

Head First JavaScript Programming

A Brain-Friendly Guide



Watch out for
common JavaScript
traps and pitfalls

**A learner's guide to
JavaScript programming**

Launch your
programming
career in
one chapter



Avoid
embarrassing
typing conversion
mistakes



Bend your mind
around 120 puzzles
& exercises



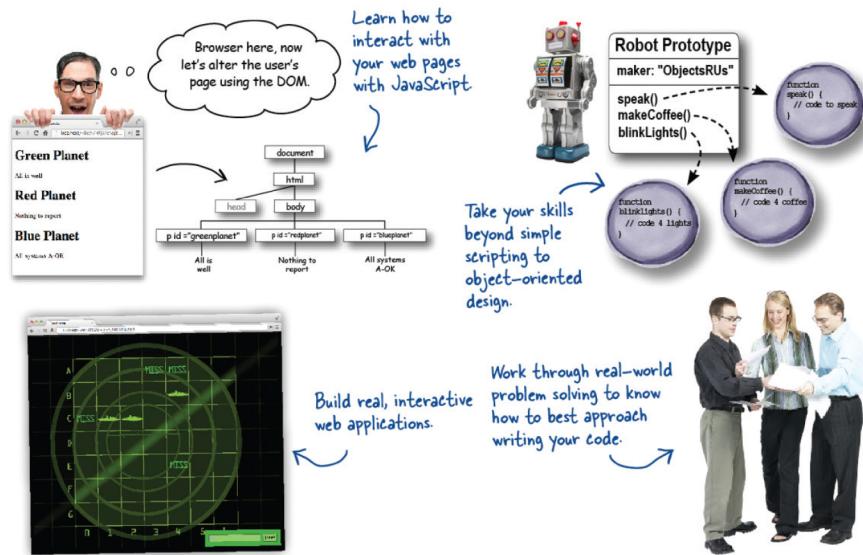
Learn why everything
your friends know about
functions & objects are
probably wrong

Eric Freeman & Elisabeth Robson

JavaScript Programming

What will you learn from this book?

This brain-friendly guide teaches you everything from JavaScript language fundamentals to advanced topics, including objects, functions, and the browser's document object model. You won't just be reading—you'll be playing games, solving puzzles, pondering mysteries, and interacting with JavaScript in ways you never imagined. And you'll write real code, lots of it, so you can start building your own web applications.



What's so special about this book?

Using the latest research in neurobiology, cognitive science, and learning theory, *Head First JavaScript Programming* employs a visually rich format designed for the way your brain works, not a text-heavy approach that puts you to sleep.

"An excellent introduction to programming combined with advanced topics like object construction, inheritance, and closures allows readers to move from the basics to some of the most interesting concepts in modern computer programming."

—Peter Casey
Professor, Central Oregon
Community College

"This book takes you behind the scenes of JavaScript and leaves you with a deep understanding of how this remarkable programming language works."

—Chris Fuselier
Engineering Consultant

"I wish I'd had *Head First JavaScript Programming* when I was starting out!"

—Daniel Konopacki
Staff Software Engineer,
The Walt Disney Company

JavaScript / Programming

US \$49.99

CAN \$52.99

ISBN: 978-1-449-34013-1



twitter.com/headfirstlabs
facebook.com/HeadFirst

oreilly.com
headfirstlabs.com

Praise for *Head First JavaScript Programming*

“Warning: Do not read *Head First JavaScript Programming* unless you want to learn the fundamentals of programming with JavaScript in an entertaining and meaningful fashion. There may be an additional side effect that you may actually recall more about JavaScript than after reading typical technical books.”

— **Jesse Palmer, Senior Software Developer, Gannett Digital**

“If every elementary and middle school student studied Elisabeth and Eric’s *Head First HTML and CSS*, and if *Head First JavaScript Programming* and *Head First HTML5 Programming* were part of the high school math and science curriculum, then our country would never lose its competitive edge.”

— **Michael Murphy, senior systems consultant, The History Tree**

“The *Head First* series utilizes elements of modern learning theory, including constructivism, to bring readers up to speed quickly. The authors have proven with this book that expert-level content can be taught quickly and efficiently. Make no mistake here, this is a serious JavaScript book, and yet, fun reading!”

— **Frank Moore, Web designer and developer**

“Looking for a book that will keep you interested (and laughing) but teach you some serious programming skills? *Head First JavaScript Programming* is it!”

— **Tim Williams, software entrepreneur**

“Add this book to your library regardless of your programming skill level!”

— **Chris Fuselier, engineering consultant**

“Robson and Freeman have done it again! Using the same fun and information-packed style as their previous books in the *Head First* series, *Head First JavaScript Programming* leads you through entertaining and useful projects that, chapter-by-chapter, allow programmers—even nonspecialists like myself—to develop a solid foundation in modern JavaScript programming that we can use to solve real problems.”

— **Russell Alleen-Willems, digital archeologist, DiachronicDesign.com**

“Freeman and Robson continue to use innovative teaching methods for communicating complex concepts to basic principles.”

— **Mark Arana, Strategy & Innovation, The Walt Disney Studios**

Praise for other books by Eric T. Freeman and Elisabeth Robson

“Just the right tone for the geeked-out, casual-cool guru coder in all of us. The right reference for practical development strategies—gets my brain going without having to slog through a bunch of tired, stale professor-speak.”

— **Travis Kalanick, CEO Uber**

“This book’s admirable clarity, humor and substantial doses of clever make it the sort of book that helps even non-programmers think well about problem-solving.”

— **Cory Doctorow, co-editor of Boing Boing, Science Fiction author**

“I feel like a thousand pounds of books have just been lifted off of my head.”

— **Ward Cunningham, inventor of the Wiki**

“One of the very few software books I’ve ever read that strikes me as indispensable. (I’d put maybe 10 books in this category, at the outside.)”

— **David Gelernter, Professor of Computer Science, Yale University**

“I laughed, I cried, it moved me.”

— **Daniel Steinberg, Editor-in-Chief, java.net**

“I can think of no better tour guides than Eric and Elisabeth.”

— **Miko Matsumura, VP of Marketing and Developer Relations at Hazelcast
Former Chief Java Evangelist, Sun Microsystems**

“I literally love this book. In fact, I kissed this book in front of my wife.”

— **Satish Kumar**

“The highly graphic and incremental approach precisely mimics the best way to learn this stuff...”

— **Danny Goodman, author of *Dynamic HTML: The Definitive Guide***

“Eric and Elisabeth clearly know their stuff. As the Internet becomes more complex, inspired construction of web pages becomes increasingly critical. Elegant design is at the core of every chapter here, each concept conveyed with equal doses of pragmatism and wit.”

— **Ken Goldstein, former CEO of Shop.com and author of
*This is Rage: A Novel of Silicon Valley and Other Madness***

Other O'Reilly books by Eric T. Freeman and Elisabeth Robson

Head First Design Patterns
Head First HTML and CSS
Head First HTML5 Programming

Other related books from O'Reilly

Head First HTML5 Programming
JavaScript: The Definitive Guide
JavaScript Enlightenment

Other books in O'Reilly's *Head First* series

Head First HTML and CSS
Head First HTML5 Programming
Head First Design Patterns
Head First Servlets and JSP
Head First SQL
Head First Software Development
Head First C#
Head First Java
Head First Object-Oriented Analysis and Design (OOA&D)
Head First Ajax
Head First Rails
Head First PHP & MySQL
Head First Web Design
Head First Networking
Head First iPhone and iPad Development
Head First jQuery

Head First JavaScript Programming



Wouldn't it be dreamy if there was
a JavaScript book that was more
fun than going to the dentist and
more revealing than an IRS form?
It's probably just a fantasy...

Eric T. Freeman
Elisabeth Robson

O'REILLY®

Beijing • Cambridge • Köln • Sebastopol • Tokyo

Head First JavaScript Programming

by Eric T. Freeman and Elisabeth Robson

Copyright © 2014 Eric Freeman, Elisabeth Robson. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly Media books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://my.safaribooksonline.com>). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Editors: Meghan Blanchette, Courtney Nash

Cover Designer: Randy Comer

Code Monkeys: Eric T. Freeman, Elisabeth Robson

Production Editor: Melanie Yarbrough

Indexer: Potomac Indexing

Proofreader: Rachel Monaghan

Page Viewer: Oliver



Printing History:

March 2014: First Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. The *Head First* series designations, *Head First JavaScript Programming*, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and the authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

In other words, if you use anything in *Head First JavaScript Programming* to, say, run a nuclear power plant, you're on your own. We do, however, encourage you to visit Webville.

No variables were harmed in the making of this book.

ISBN: 978-1-449-34013-1

[M]

To JavaScript—you weren't born with a silver spoon in your mouth, but you've outclassed every language that's challenged you in the browser.

Authors of Head First JavaScript Programming



← Eric Freeman

Eric is described by Head First series co-creator Kathy Sierra as “one of those rare individuals fluent in the language, practice, and culture of multiple domains from hipster hacker, corporate VP, engineer, think tank.”

Professionally, Eric recently ended nearly a decade as a media company executive—having held the position of CTO of Disney Online & Disney.com at The Walt Disney Company. Eric is now devoting his time to WickedlySmart, a startup he co-created with Elisabeth.

By training, Eric is a computer scientist, having studied with industry luminary David Gelernter during his Ph.D. work at Yale University. His dissertation is credited as the seminal work in alternatives to the desktop metaphor, and also as the first implementation of activity streams, a concept he and Dr. Gelernter developed.

In his spare time, Eric is deeply involved with music; you’ll find Eric’s latest project, a collaboration with ambient music pioneer Steve Roach, available on the iPhone app store under the name Immersion Station.

Eric lives with his wife and young daughter on Bainbridge Island. His daughter is a frequent visitor to Eric’s studio, where she loves to turn the knobs of his synths and audio effects.

