sitepoint

SIMPLY JAVASCRIPT

BY KEVIN YANK
& CAMERON ADAMS

UNLEASH THE AWESOME POWER OF JAVASCRIPT

Simply JavaScript (Chapters 1, 2, and 3)

Thank you for downloading these three chapters of *Simply JavaScript* by Kevin Yank and Cameron Adams.

This excerpt encapsulates the Summary of Contents, Information about the Author and SitePoint, Table of Contents, Preface, three chapters of the book, and the Index.

We hope you find this information useful in evaluating the book.

Summary of Contents of this Excerpt

Summary of Additional Book Contents

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
Printing History:
First Edition: June 2007

Editor: Georgina Laidlaw
Index Editor: Max McMaster
Cover Design: Alex Walker

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!

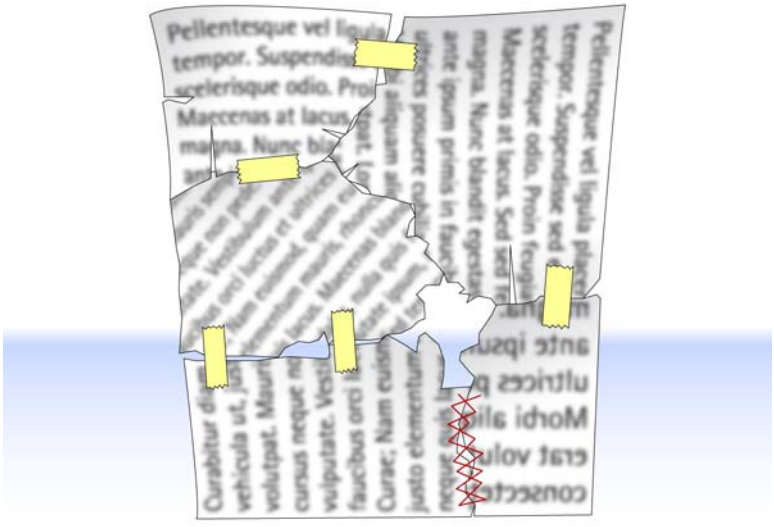
[1] Throughout this book, we'll refer to HTML and XHTML as just HTML. Which *you* choose is up to you, and doesn't have a much to do with JavaScript. In case it matters to you, the HTML code we'll present in this book will be valid XHTML 1.0 Strict.

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**.

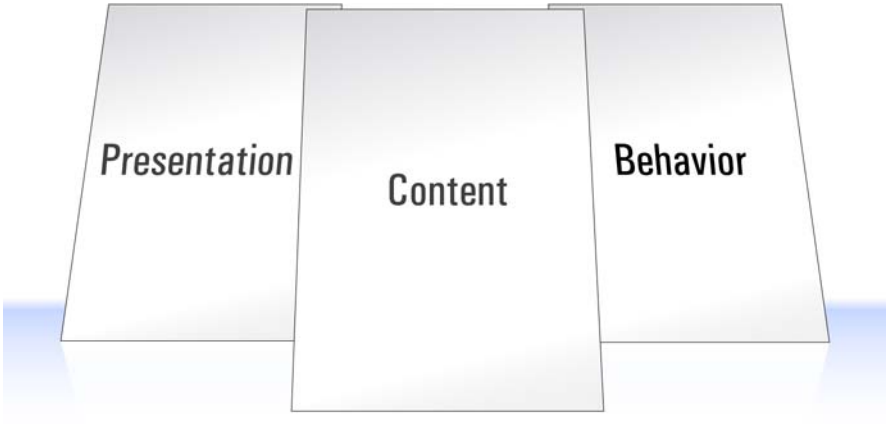


Figure 1.1. Separation of concerns

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.

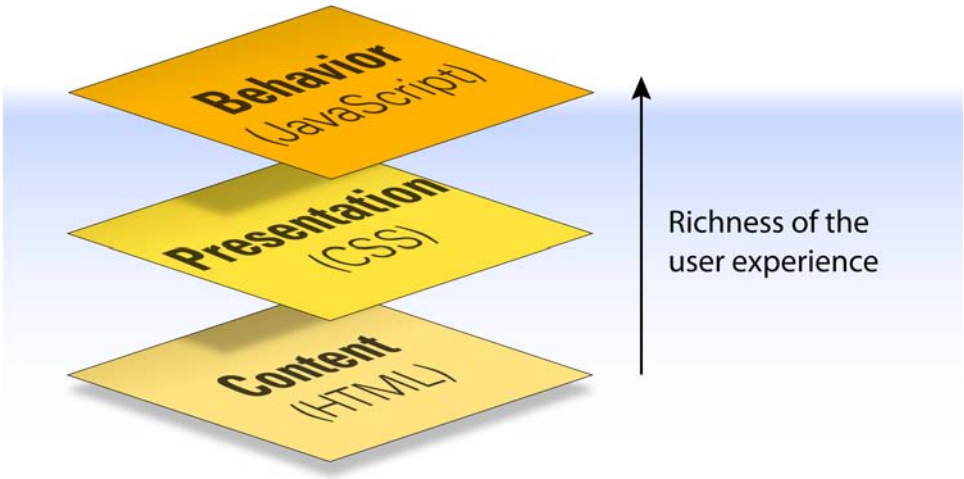


Figure 1.2. The three layers of the Web

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.

