type, listenerRecord.listener2);
      delete targetWindow._allListeners[id];
    }
  };

  Core._listenerCounter = 0;
}

Core.addClass = function(target, theClass)
{
  if (!Core.hasClass(target, theClass))
  {
    if (target.className == "")
    {
      target.className = theClass;
    }
    else
    {
      target.className += " " + theClass;
    }
  }
};

Core.getElementsByClass = function(theClass)
{
  var elementArray = [];

  if (document.all)
```

```
  {
    elementArray = document.all;
  }
  else
  {
    elementArray = document.getElementsByTagName("*");
  }

  var matchedArray = [];
  var pattern = new RegExp("(^| )" + theClass + "( |$)");

  for (var i = 0; i < elementArray.length; i++)
  {
    if (pattern.test(elementArray[i].className))
    {
      matchedArray[matchedArray.length] = elementArray[i];
    }
  }

  return matchedArray;
};

Core.hasClass = function(target, theClass)
{
  var pattern = new RegExp("(^| )" + theClass + "( |$)");

  if (pattern.test(target.className))
  {
    return true;
  }

  return false;
};

Core.removeClass = function(target, theClass)
{
  var pattern = new RegExp("(^| )" + theClass + "( |$)");

  target.className = target.className.replace(pattern, "$1");
  target.className = target.className.replace(/ $/, "");
};

Core.getComputedStyle = function(element, styleProperty)
{
```

```
  var computedStyle = null;

  if (typeof element.currentStyle != "undefined")
  {
    computedStyle = element.currentStyle;
  }
  else
  {
    computedStyle =
        document.defaultView.getComputedStyle(element, null);
  }

  return computedStyle[styleProperty];
};

Core.start = function(runnable)
{
  Core.addEventListener(window, "load", runnable.init);
};
```

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

S

T

U

V

W

X

Y

Z