Write to Eric at eric@wickedlysmart.com or visit his site at <http://ericfreeman.com>.



↙ Elisabeth Robson

Elisabeth is a software engineer, writer, and trainer. She has been passionate about technology since her days as a student at Yale University, where she earned a Masters of Science in Computer Science and designed a concurrent, visual programming language and software architecture.

Elisabeth’s been involved with the Internet since the early days; she co-created the award-winning Web site, The Ada Project, one of the first Web sites designed to help women in computer science find career and mentorship information online.

She’s currently co-founder of WickedlySmart, an online education experience centered on web technologies, where she creates books, articles, videos and more. Previously, as Director of Special Projects at O’Reilly Media, Elisabeth produced in-person workshops and online courses on a variety of technical topics and developed her passion for creating learning experiences to help people understand technology. Prior to her work with O’Reilly, Elisabeth spent time spreading fairy dust at The Walt Disney Company, where she led research and development efforts in digital media.

When not in front of her computer, you’ll find Elisabeth hiking, cycling or kayaking in the great outdoors, with her camera nearby, or cooking vegetarian meals.

You can send her email at beth@wickedlysmart.com or visit her blog at <http://elisabethrobson.com>.

Table of Contents (summary)

| | |
|--|-----|
| Intro | xxv |
| 1 A quick dip into JavaScript: <i>Getting your feet wet</i> | 1 |
| 2 Writing real code: <i>Going further</i> | 43 |
| 3 Introducing functions: <i>Getting functional</i> | 79 |
| 4 Putting some order in your data: <i>Arrays</i> | 125 |
| 5 Understanding objects: <i>A trip to Objectville</i> | 173 |
| 6 Interacting with your web page: <i>Getting to know the DOM</i> | 229 |
| 7 Types, equality, conversion, and all that jazz: <i>Serious types</i> | 265 |
| 8 Bringing it all together: <i>Building an app</i> | 317 |
| 9 Asynchronous coding: <i>Handling events</i> | 381 |
| 10 First-class functions: <i>Liberated functions</i> | 429 |
| 11 Anonymous functions, scope, and closures: <i>Serious functions</i> | 475 |
| 12 Advanced object construction: <i>Creating objects</i> | 521 |
| 13 Using prototypes: <i>Extra-strength objects</i> | 563 |
| Appendix: The Top Ten Topics (we didn't cover): <i>Leftovers</i> | 623 |

Table of Contents (the real thing)

Intro

Your brain on JavaScript. Here you are trying to *learn* something, while here your *brain* is doing you a favor by making sure the learning doesn't *stick*. Your brain's thinking, "Better leave room for more important things, like which wild animals to avoid and whether naked snowboarding is a bad idea." So how *do* you trick your brain into thinking that your life depends on knowing JavaScript programming?



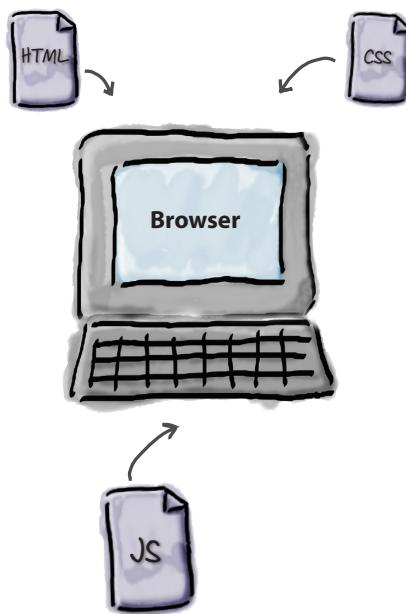
| | |
|---|--------|
| Who is this book for? | xxvi |
| We know what you're thinking. | xxvii |
| We think of a "Head First" reader as a learner. | xxviii |
| Metacognition: thinking about thinking | xxix |
| Here's what WE did: | xxx |
| Here's what YOU can do to bend your brain into submission | xxxii |
| Read Me | xxxii |
| Tech Reviewers | xxxv |
| Acknowledgments* | xxxvi |

a quick dip into javascript

1

Getting your feet wet

JavaScript gives you superpowers. The **true programming language** of the web, JavaScript lets you **add behavior** to your web pages. No more dry, boring, static pages that just sit there looking at you—with JavaScript you're going to be able to reach out and touch your users, react to interesting events, grab data from the web to use in your pages, draw graphics right in your web pages and a lot more. And once you know JavaScript you'll also be in a position to create **totally new** behaviors for your users.



| | |
|---|----|
| The way JavaScript works | 2 |
| How you're going to write JavaScript | 3 |
| How to get JavaScript into your page | 4 |
| JavaScript, you've come a long way baby... | 6 |
| How to make a statement | 10 |
| Variables and values | 11 |
| Back away from that keyboard! | 12 |
| Express yourself | 15 |
| Doing things more than once | 17 |
| How the while loop works | 18 |
| Making decisions with JavaScript | 22 |
| And, when you need to make LOTS of decisions | 23 |
| Reach out and communicate with your user | 25 |
| A closer look at console.log | 27 |
| Opening the console | 28 |
| Coding a Serious JavaScript Application | 29 |
| How do I add code to my page? (let me count the ways) | 32 |
| We're going to have to separate you two | 33 |



writing real code



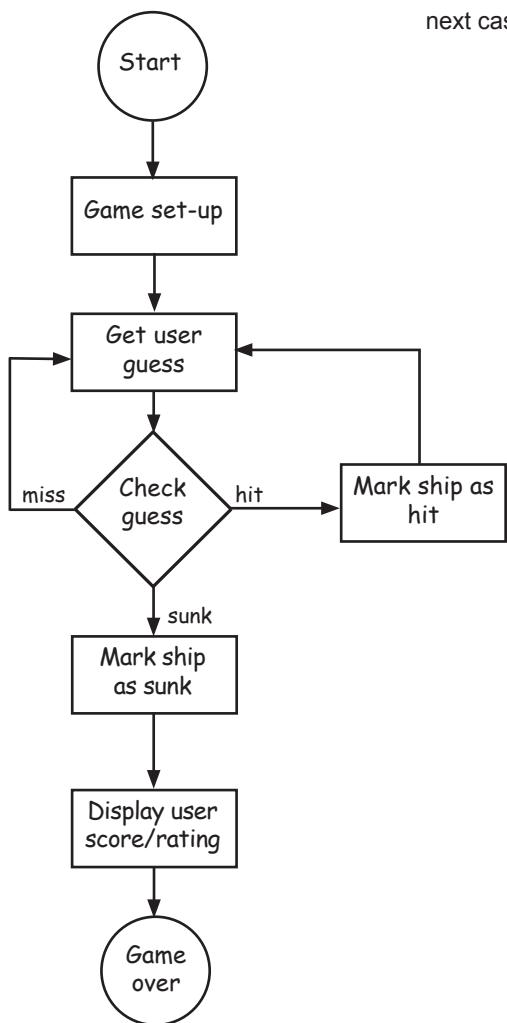
2

Going further

You already know about variables, types, expressions...

we could go on. The point is, you already know a few things about

JavaScript. In fact, you know enough to write some **real code**. Some code that does something interesting, some code that someone would want to use. What you're lacking is the **real experience** of writing code, and we're going to remedy that right here and now. How? By jumping in head first and coding up a casual game, all written in JavaScript. Our goal is ambitious but we're going to take it one step at a time. Come on, let's get this started, and if you want to launch the next casual startup, we won't stand in your way; the code is yours.



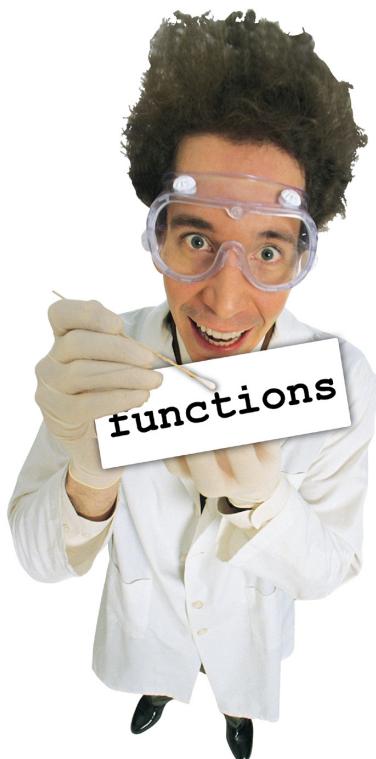
| | |
|--|----|
| Let's build a Battleship game | 44 |
| Our first attempt... | 44 |
| First, a high-level design | 45 |
| Working through the Pseudocode | 47 |
| Oh, before we go any further, don't forget the HTML! | 49 |
| Writing the Simple Battleship code | 50 |
| Now let's write the game logic | 51 |
| Step One: setting up the loop, getting some input | 52 |
| How prompt works | 53 |
| Checking the user's guess | 54 |
| So, do we have a hit? | 56 |
| Adding the hit detection code | 57 |
| Provide some post-game analysis | 58 |
| And that completes the logic! | 60 |
| Doing a little Quality Assurance | 61 |
| Can we talk about your verbosity... | 65 |
| Finishing the Simple Battleship game | 66 |
| How to assign random locations | 67 |
| The world-famous recipe for generating a random number | 67 |
| Back to do a little more QA | 69 |
| Congrats on your first true JavaScript program, and a short word about reusing code | 71 |

introducing functions

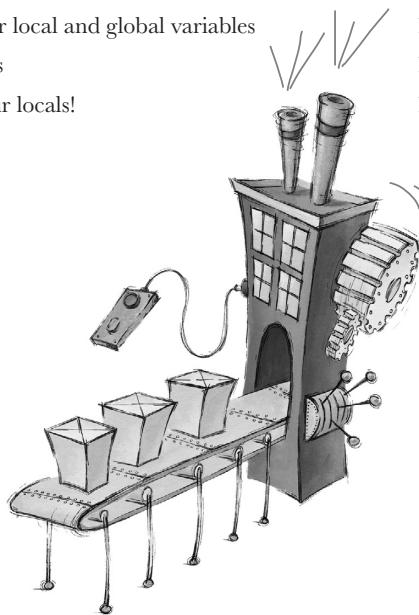
3

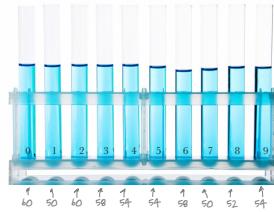
Getting functional

Get ready for your first superpower. You've got some programming under your belt; now it's time to really move things along with **functions**. Functions give you the power to write code that can be applied to all sorts of different circumstances, code that can be **reused** over and over, code that is much more **manageable**, code that can be **abstracted** away and given a simple name so you can forget all the complexity and get on with the important stuff. You're going to find not only that functions are your gateway from scripter to programmer, they're the key to the JavaScript programming style. In this chapter we're going to start with the basics: the mechanics, the ins and outs of how functions really work, and then you'll keep honing your function skills throughout the rest of the book. So, let's get a good foundation started, *now*.