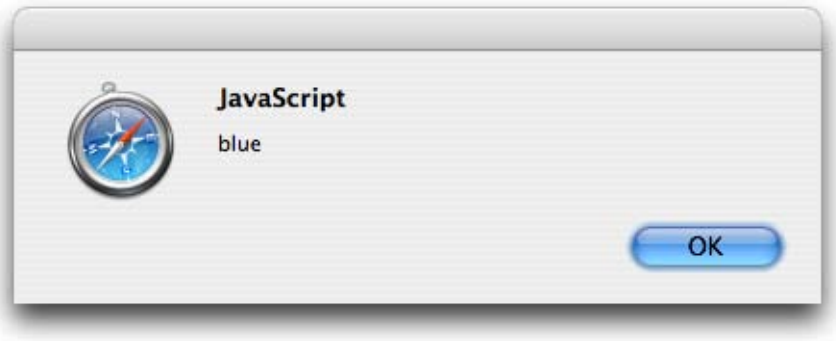


Figure 2.1. JavaScript replacing the variable name with its value

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.

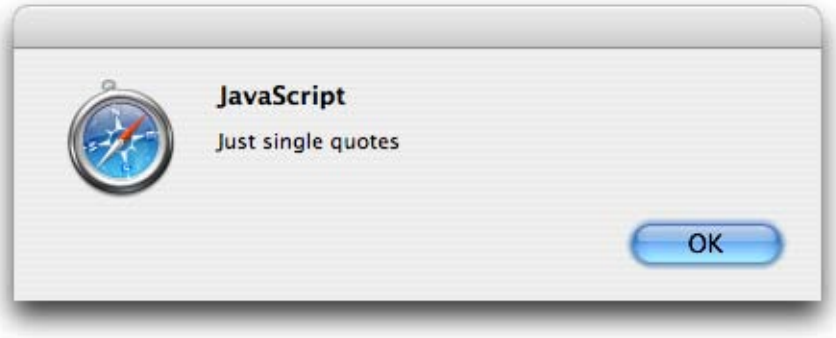


Figure 2.2. The string's value displaying without the quotes used to create the string

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.

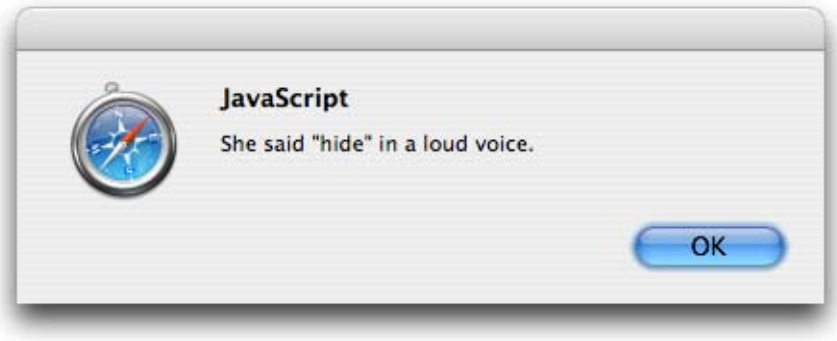


Figure 2.3. The string's value displaying without the backslashes used to escape quote marks in the string

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.

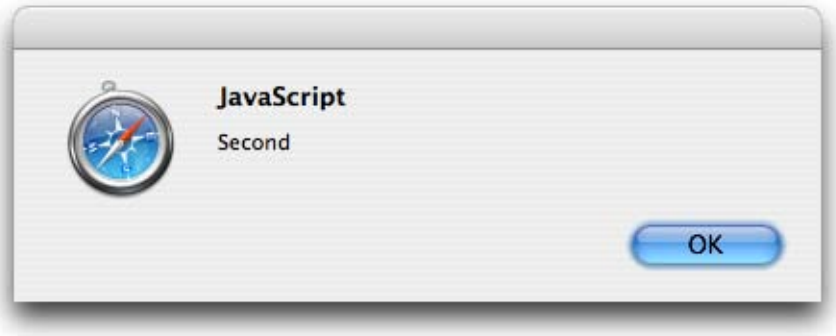


Figure 2.5. An alert box displaying a value retrieved from an array

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.

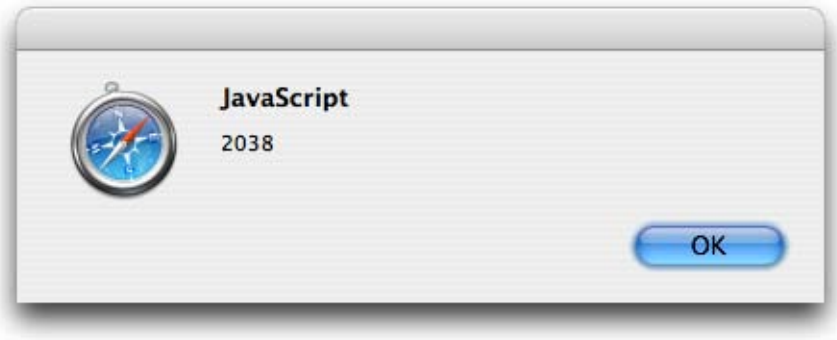


Figure 2.7. Finding a postcode using an associative array

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.

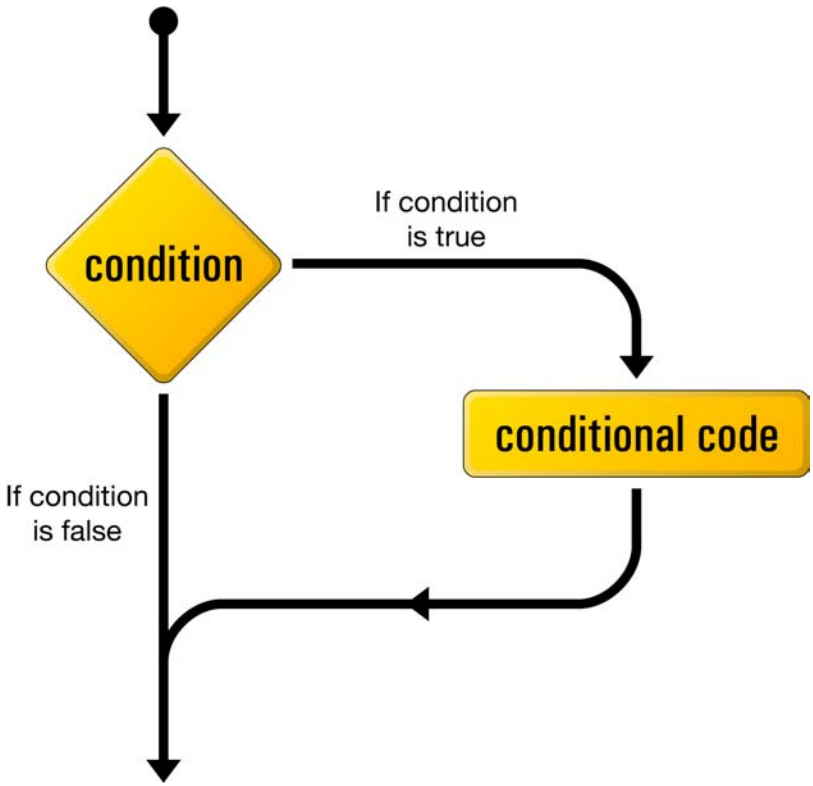


Figure 2.8. The logical flow of an `if` statement

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.");
}
```

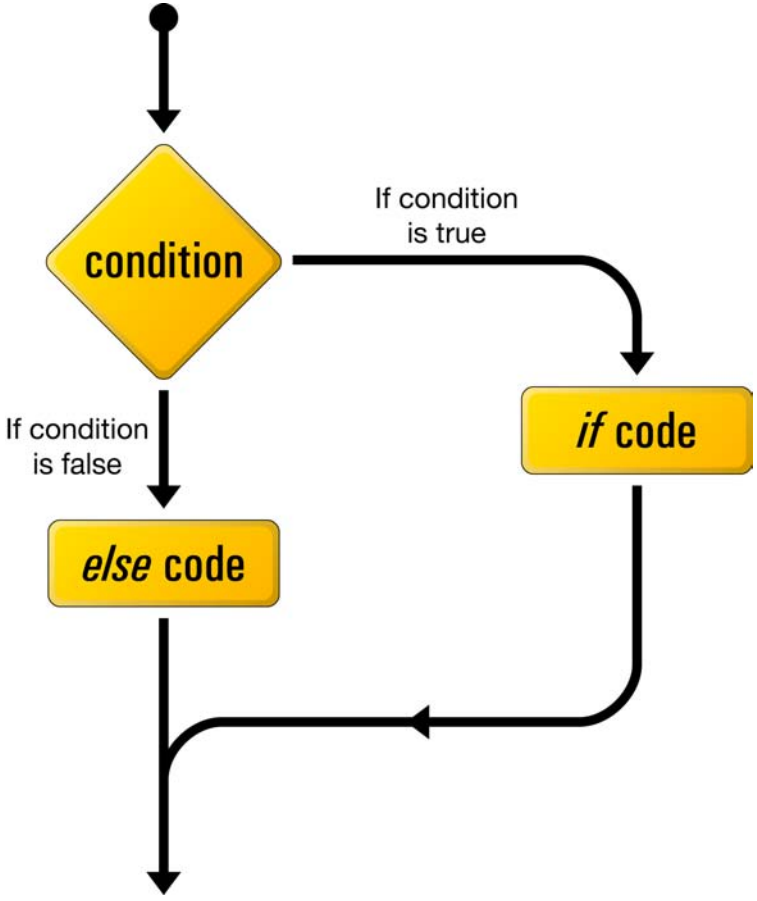


Figure 2.9. The logical flow of an `if-else` statement

```
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!