| | |
|--|-----|
| What's wrong with the code anyway? | 81 |
| By the way, did we happen to mention FUNCTIONS? | 83 |
| Okay, but how does it actually work? | 84 |
| What can you pass to a function? | 89 |
| JavaScript is pass-by-value. | 92 |
| Weird Functions | 94 |
| Functions can return things too | 95 |
| Tracing through a function with a return statement | 96 |
| Global and local variables | 99 |
| Knowing the scope of your local and global variables | 101 |
| The short lives of variables | 102 |
| Don't forget to declare your locals! | 103 |





putting some order in your data

4 Arrays

There's more to JavaScript than numbers, strings and booleans. So far you've been writing JavaScript code with **primitives**—simple strings, numbers and booleans, like “Fido”, 23, and true. And you can do a lot with primitive types, but at some point you've got to deal with **more data**. Say, all the items in a shopping cart, or all the songs in a playlist, or a set of stars and their apparent magnitude, or an entire product catalog. For that we need a little more *ummph*. The type of choice for this kind of ordered data is a JavaScript **array**, and in this chapter we're going to walk through how to put your data into an array, how to pass it around and how to operate on it. We'll be looking at a few other ways to **structure your data** in later chapters but let's get started with arrays.

| | |
|---|-----|
| Can you help Bubbles-R-U斯? | 126 |
| How to represent multiple values in JavaScript | 127 |
| How arrays work | 128 |
| How big is that array anyway? | 130 |
| The Phrase-O-Matic | 132 |
| Meanwhile, back at Bubbles-R-U斯... | 135 |
| How to iterate over an array | 138 |
| But wait, there's a better way to iterate over an array | 140 |
| Can we talk about your verbosity? | 146 |
| Redoing the for loop with the post-increment operator | 147 |
| Quick test drive | 147 |
| Creating an array from scratch (and adding to it) | 151 |
| And the winners are... | 155 |
| A quick survey of the code... | 157 |
| Writing the printAndGetHighScore function | 158 |
| Refactoring the code using printAndGetHighScore | 159 |
| Putting it all together... | 161 |



understanding objects

5

A trip to Objectville

So far you've been using primitives and arrays in your code. And, you've approached coding in quite a **procedural manner** using simple statements, conditionals and for/while loops with functions—that's not exactly **object-oriented**. In fact, it's not object-oriented *at all!* We did use a few objects here and there without really knowing it, but you haven't written any of your own objects yet. Well, the time has come to leave this boring procedural town behind to create some **objects** of your own. In this chapter, you're going to find out why using objects is going to make your life so much better—well, better in a **programming sense** (we can't really help you with your fashion sense *and* your JavaScript skills all in one book). Just a warning: once you've discovered objects you'll never want to come back. Send us a postcard when you get there.

| | |
|--|-----|
| Did someone say “Objects”?! | 174 |
| Thinking about properties... | 175 |
| How to create an object | 177 |
| What is Object-Oriented Anyway? | 180 |
| How properties work | 181 |
| How does a variable hold an object? Inquiring minds want to know... | 186 |
| Comparing primitives and objects | 187 |
| Doing even more with objects... | 188 |
| Stepping through pre-qualification | 190 |
| Let's talk a little more about passing objects to functions | 192 |
| Oh Behave! Or, how to add behavior to your objects | 198 |
| Improving the drive method | 199 |
| Why doesn't the drive method know about the started property? | 202 |
| How this works | 204 |
| How behavior affects state... Adding some Gas-o-line | 210 |
| Now let's affect the behavior with the state | 211 |
| Congrats on your first objects! | 213 |
| Guess what? There are objects all around you! (and they'll make your life easier) | 214 |



interacting with your web page

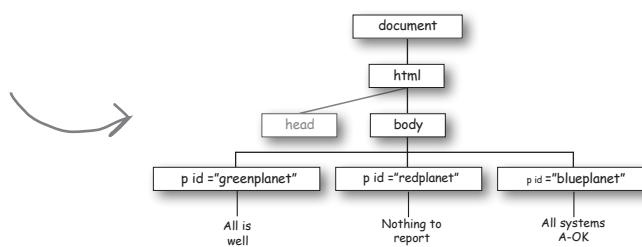
6

Getting to know the DOM

You've come a long way with JavaScript. In fact you've evolved from a newbie to a scripter to, well, a **programmer**. But, there's something missing. To really begin leveraging your JavaScript skills you need to know how to interact with the web page your code lives in. Only by doing that are you going to be able to write pages that are **dynamic**, pages that react, that respond, that update themselves after they've been loaded. So how do you interact with the page? By using the **DOM**, otherwise known as the **document object model**. In this chapter we're going to break down the DOM and see just how we can use it, along with JavaScript, to teach your page a few new tricks.



| | |
|---|-----|
| The “crack the code challenge.” | 230 |
| So what does the code do? | 231 |
| How JavaScript really interacts with your page | 233 |
| How to bake your very own DOM | 234 |
| A first taste of the DOM | 235 |
| Getting an element with getElementById | 240 |
| What, exactly, am I getting from the DOM? | 241 |
| Finding your inner HTML | 242 |
| What happens when you change the DOM | 244 |
| A test drive around the planets | 247 |
| Don’t even think about running my code until the page is fully loaded! | 249 |
| You say “event handler,” I say “callback” | 250 |
| How to set an attribute with setAttribute | 255 |
| More fun with attributes! (you can GET attributes too) | 256 |
| Don’t forget getElementById can return null too! | 256 |
| Any time you ask for something, you need to make sure you got back what you expected... | 256 |
| So what else is a DOM good for anyway? | 258 |

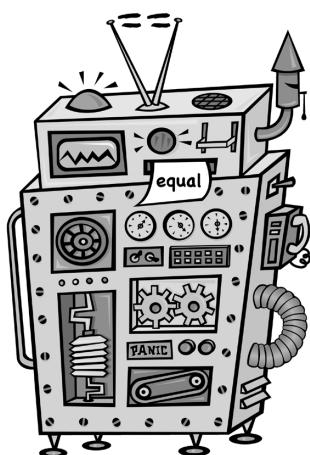


types, equality, conversion, and all that jazz

7

Serious types

It's time to get serious about our types. One of the great things about JavaScript is you can get a long way without knowing a lot of details of the language. But to truly **master the language**, get that promotion and get on to the things you really want to do in life, you have to rock at **types**. Remember what we said way back about JavaScript? That it didn't have the luxury of a silver-spoon, academic, peer-reviewed language definition? Well that's true, but the academic life didn't stop Steve Jobs and Bill Gates, and it didn't stop JavaScript either. It does mean that JavaScript doesn't have the... well, the most thought-out type system, and we'll find a few **idiosyncrasies** along the way. But, don't worry, in this chapter we're going to nail all that down, and soon you'll be able to avoid all those embarrassing moments with types.



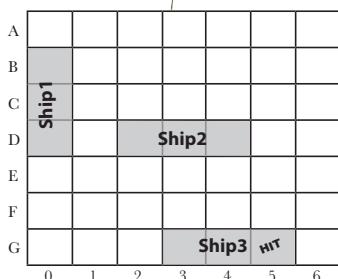
| | |
|---|-----|
| The truth is out there... | 266 |
| Watch out, you might bump into undefined when you aren't expecting it... | 268 |
| How to use null | 271 |
| Dealing with NaN | 273 |
| It gets even weirder | 273 |
| We have a confession to make | 275 |
| Understanding the equality operator (otherwise known as ==) | 276 |
| How equality converts its operands (sounds more dangerous than it actually is) | 277 |
| How to get strict with equality | 280 |
| Even more type conversions... | 286 |
| How to determine if two objects are equal | 289 |
| The truthy is out there... | 291 |
| What JavaScript considers falsey | 292 |
| The Secret Life of Strings | 294 |
| How a string can look like a primitive and an object | 295 |
| A five-minute tour of string methods (and properties) | 297 |
| Chair Wars | 301 |

bringing it all together

8

Building an app

Put on your toolbelt. That is, the toolbelt with all your new coding skills, your knowledge of the DOM, and even some HTML & CSS. We're going to bring everything together in this chapter to create our first true **web application**. No more **silly toy games** with one battleship and a single row of hiding places. In this chapter we're building the **entire experience**: a nice big game board, multiple ships and user input right in the web page. We're going to create the page structure for the game with HTML, visually style the game with CSS, and write JavaScript to code the game's behavior. Get ready: this is an all out, pedal to the metal development chapter where we're going to lay down some serious code.