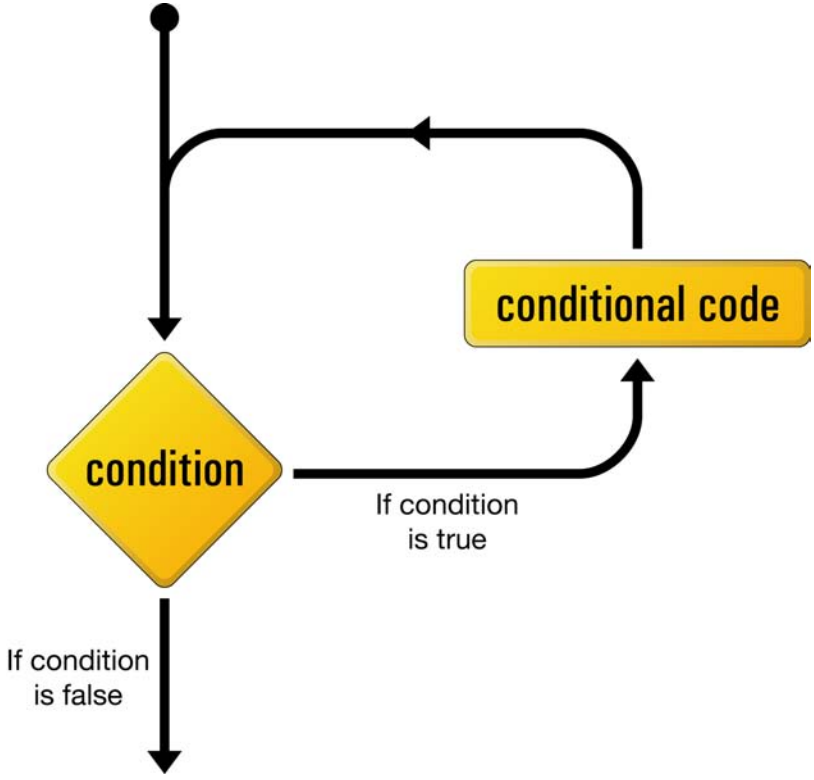


Figure 2.10. The logical flow of a `while` loop

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.

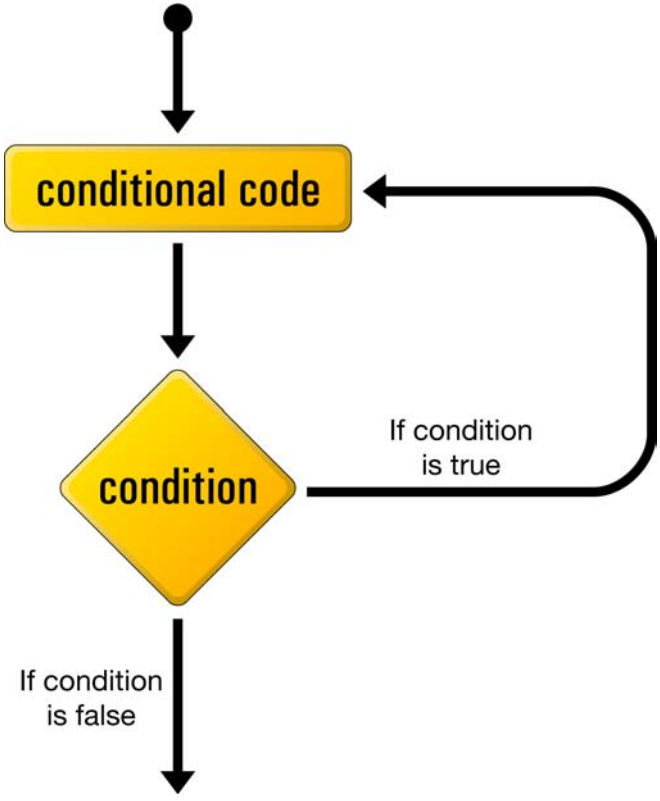


Figure 2.11. The logical flow of a `do-while` loop

```
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:

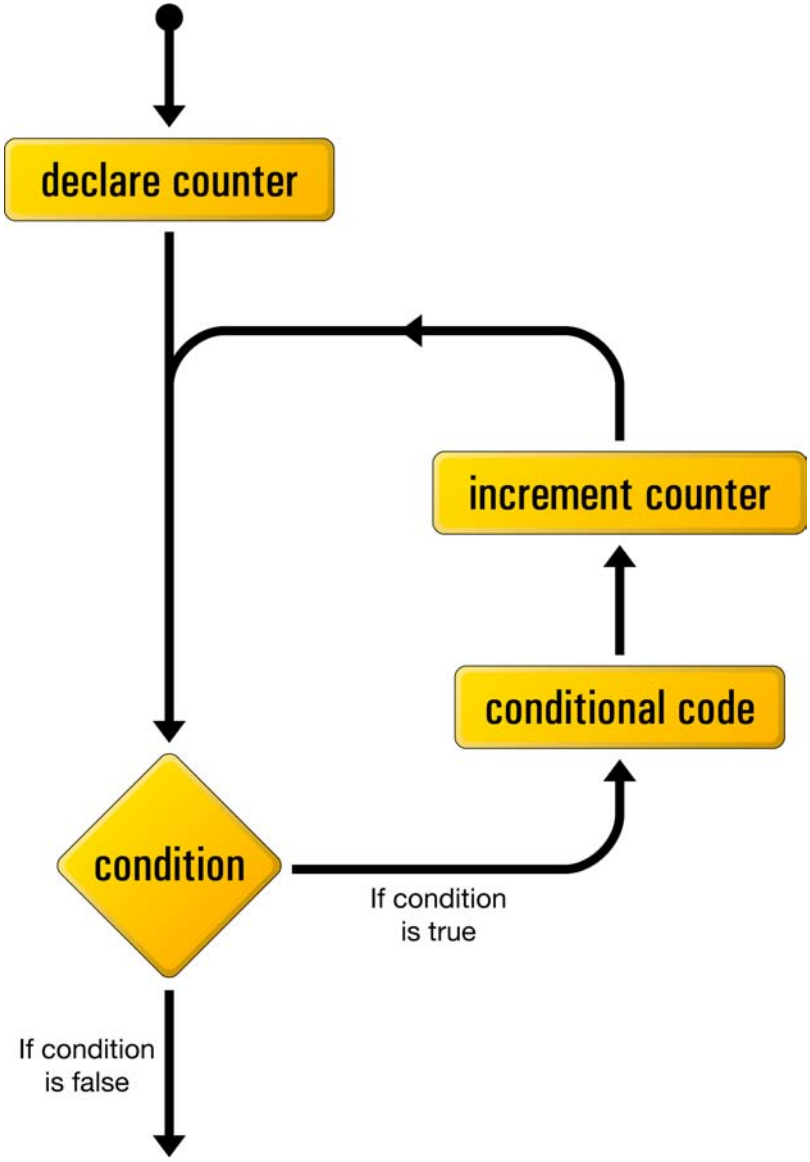


Figure 2.12. The logical flow of a `for` loop

```
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.

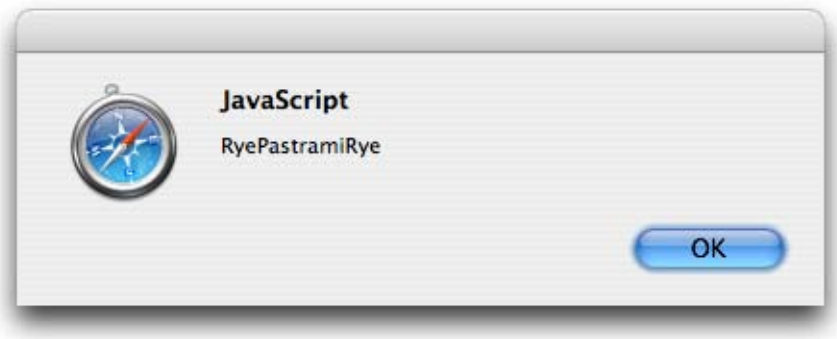


Figure 2.13. Using a function argument as a variable

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.

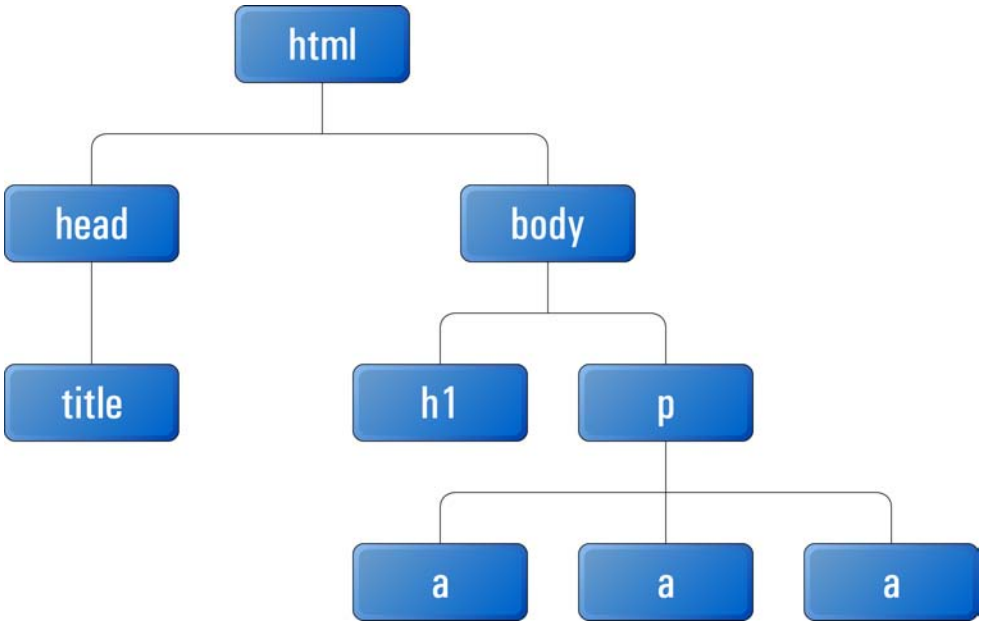


Figure 3.1. Each element on an HTML page linking to its parent in the DOM

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.