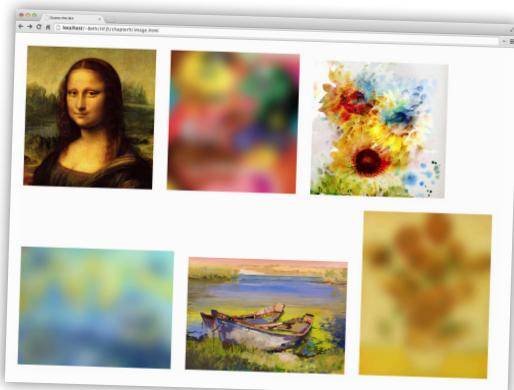
| | |
|---|-----|
| This time, let's build a REAL Battleship game | 318 |
| Stepping back... to HTML and CSS | 319 |
| Creating the HTML page: the Big Picture | 320 |
| Adding some more style | 324 |
| Using the hit and miss classes | 327 |
| How to design the game | 329 |
| Implementing the View | 331 |
| How displayMessage works | 331 |
| How displayHit and displayMiss work | 333 |
| The Model | 336 |
| How we're going to represent the ships | 338 |
| Implementing the model object | 341 |
| Setting up the fire method | 342 |
| Implementing the Controller | 349 |
| Processing the player's guess | 350 |
| Planning the code... | 351 |
| Implementing parseGuess | 352 |
| Counting guesses and firing the shot | 355 |
| How to add an event handler to the Fire! button | 359 |
| Passing the input to the controller | 360 |
| How to place ships | 364 |
| Writing the generateShip method | 365 |
| Generate the starting location for the new ship | 366 |
| Completing the generateShip method | 367 |

9

asynchronous coding

Handling events

After this chapter you're going to realize you aren't in Kansas anymore. Up until now, you've been writing code that typically executes from top to bottom—sure, your code might be a little more complex than that, and make use of a few functions, objects and methods, but at some point the code just runs its course. Now, we're awfully sorry to break this to you this late in the book, but that's **not how you typically write JavaScript code**. Rather, most JavaScript is written to **react to events**. What kind of events? Well, how about a user clicking on your page, data arriving from the network, timers expiring in the browser, changes happening in the DOM and that's just a few examples. In fact, all kinds of events are happening **all the time**, behind the scenes, in your browser. In this chapter we're going rethink our approach to JavaScript coding, and learn how and why we should write code that reacts to events.



| | |
|---|-----|
| What are events? | 383 |
| What's an event handler? | 384 |
| How to create your first event handler | 385 |
| Test drive your event | 386 |
| Getting your head around events... by creating a game | 388 |
| Implementing the game | 389 |
| Test drive | 390 |
| Let's add some more images | 394 |
| Now we need to assign the same event handler to each image's onclick property | 395 |
| How to reuse the same handler for all the images | 396 |
| How the event object works | 399 |
| Putting the event object to work | 401 |
| Test drive the event object and target | 402 |
| Events and queues | 404 |
| Even more events | 407 |
| How setTimeout works | 408 |
| Finishing the image game | 412 |
| Test driving the timer | 413 |

10

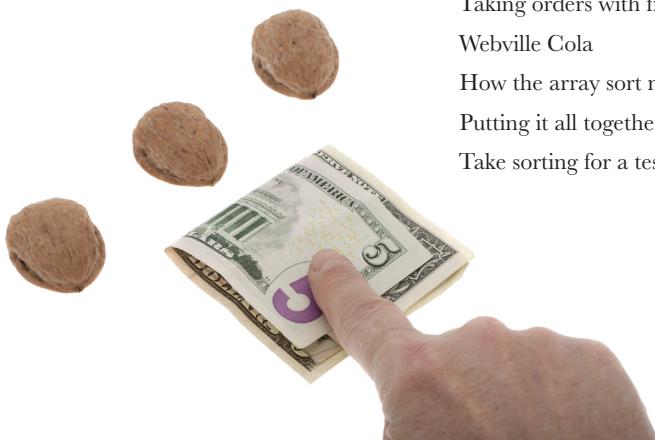
first class functions

Liberated functions

Know functions, then rock. Every art, craft, and discipline has a key principle that separates the intermediate players from the rock star virtuosos—when it comes to JavaScript, it's truly understanding **functions** that makes the difference. Functions are fundamental to JavaScript, and many of the techniques we use to **design and organize** code depend on advanced knowledge and use of functions. The path to learning functions at this level is an interesting and often mind-bending one, so get ready... This chapter is going to be a bit like Willy Wonka giving a tour of the chocolate factory—you're going to encounter some wild, wacky and wonderful things as you learn more about JavaScript functions.



| | |
|--|-----|
| The mysterious double life of the function keyword | 430 |
| Function declarations versus function expressions | 431 |
| Parsing the function declaration | 432 |
| What's next? The browser executes the code | 433 |
| Moving on... The conditional | 434 |
| How functions are values too | 439 |
| Did we mention functions have First Class status in JavaScript? | 442 |
| Flying First Class | 443 |
| Writing code to process and check passengers | 444 |
| Iterating through the passengers | 446 |
| Passing a function to a function | 447 |
| Returning functions from functions | 450 |
| Writing the flight attendant drink order code | 451 |
| The flight attendant drink order code: a different approach | 452 |
| Taking orders with first class functions | 454 |
| Webville Cola | 457 |
| How the array sort method works | 459 |
| Putting it all together | 460 |
| Take sorting for a test drive | 462 |



anonymous functions, scopes, and closures

11

Serious functions

You've put functions through their paces, but there's more to learn.

In this chapter we take it further; we get hard-core. We're going to show you how to **really handle** functions. This won't be a super long chapter, but it will be intense, and at the end you're going to be more expressive with your JavaScript than you thought possible. You're also going to be ready to take on a coworker's code, or jump into an open source JavaScript library, because we're going to cover some common coding idioms and conventions around functions. And if you've never heard of an **anonymous function** or a **closure**, boy are you in the right place.

| | |
|--|-----|
| Taking a look at the other side of functions... | 476 |
| How to use an anonymous function | 477 |
| We need to talk about your verbosity, again | 479 |
| When is a function defined? It depends... | 483 |
| What just happened? Why wasn't fly defined? | 484 |
| How to nest functions | 485 |
| How nesting affects scope | 486 |
| A little review of lexical scope | 488 |
| Where things get interesting with lexical scope | 489 |
| Functions Revisited | 491 |
| Calling a function (revisited) | 492 |
| What the heck is a closure? | 495 |
| Closing a function | 496 |
| Using closures to implement a magic counter | 498 |
| Looking behind the curtain... | 499 |
| Creating a closure by passing a function expression as an argument | 501 |
| The closure contains the actual environment, not a copy | 502 |
| Creating a closure with an event handler | 503 |
| How the Click me! closure works | 506 |



12

advanced object construction

Creating objects

So far we've been crafting objects by hand. For each object, we've used an **object literal** to specify each and every property. That's okay on a small scale, but for serious code we need something better. That's where **object constructors** come in. With constructors we can create objects much more easily, and we can create objects that all adhere to the same **design blueprint**—meaning we can use constructors to ensure each object has the same properties and includes the same methods. And with constructors we can write object code that is much more **concise** and a lot less error prone when we're creating lots of objects. So, let's get started and after this chapter you'll be talking constructors just like you grew up in Objectville.

| | |
|--|-----|
| Creating objects with object literals | 522 |
| Using conventions for objects | 523 |
| Introducing Object Constructors | 525 |
| How to create a Constructor | 526 |
| How to use a Constructor | 527 |
| How constructors work | 528 |
| You can put methods into constructors as well | 530 |
| It's Production Time! | 536 |
| Let's test drive some new cars | 538 |
| Don't count out object literals just yet | 539 |
| Rewiring the arguments as an object literal | 540 |
| Reworking the Car constructor | 541 |
| Understanding Object Instances | 543 |
| Even constructed objects can have their own independent properties | 546 |
| Real World Constructors | 548 |
| The Array object | 549 |
| Even more fun with built-in objects | 551 |

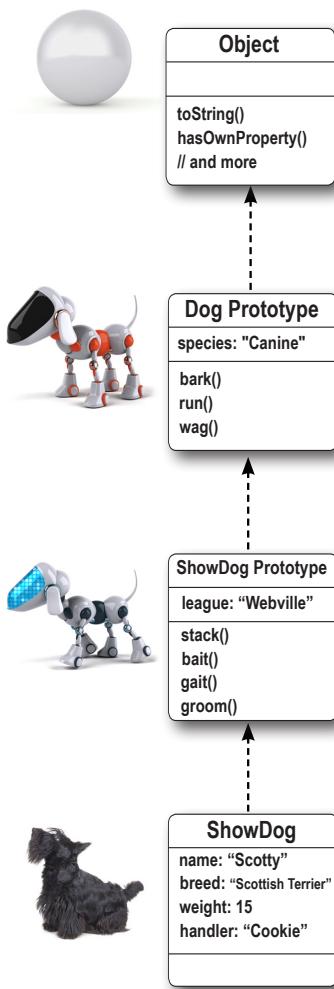


using prototypes

13

Extra strength objects

Learning how to create objects was just the beginning. It's time to put some muscle on our objects. We need more ways to create **relationships** between objects and to **share code** among them. And, we need ways to extend and enhance existing objects. In other words, we need more tools. In this chapter, you're going to see that JavaScript has a very powerful **object model**, but one that is a bit different than the status quo object-oriented language. Rather than the typical class-based object-oriented system, JavaScript instead opts for a more powerful **prototype** model, where objects can inherit and extend the behavior of other objects. What is that good for? You'll see soon enough. Let's get started...



| | |
|---|-----|
| Hey, before we get started, we've got a better way to diagram our objects | 565 |
| Revisiting object constructors: we're reusing code, but are we being efficient? | 566 |
| Is duplicating methods really a problem? | 568 |
| What are prototypes? | 569 |
| Inheriting from a prototype | 570 |
| How inheritance works | 571 |
| Overriding the prototype | 573 |
| How to set up the prototype | 576 |
| Prototypes are dynamic | 582 |
| A more interesting implementation of the sit method | 584 |
| One more time: how the sitting property works | 585 |
| How to approach the design of the show dogs | 589 |
| Setting up a chain of prototypes | 591 |
| How inheritance works in a prototype chain | 592 |
| Creating the show dog prototype | 594 |
| Creating a show dog Instance | 598 |
| A final cleanup of show dogs | 602 |
| Stepping through Dog.call | 604 |
| The chain doesn't end at dog | 607 |
| Using inheritance to your advantage...by overriding built-in behavior | 608 |
| Using inheritance to your advantage...by extending a built-in object | 610 |
| Grand Unified Theory of Everything | 612 |
| Better living through objects | 612 |
| Putting it all together | 613 |
| What's next? | 613 |

14

Appendix: Leftovers

The top ten topics (we didn't cover)

We've covered a lot of ground, and you're almost finished with this book.