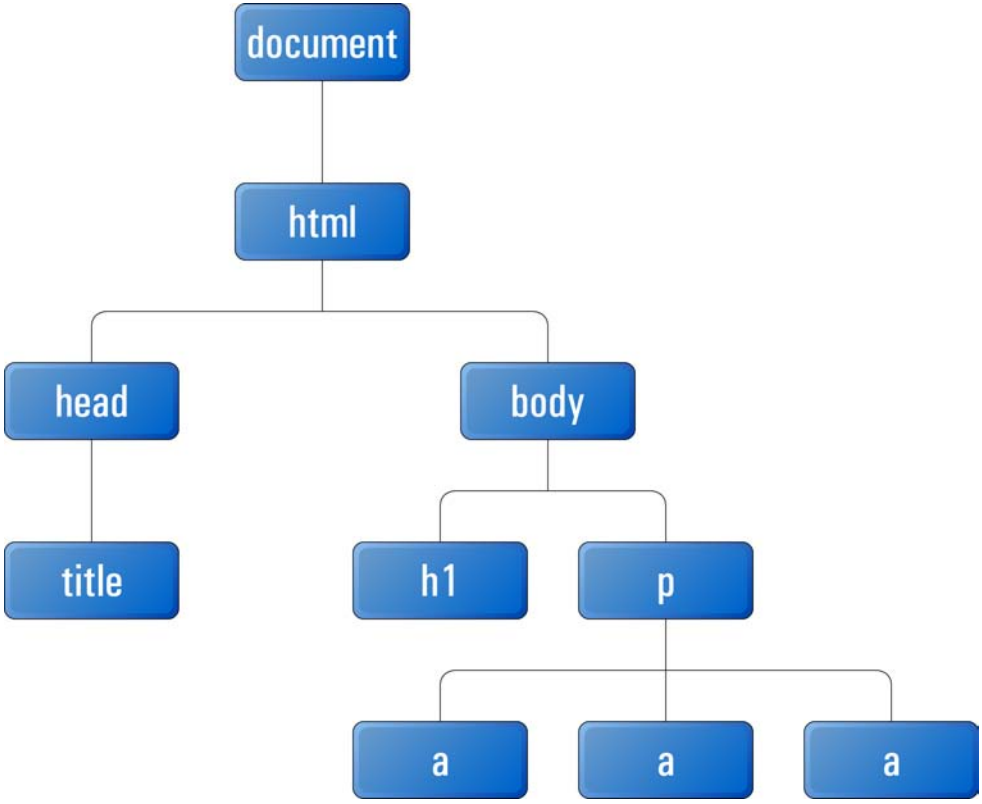


Figure 3.2. The DOM tree, including the document node

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.

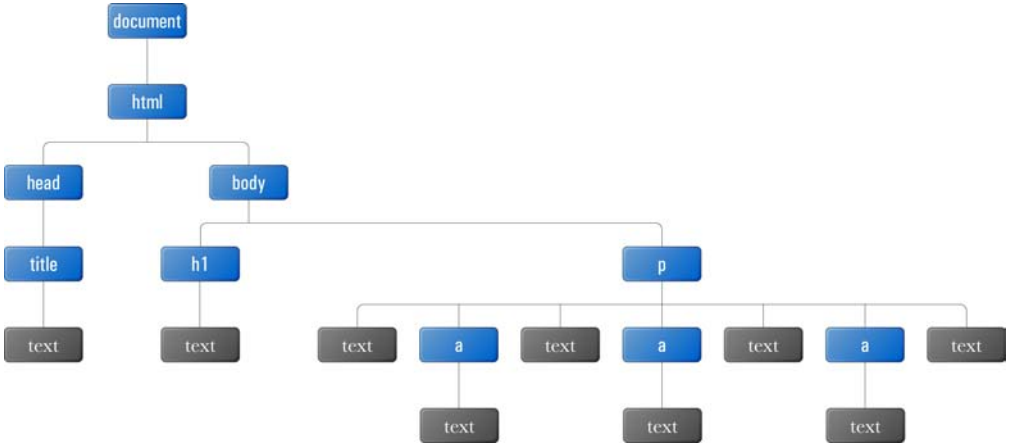


Figure 3.3. The complete DOM tree, including text nodes

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.

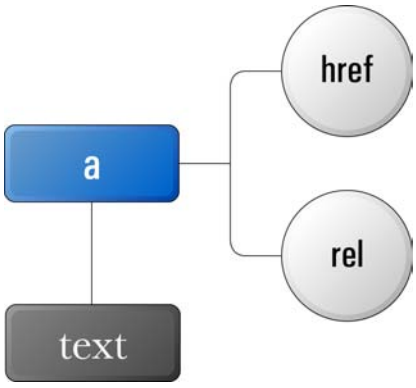


Figure 3.4. The `href` and `rel` attributes represented as attribute nodes in the DOM

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.

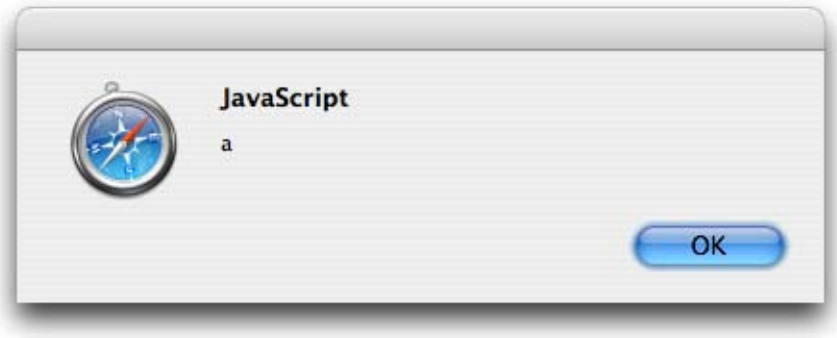


Figure 3.5. Displaying an element's tag name using the nodeName property

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.

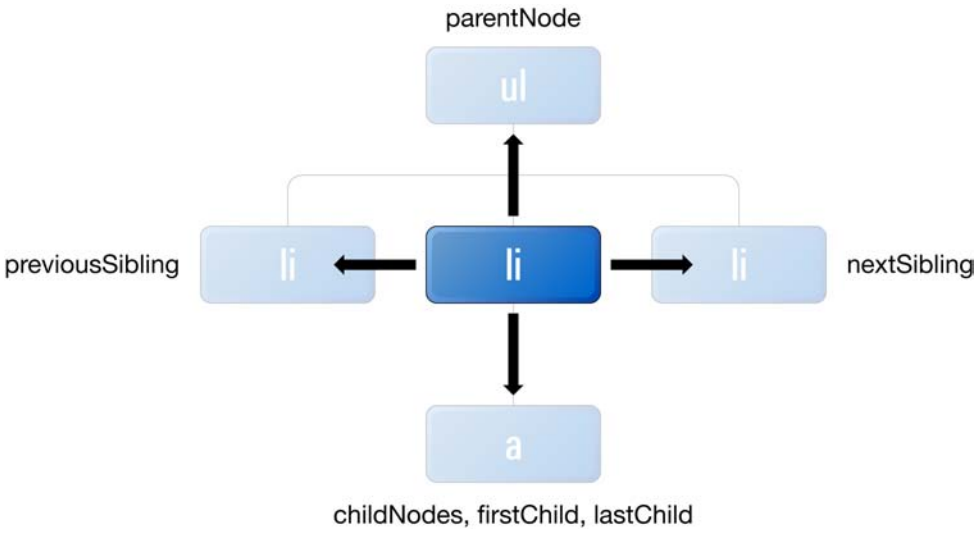


Figure 3.6. Moving around the DOM tree using the element node's DOM properties

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.