We'll miss you, but before we let you go, we wouldn't feel right about sending you out into the world without a little more preparation. We can't possibly fit everything you'll need to know into this relatively small chapter.

Actually, we *did* originally include everything you need to know about JavaScript Programming (not already covered by the other chapters), by reducing the type point size to .00004. It all fit, but nobody could read it.

So we threw most of it away, and kept the best bits for this Top Ten appendix. This really *is* the end of the book.

Except for the index, of course (a must-read!).

| | |
|--|-----|
| #1 jQuery | 624 |
| #2 Doing more with the DOM | 626 |
| #3 The Window Object | 627 |
| #4 Arguments | 628 |
| #5 Handling exceptions | 629 |
| #6 Adding event handlers with addEventListener | 630 |
| #7 Regular Expressions | 632 |
| #8 Recursion | 634 |
| #9 JSON | 636 |
| #10 Server-side JavaScript | 637 |

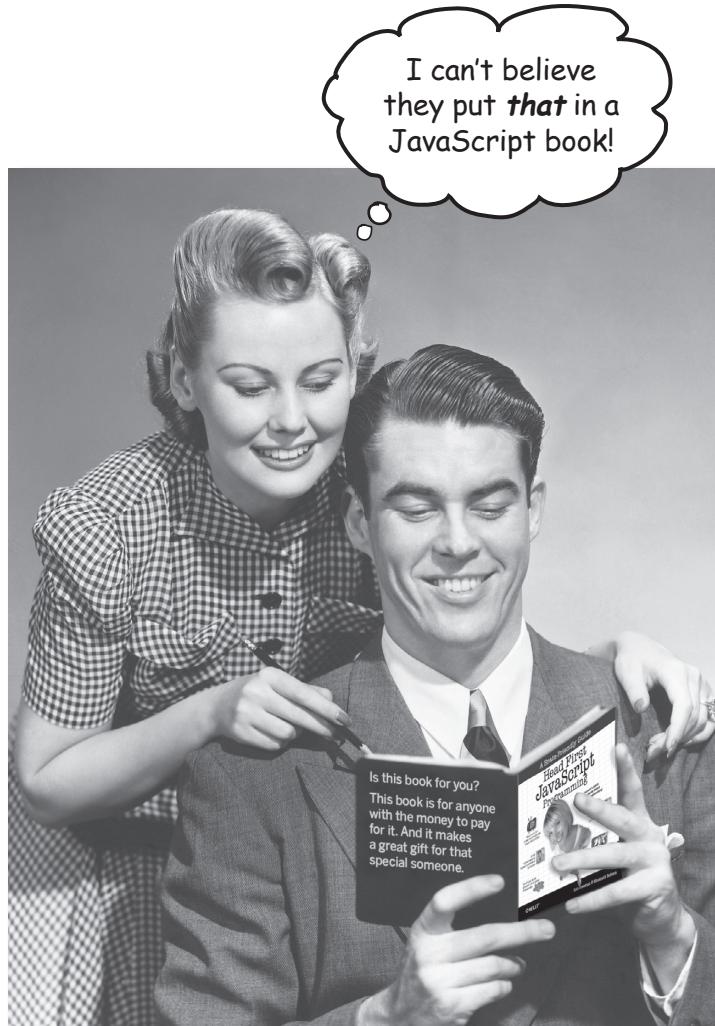


i Index

641

how to use this book

Intro



In this section, we answer the burning question:
"So, why DID they put that in a JavaScript book?"

Who is this book for?

If you can answer “yes” to all of these:

- ① Do you have access to a computer with a **modern web browser** and a **text editor**?
- ② Do you want to **learn, understand and remember** how to **program with JavaScript** using the best techniques and the most recent standards?
- ③ Do you prefer **stimulating dinner party conversation** to **dry, dull, academic lectures**?

this book is for you.



We consider an updated version of Safari, Chrome, Firefox or IE version 9 or newer to be modern.

[Note from marketing: this book is for anyone with a credit card.]

Who should probably back away from this book?

If you can answer “yes” to any one of these:

- ① Are you **completely new to web development**?
Are HTML and CSS foreign concepts to you? If so, you'll probably want to start with *Head First HTML and CSS* to understand how to put web pages together before tackling JavaScript.
- ② Are you a kick-butt web developer looking for a **reference book**?
③ Are you **afraid to try something different**? Would you rather have a root canal than mix stripes with plaid? Do you believe that a technical book can't be serious if JavaScript objects are anthropomorphized?

this book is not for you.



We know what you're thinking.

“How can this be a serious book?”

“What’s with all the graphics?”

“Can I actually learn it this way?”

And we know what your brain is thinking.

Your brain craves novelty. It’s always searching, scanning, *waiting* for something unusual. It was built that way, and it helps you stay alive.

Today, you’re less likely to be a tiger snack. But your brain’s still looking. You just never know.

So what does your brain do with all the routine, ordinary, normal things you encounter? Everything it *can* to stop them from interfering with the brain’s *real* job—recording things that *matter*. It doesn’t bother saving the boring things; they never make it past the “this is obviously not important” filter.

How does your brain *know* what’s important? Suppose you’re out for a day hike and a tiger jumps in front of you. What happens inside your head and body?

Neurons fire. Emotions crank up. *Chemicals surge*.

And that’s how your brain knows...

This must be important! Don’t forget it!

But imagine you’re at home, or in a library. It’s a safe, warm, tiger-free zone. You’re studying. Getting ready for an exam. Or trying to learn some tough technical topic your boss thinks will take a week, ten days at the most.

Just one problem. Your brain’s trying to do you a big favor. It’s trying to make sure that this *obviously* non-important content doesn’t clutter up scarce resources. Resources that are better spent storing the really *big* things. Like tigers. Like the danger of fire. Like how you should never again snowboard in shorts.

And there’s no simple way to tell your brain, “Hey brain, thank you very much, but no matter how dull this book is, and how little I’m registering on the emotional Richter scale right now, I really *do* want you to keep this stuff around.”

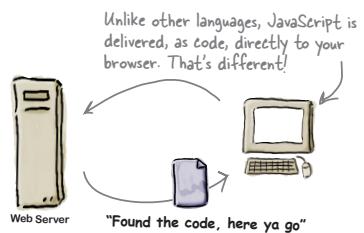


We think of a “Head First” reader as a learner.

So what does it take to *learn* something? First, you have to *get it*, then make sure you *don't forget it*. It's not about pushing facts into your head. Based on the latest research in cognitive science, neurobiology and educational psychology, *learning takes a lot more than text on a page*. We know what turns your brain on.

Some of the Head First learning principles:

Make it visual. Images are far more memorable than words alone, and make learning much more effective (up to 89% improvement in recall and transfer studies). It also makes things more understandable. **Put the words within or near the graphics** they relate to, rather than on the bottom or on another page, and learners will be up to twice as likely to solve problems related to the content.



Use a conversational and personalized style. In recent studies, students performed up to 40% better on post-learning tests if the content spoke directly to the reader, using a first-person, conversational style rather than taking a formal tone. Tell stories instead of lecturing. Use casual language. Don't take yourself too seriously. Which would you pay more attention to: a stimulating dinner party companion, or a lecture?

I really think
JavaScript should go
in the <head> element.



Not so fast! There are
performance and page
loading implications!

Now that I have your
attention, you should be
more careful using global
variables.



Get the learner to think more deeply. In other words, unless you actively flex your neurons, nothing much happens in your head. A reader has to be motivated, engaged, curious and inspired to solve problems, draw conclusions and generate new knowledge. And for that, you need challenges, exercises and thought-provoking questions, and activities that involve both sides of the brain and multiple senses.

Get—and keep—the reader's attention. We've all

had the “I really want to learn this but I can't stay awake past page one”

experience. Your brain pays attention to things that are out of the ordinary, interesting, strange, eye-catching, unexpected. Learning a new, tough, technical topic doesn't have to be boring. Your brain will learn much more quickly if it's not.

Touch their emotions. We now know that your ability to remember something is largely dependent on its emotional content. You remember what you care about. You remember when you *feel* something. No, we're not talking heart-wrenching stories about a boy and his dog. We're talking emotions like surprise, curiosity, fun, “what the...?”, and the feeling of “I Rule!” that comes when you solve a puzzle, learn something everybody else thinks is hard, or realize you know something that “I'm more technical than thou” Bob from engineering doesn't.



Metacognition: thinking about thinking

If you really want to learn, and you want to learn more quickly and more deeply, pay attention to how you pay attention. Think about how you think. Learn how you learn.

Most of us did not take courses on metacognition or learning theory when we were growing up. We were *expected* to learn, but rarely *taught* how to learn.

But we assume that if you're holding this book, you really want to learn how to create JavaScript programs. And you probably don't want to spend a lot of time. And you want to *remember* what you read, and be able to apply it. And for that, you've got to *understand* it. To get the most from this book, or *any* book or learning experience, take responsibility for your brain. Your brain on *this* content.

The trick is to get your brain to see the new material you're learning as Really Important. Crucial to your well-being. As important as a tiger. Otherwise, you're in for a constant battle, with your brain doing its best to keep the new content from sticking.

So how **DO** you get your brain to think JavaScript is as important as a tiger?

There's the slow, tedious way, or the faster, more effective way. The slow way is about sheer repetition. You obviously know that you *are* able to learn and remember even the dullest of topics, if you keep pounding on the same thing. With enough repetition, your brain says, "This doesn't *feel* important to him, but he keeps looking at the same thing *over and over and over*, so I suppose it must be."

The faster way is to do **anything that increases brain activity**, especially different *types* of brain activity. The things on the previous page are a big part of the solution, and they're all things that have been proven to help your brain work in your favor. For example, studies show that putting words *within* the pictures they describe (as opposed to somewhere else in the page, like a caption or in the body text) causes your brain to try to make sense of how the words and picture relate, and this causes more neurons to fire. More neurons firing = more chances for your brain to *get* that this is something worth paying attention to, and possibly recording.

A conversational style helps because people tend to pay more attention when they perceive that they're in a conversation, since they're expected to follow along and hold up their end. The amazing thing is, your brain doesn't necessarily *care* that the "conversation" is between you and a book! On the other hand, if the writing style is formal and dry, your brain perceives it the same way you experience being lectured to while sitting in a roomful of passive attendees. No need to stay awake.

But pictures and conversational style are just the beginning.



Here's what WE did:

We used **pictures**, because your brain is tuned for visuals, not text. As far as your brain's concerned, a picture really *is* worth 1024 words. And when text and pictures work together, we embedded the text *in* the pictures because your brain works more effectively when the text is *within* the thing the text refers to, as opposed to in a caption or buried in the text somewhere.

We used **redundancy**, saying the same thing in *different* ways and with different media types, and *multiple senses*, to increase the chance that the content gets coded into more than one area of your brain.

We used concepts and pictures in **unexpected** ways because your brain is tuned for novelty, and we used pictures and ideas with at least *some emotional content*, because your brain is tuned to pay attention to the biochemistry of emotions. That which causes you to *feel* something is more likely to be remembered, even if that feeling is nothing more than a little **humor**, **surprise** or **interest**.

We used a personalized, **conversational style**, because your brain is tuned to pay more attention when it believes you're in a conversation than if it thinks you're passively listening to a presentation. Your brain does this even when you're *reading*.

We included more than 100 **activities**, because your brain is tuned to learn and remember more when you **do** things than when you *read* about things. And we made the exercises challenging-yet-do-able, because that's what most *people* prefer.

We used **multiple learning styles**, because *you* might prefer step-by-step procedures, while someone else wants to understand the big picture first, while someone else just wants to see a code example. But regardless of your own learning preference, *everyone* benefits from seeing the same content represented in multiple ways.

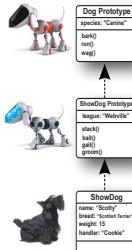
We include content for **both sides of your brain**, because the more of your brain you engage, the more likely you are to learn and remember, and the longer you can stay focused. Since working one side of the brain often means giving the other side a chance to rest, you can be more productive at learning for a longer period of time.

And we included **stories** and exercises that present **more than one point of view**, because your brain is tuned to learn more deeply when it's forced to make evaluations and judgements.

We included **challenges**, with exercises, and by asking **questions** that don't always have a straight answer, because your brain is tuned to learn and remember when it has to *work* at something. Think about it—you can't get your *body* in shape just by *watching* people at the gym. But we did our best to make sure that when you're working hard, it's on the *right* things. That **you're not spending one extra dendrite** processing a hard-to-understand example, or parsing difficult, jargon-laden or overly terse text.

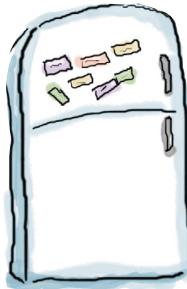
We used **people**. In stories, examples, pictures, etc., because, well, *you're* a person. And your brain pays more attention to *people* than it does to *things*.

We used an **80/20** approach. We assume that if you're going to be a kick-butt JavaScript developer, this won't be your only book. So we don't talk about *everything*. Just the stuff you'll actually *need*.



Be the Browser





Cut this out and stick it
on your refrigerator.

Here's what YOU can do to bend your brain into submission

So, we did our part. The rest is up to you. These tips are a starting point; listen to your brain and figure out what works for you and what doesn't. Try new things.

① Slow down. The more you understand, the less you have to memorize.

Don't just *read*. Stop and think. When the book asks you a question, don't just skip to the answer. Imagine that someone really *is* asking the question. The more deeply you force your brain to think, the better chance you have of learning and remembering.

② Do the exercises. Write your own notes.

We put them in, but if we did them for you, that would be like having someone else do your workouts for you. And don't just *look* at the exercises. **Use a pencil.** There's plenty of evidence that physical activity *while* learning can increase the learning.

③ Read the “There are No Dumb Questions”

That means all of them. They're not optional sidebars—**they're part of the core content!** Don't skip them.

④ Make this the last thing you read before bed. Or at least the last challenging thing.

Part of the learning (especially the transfer to long-term memory) happens *after* you put the book down. Your brain needs time on its own, to do more processing. If you put in something new during that processing-time, some of what you just learned will be lost.

⑤ Drink water. Lots of it.

Your brain works best in a nice bath of fluid. Dehydration (which can happen before you ever feel thirsty) decreases cognitive function.

⑥ Talk about it. Out loud.

Speaking activates a different part of the brain. If you're trying to understand something, or increase your chance of remembering it later, say it out loud. Better still, try to explain it out loud to someone else. You'll learn more quickly, and you might uncover ideas you hadn't known were there when you were reading about it.

⑦ Listen to your brain.

Pay attention to whether your brain is getting overloaded. If you find yourself starting to skim the surface or forget what you just read, it's time for a break. Once you go past a certain point, you won't learn faster by trying to shove more in, and you might even hurt the process.

⑧ Feel something!

Your brain needs to know that this *matters*. Get involved with the stories. Make up your own captions for the photos. Groaning over a bad joke is *still* better than feeling nothing at all.

⑨ Create something!

Apply this to something new you're designing, or rework an older project. Just do *something* to get some experience beyond the exercises and activities in this book. All you need is a pencil and a problem to solve... a problem that might benefit from using JavaScript.

⑩ Get Sleep.

You've got to create a lot of new brain connections to learn to program. Sleep often; it helps.

Read Me

This is a learning experience, not a reference book. We deliberately stripped out everything that might get in the way of learning whatever it is we're working on at that point in the book. And the first time through, you need to begin at the beginning, because the book makes assumptions about what you've already seen and learned.

We teach the GOOD parts of JavaScript, and warn you about the BAD parts.

JavaScript is a programming language that didn't come up through the ivy leagues with plenty of time for academic peer review. JavaScript was thrown out into the world out of necessity and grew up in the early browser neighborhood. So, be warned: JavaScript has some great parts and some not so great parts. But, overall, JavaScript is brilliant, if you use it intelligently.

In this book, we teach you to use the great parts to best advantage, and we'll point out the bad parts, and advise you to drive around them.

We don't exhaustively cover every single aspect of the language.

There's a lot you can learn about JavaScript. This book is not a reference book; it's a learning book, so it doesn't cover everything there is to know about JavaScript. Our goal is to teach you the fundamentals of using JavaScript so that you can pick up any old reference book and do whatever you want with JavaScript.

This book does teach you JavaScript in the browser.

The browser is not only the most common environment that JavaScript runs in, it's also the most convenient (everyone has a computer with a text editor and a browser, and that's all you need to get started with JavaScript). Running JavaScript in the browser also means you get instant gratification: you can write code and all you have to do is reload your web page to see what it does.

This book advocates well-structured and readable code based on best practices.

You want to write code that you and other people can read and understand, code that will still work in next year's browsers. You want to write code in the most straight-forward way so you can get the job done and get on to better things. In this book we're going to teach you to write clear, well-organized code that anticipates change from the get-go. Code you can be proud of, code you'll want to frame and put on the wall (just take it down before you bring your date over).

We encourage you to use more than one browser with this book.

We teach you to write JavaScript that is based on standards, but you're still likely to encounter minor differences in the way web browsers interpret JavaScript. While we'll do our best to ensure all the code in the book works in all modern browsers, and even show you a couple

of tricks to make sure your code is supported by those browsers, we encourage you to pick at least two browsers and test your JavaScript using them. This will give you experience in seeing the differences among browsers and in creating JavaScript code that works well in a variety of browsers with consistent results.

Programming is serious business. You're going to have to work, sometimes hard.

If you've already had some programming experience, then you know what we're talking about. If you're coming straight from *Head First HTML and CSS*, then you're going to find writing code is a little, well, different. Programming requires a different way of thinking. Programming is logical, at times very abstract, and requires you to think in an algorithmic way. But no worries; we're going to do all that in a brain-friendly way. Just take it a bit at a time, make sure you're well nourished and get plenty of sleep. That way, these new programming concepts will really sink in.

The activities are NOT optional.

The exercises and activities in this book are *not* add-ons; they're part of the core content of the book. Some of them are to help with memory, some are for understanding, and some will help you apply what you've learned. Don't skip the exercises. The crossword puzzles are the only things you don't have to do, but they're good for giving your brain a chance to think about the words in a different context.