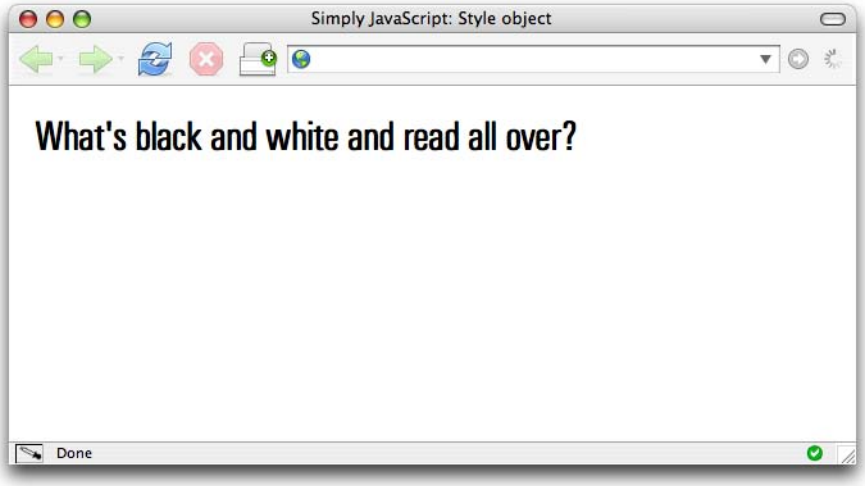


Figure 3.7. A standard page

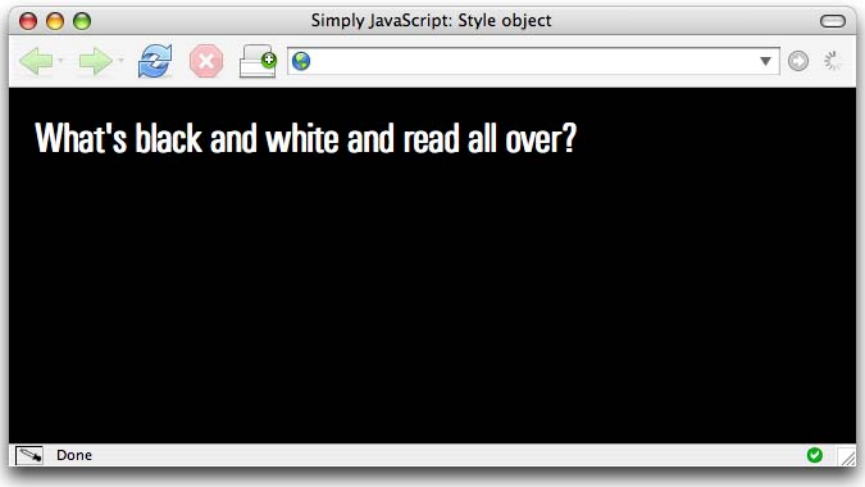


Figure 3.8. The same page, altered using `style`

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.