The redundancy is intentional and important.

One distinct difference in a Head First book is that we want you to really get it. And we want you to finish the book remembering what you've learned. Most reference books don't have retention and recall as a goal, but this book is about learning, so you'll see some of the same concepts come up more than once.

The examples are as lean as possible.

Our readers tell us that it's frustrating to wade through 200 lines of an example looking for the two lines they need to understand. Most examples in this book are shown within the smallest possible context, so that the part you're trying to learn is clear and simple. Don't expect all of the examples to be robust, or even complete—they are written specifically for learning, and aren't always fully-functional.

We've placed all the example files on the Web so you can download them. You'll find them at <http://wickedlysmart.com/hfjs>.

The 'Brain Power' exercises don't usually have answers.

For some of them, there is no right answer, and for others, part of the learning experience of the Brain Power activities is for you to decide if and when your answers are right. In some of the Brain Power exercises you will find hints to point you in the right direction.

We often give you only the code, not the markup.

After we get past the first chapter or two, we often give you just the JavaScript code and assume you'll wrap it in a nice HTML wrapper. Here's a simple HTML page you can use with most of the code in this book, and if we want you to use other HTML, we'll tell you:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Your HTML Page</title>
    <script>
      ← Your JavaScript code will typically go here.
    </script>
  </head>
  <body>
    ← Any web page content will go here.
  </body>
</html>
```

↑
But don't worry; at the beginning of the book we'll take you through everything.

Get the code examples, help and discussion

You'll find everything you need for this book online at <http://wickedlysmart.com/hfjs>, including code sample files and additional support material including videos.

Tech Reviewers



Jeff Straw

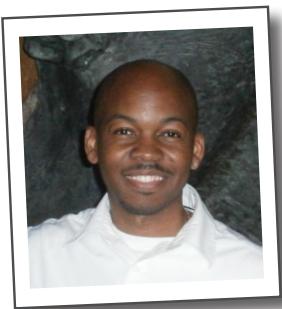


Ismael Martin "Bing" Demiddel

These guys really rocked it; they were there for us throughout the review process and provided invaluable, detailed feedback on everything!



Frank D. Moore



Alfred J. Speller



Bruce Forkush



Javier Ruedas

Thank you to our amazing review team

This book has been more carefully reviewed than any of our previous books. In fact, over 270 people joined our WickedlySmart Insiders program and participated in reading and critiquing this book in real time as we wrote it. This worked better than we ever imagined and was instrumental in shaping every aspect of *Head First JavaScript Programming*. Our heartfelt thanks to everyone who participated; the book is so much better because of you.

The amazing technical reviewers pictured above provided feedback above and beyond, and each made significant contributions to this book. The following reviewers also made contributions across different aspects of the book: **Galina N. Orlova, J. Patrick Kelley, Claus-Peter Kahl, Rob Cleary, Rebeca Dunn-Krahn, Olaf Schoenrich, Jim Cupec, Matthew M. Hanrahan, Russell Alleen-Willems, Christine J. Wilson, Louis-Philippe Breton, Timo Glaser, Charmaine Gray, Lee Beckham, Michael Murphy, Dave Young, Don Smallidge, Alan Rusyak, Eric R. Liscinsky, Brent Fazekas, Sue Starr, Eric (Orange Pants) Johnson, Jesse Palmer, Manabu Kawakami, Alan McIvor, Alex Kelley, Yvonne Bichsel Truhon, Austin Throop, Tim Williams, J. Albert Bowden II, Rod Shelton, Nancy DeHaven Hall, Sue McGee, Francisco Debs, Miriam Berkland, Christine H Grecco, Elhadji Barry, Athanasios Valsamakis, Peter Casey, Dustin Wollam and Robb Kerley.**

Acknowledgments*

We're also extremely grateful to our esteemed technical reviewer **David Powers**.

Powers. The truth is we don't write books without David anymore, he's just saved our butts too many times. It's getting a little like Elton and Bernie; we're starting to ask ourselves if we actually could write a book without him. David helps us forces us to make the book more sound and technically accurate, and his second career as a standup comic really comes in handy when we're tuning the more humorous parts of the book. Thank you once again David—you're the ultimate professional and we sleep better at night knowing we've passed your technical muster.

At O'Reilly:



↑ Meghan Blanchette

A huge, massive thanks to our editor, **Meghan Blanchette**, who cleared the path for this book, removed every obstacle to its completion, waited patiently and sacrificed family

time to get it done. She's also the person who keeps us sane in our relationship with O'Reilly (and keeps O'Reilly sane in their relationship with us). We love you and can't wait to collaborate with you again!



↓ Esteemed Reviewer,
David Powers

And another big shoutout to our Chief Editor Emeritus, **Mike Hendrickson**, who spearheaded this book from the very beginning. Thanks again Mike; none of our books would have happened without you. You've been our champion for well over a decade and we love you for it!



↑ Mike Hendrickson

*The large number of acknowledgments is because we're testing the theory that everyone mentioned in a book acknowledgment will buy at least one copy, probably more, what with relatives and everything. If you'd like to be in the acknowledgment of our *next* book, and you have a large family, write to us.

Also At O'Reilly:

Our sincerest thanks as well to the whole O'Reilly team: **Melanie Yarbrough, Bob Pfahler** and **Dan Fauxsmith**, who wrangled the book into shape; to **Ed Stephenson, Huguette Barriere**, and **Leslie Crandell** who led the way on marketing and we appreciate their out-of-the-box approach. Thanks to **Ellie Volkhausen, Randy Comer** and **Karen Montgomery** for their inspired cover design that continues to serve us well. Thank you, as always, to **Rachel Monaghan** for her hardcore copyedit (and for keeping it all fun), and to **Bert Bates** for his valuable feedback.

1 a quick dip into javascript

Getting your feet wet



Come on in, the water's great! We're going to dive right in and check out JavaScript, write some code, run it and watch it interact with your browser! You're going to be writing code in no time.

JavaScript gives you superpowers. The true programming

language of the web, JavaScript lets you **add behavior** to your web pages. No more dry, boring, static pages that just sit there looking at you—with JavaScript you're going to be able to reach out and touch your users, react to interesting events, grab data from the web to use in your pages, draw graphics right in your web pages and a lot more. And once you know JavaScript you'll also be in a position to create **totally new** behaviors for your users.

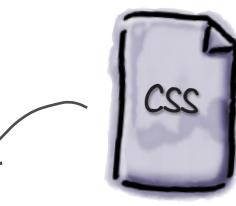
You'll be in good company too, JavaScript's not only one of the **most popular** programming languages, it's also **supported** in all modern (and most ancient) browsers; JavaScript's even branching out and being **embedded** in a lot of environments outside the browser. More on that later; for now, let's get started!

The way JavaScript works

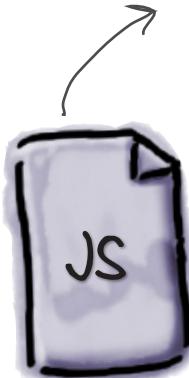
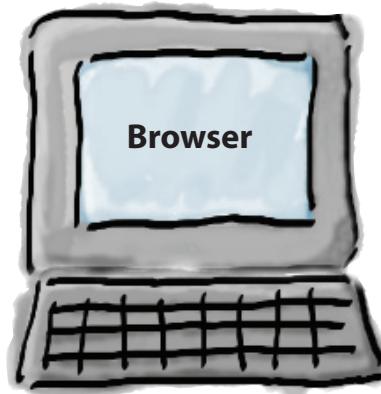
If you're used to creating structure, content, layout and style in your web pages, isn't it time to add a little behavior as well? These days, there's no need for the page to just *sit there*. Great pages should be dynamic, interactive, and they should work with your users in new ways. That's where JavaScript comes in. Let's start by taking a look at how JavaScript fits into the *web page ecosystem*:



You already know we use HTML, or Hypertext Markup Language, to specify all the **content** of your pages along with their **structure**, like paragraphs, headings and sections.



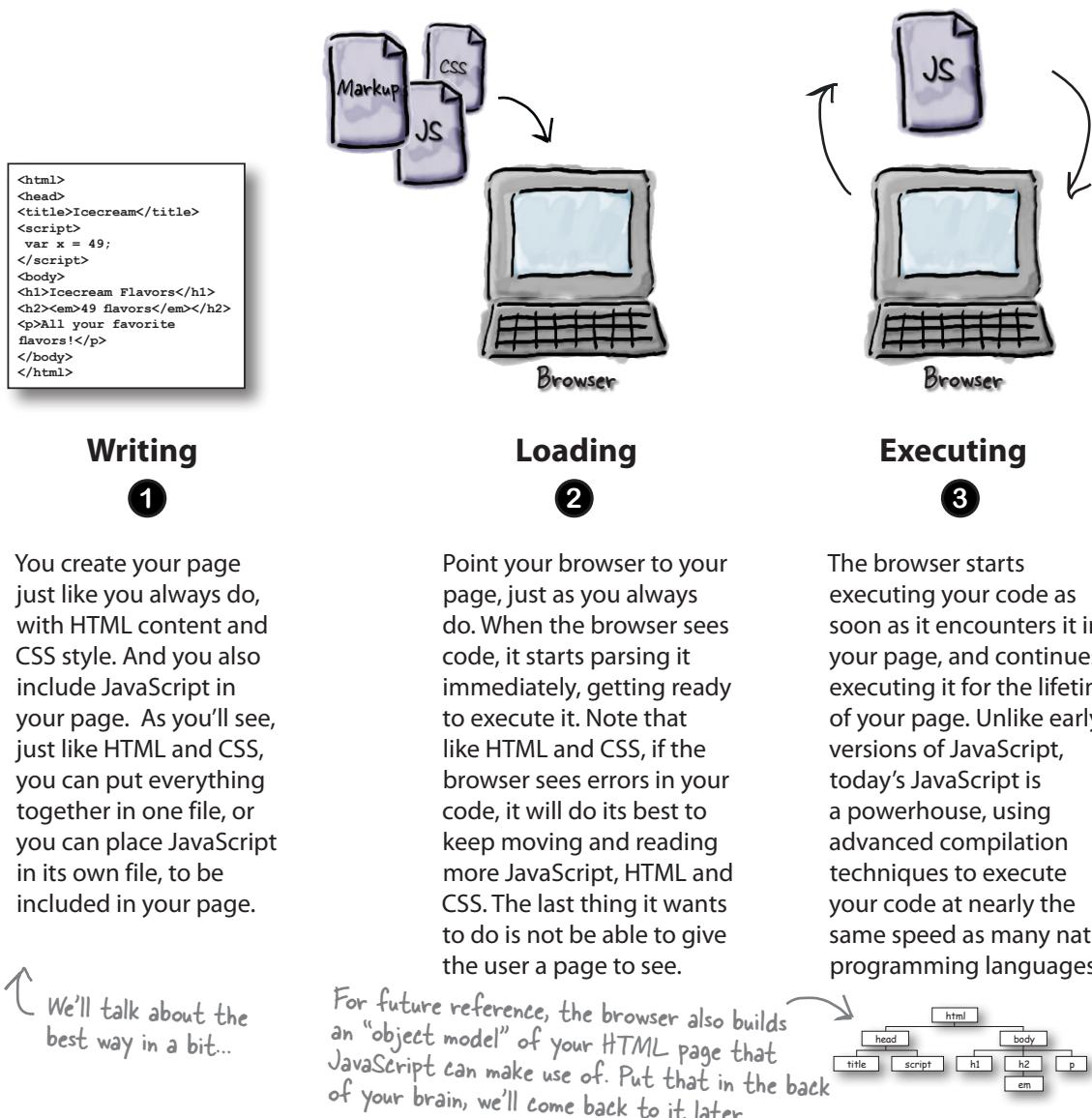
And you already know that we use CSS, or Cascading Style Sheets, to specify how the HTML is presented...the colors, fonts, borders, margins, and the layout of your page. CSS gives you **style**, and it does it in a way that is separate from the structure of the page.



So let's introduce JavaScript, HTML & CSS's computational cousin. JavaScript lets you create **behavior** in your web pages. Need to react when a user clicks on your "On Sale for the next 30 seconds!" button? Double check your user's form input on the fly? Grab some tweets from Twitter and display them? Or how about play a game? Look to JavaScript. JavaScript gives you a way to add programming to your page so that you can compute, react, draw, communicate, alert, alter, update, change, and we could go on... anything dynamic, that's JavaScript in action.

How you're going to write JavaScript

JavaScript is fairly unique in the programming world. With your typical programming language you have to write it, compile it, link it and deploy it. JavaScript is much more fluid and flexible. With JavaScript all you need to do is write JavaScript right into your page, and then load it into a browser. From there, the browser will happily begin executing your code. Let's take a closer look at how this works:



How to get JavaScript into your page

First things first. You can't get very far with JavaScript if you don't know how to get it into a page. So, how do you do that? Using the `<script>` element of course!

Let's take a boring old, garden-variety web page and add some dynamic behavior using a `<script>` element. Now, at this point, don't worry too much about the details of what we're putting into the `<script>` element—your goal right now is to get some JavaScript working.