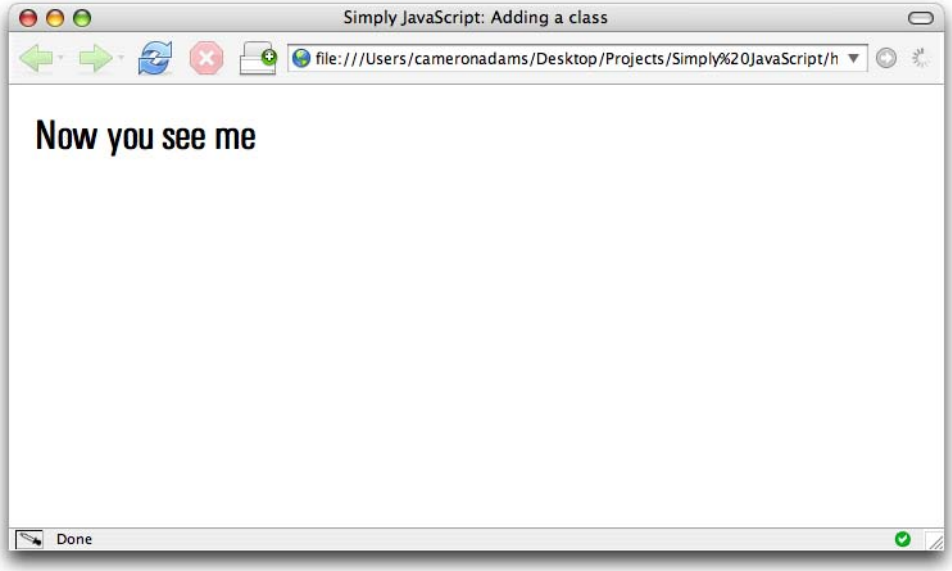


Figure 3.9. The page before we start work

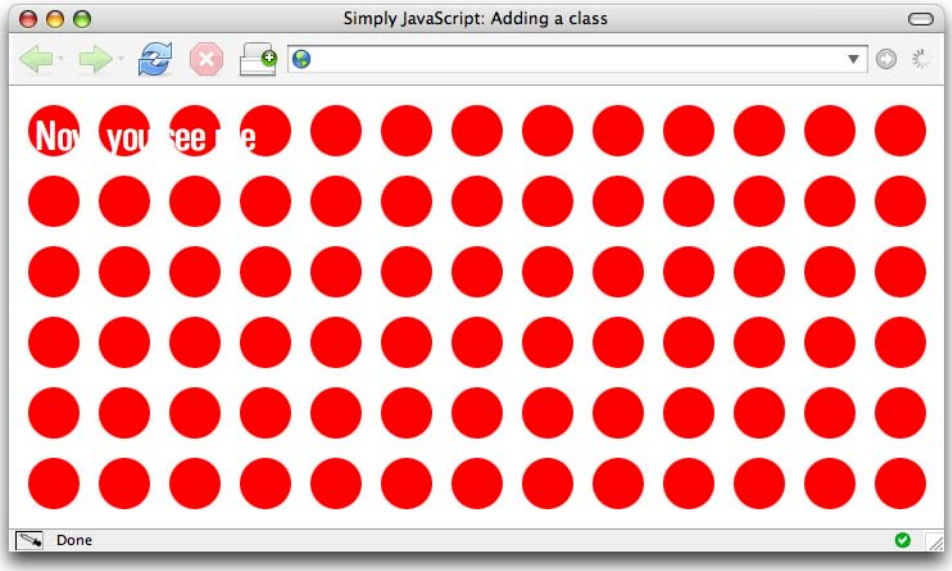


Figure 3.10. The display after we change the class of the body element

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.

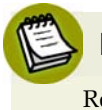
Finding All Tables with Class `dataTable`

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`:

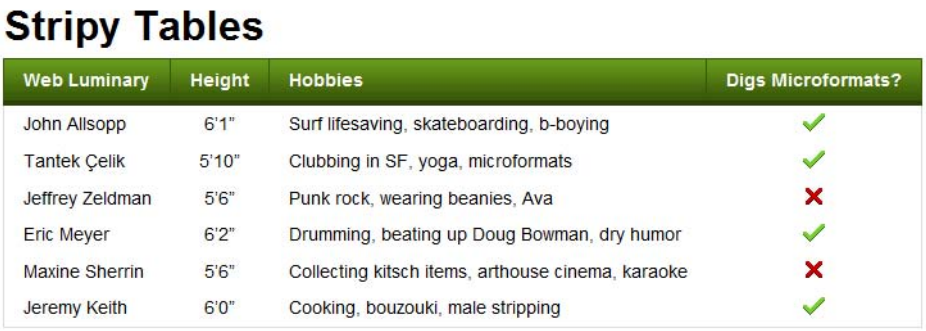
stripy_tables.js *(excerpt)*

```
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)*

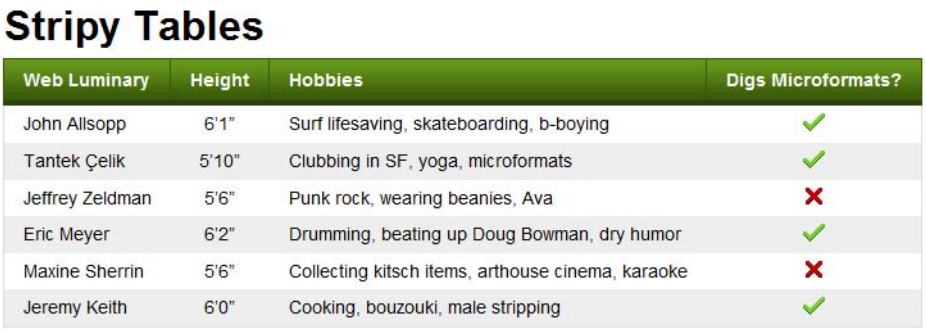
```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en-US">
  <head>
    <title>Stripy Tables</title>
    <meta http-equiv="Content-Type"
```



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!

What's Next?

If you've enjoyed these chapters from *Simply JavaScript* and are ready to **unleash the awesome power of JavaScript**, why not order yourself a copy?

Packed with full-color examples, *Simply JavaScript* is all you need to start programming in JavaScript the right way. Learn how easy it is to use JavaScript to solve real-world problems, build smarter forms, track user events (such as mouse clicks and key strokes), and design eye-catching animations. Then move on to more powerful techniques using the DOM and Ajax.

In the rest of the book, you'll learn how to

- Use JavaScript to respond to the actions of your users.
- Create animations that bring your web site to life.
- Quickly and effectively debug and correct errors.
- Build forms that validate entries and interact with your users.
- Build a richer user experience with Ajax.
- Then explore just how far you can take the awesome power of JavaScript.

The book's full-color layout makes it one of the most usable and enjoyable JavaScript books available.

All JavaScript code used to create each of the components is available for download, and is guaranteed to be simple, efficient, best practice, and ready to use in your own web site.

Kevin and Cameron's unique use of the JavaScript library developed especially for the book makes *Simply JavaScript* the most readable, easy-to-understand beginners' book available.

You won't find a better way to learn JavaScript from scratch.

Buy the full version now!

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