```
Here's our standard HTML5 doctype, and  
<html> and <head> elements.  
<!doctype html>  
<html lang="en">  
  <head>  
    <meta charset="utf-8">  
    <title>Just a Generic Page</title>  
    <script>  
      setTimeout(wakeUpUser, 5000);  
      function wakeUpUser() {  
        alert("Are you going to stare at this boring page forever?");  
      }  
    </script>  
  </head>  
  <body>  
    <h1>Just a generic heading</h1>  
    <p>Not a lot to read about here. I'm just an obligatory paragraph living in  
    an example in a JavaScript book. I'm looking for something to make my life more  
    exciting.</p>  
  </body>  
</html>
```

And we've got a pretty generic `<body>` for this page as well.

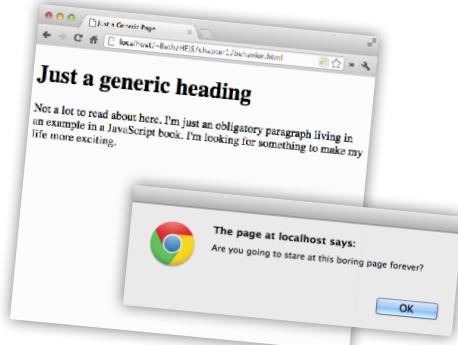
Ah, but we've added a `script` element to the `<head>` of the page.

And we've written some JavaScript code inside it.

Again, don't worry too much about what this code does. Then again, we bet you'll want to take a look at the code and see if you can think through what each part might do.

A little test drive

Go ahead and type this page into a file named “behavior.html”. Drag the file to your browser (or use File > Open) to load it. What does it do? Hint, you'll need to wait five seconds to find out.





Just relax. At this point we don't expect you to read JavaScript like you grew up with it. In fact, all we want you to do right now is get a feel for what JavaScript looks like.

That said, you're not totally off the hook because we need to get your brain revved up and working. Remember that code on the previous page? Let's just walk through it to get a feel for what it might do:

A way to create reusable code and call it "wakeUpUser"?

```
setTimeout(wakeUpUser, 5000);
function wakeUpUser() {
    alert("Are you going to stare at this boring page forever?");
}
```

Perhaps a way to count five seconds of time? Hint:
1000 milliseconds = 1 second.

Clearly a way to alert the user with a message.

there are no Dumb Questions

Q: I've heard JavaScript is a bit of a wimpy language. Is it?

A: JavaScript certainly wasn't a power lifter in its early days, but its importance to the web has grown since then, and as a result, many resources (including brain power from some of the best minds in the business) have gone into supercharging the performance of JavaScript. But, you know what? Even before JavaScript was super fast, it was always a brilliant language. As you'll see, we're going to do some very powerful things with it.

Q: Is JavaScript related to Java?

A: Only by name. JavaScript was created during a time when Java was a red hot popular language, and the inventors of JavaScript capitalized on that popularity by making use of the Java name. Both languages borrow some syntax from programming languages like C, but other than that, they are quite different.

Q: Is JavaScript the best way to create dynamic web pages?
What about solutions like Flash?

A: There was a time when Flash may have been the preferred choice for many to create interactive and more dynamic web pages, but the industry direction is moving strongly in favor of HTML5 with JavaScript. And, with HTML5, JavaScript is now the standard scripting language for the Web. Many resources are going into making JavaScript fast and efficient, and creating JavaScript APIs that extend the functionality of the browser.

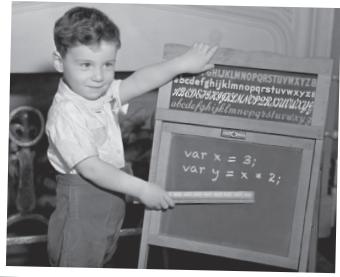
Q: My friend is using JavaScript inside Photoshop, or at least he says he is. Is that possible?

A: Yes, JavaScript is breaking out of the browser as a general scripting language for many applications from graphics utilities to music applications and even to server-side programming. Your investment in learning JavaScript is likely to pay off in ways beyond web pages in the future.

Q: You say that many other languages are compiled. What exactly does that mean and why isn't JavaScript?

A: With conventional programming languages like C, C++ or Java, you compile the code before you execute it. Compiling takes your code and produces a machine efficient representation of it, usually optimized for runtime performance. Scripting languages are typically interpreted, which means that the browser runs each line of JavaScript code as it gets to it. Scripting languages place less importance on runtime performance, and are more geared towards tasks like prototyping, interactive coding and flexibility. This was the case with early JavaScript, and was why, for many years, the performance of JavaScript was not so great. There is a middle ground however; an interpreted language can be compiled on the fly, and that's the path browser manufacturers have taken with modern JavaScript. In fact, with JavaScript you now have the conveniences of a scripting language, while enjoying the performance of a compiled language. By the way, we'll use the words *interpret*, *evaluate* and *execute* in this book. They have slightly different meanings in various contexts, but for our purposes, they all basically mean the same thing.

JavaScript, you've come a long way baby...



JavaScript 1.0

Netscape might have been before your time, but it was the first *real* browser company. Back in the mid-1990s browser competition was fierce, particularly with Microsoft, and so adding new, exciting features to the browser was a priority.

And towards that goal, Netscape wanted to create a scripting language that would allow anyone to add scripts to their pages. Enter LiveScript, a language developed in short order to meet that need. Now if you've never heard of LiveScript, that's because this was all about the time that Sun Microsystems introduced Java, and, as a result, drove their own stock to stratospheric levels. So, why not capitalize on that success and rename LiveScript to JavaScript? After all, who cares if they don't actually have anything to do with each other? Right?

Did we mention Microsoft? They created their own scripting language soon after Netscape did, named, um, JScript, and it was, um, quite similar to JavaScript. And so began the browser wars.



JavaScript 1.3

Between 1996 and 2000, JavaScript grew up. In fact, Netscape submitted JavaScript for standardization and ECMAScript was born. Never heard of ECMAScript? That's okay, now you have; just know that ECMAScript serves as the standard language definition for all JavaScript implementations (in and out of the browser).

During this time developers continued struggling with JavaScript as casualties of the browser wars (because of all the differences in browsers), although the use of JavaScript became common-place in any case. And while subtle differences between JavaScript and JScript continued to give developers headaches, the two languages began to look more and more like each other over time.

JavaScript still hadn't outgrown its reputation as an amateurish language, but that was soon to change...



JavaScript 1.8.5

Finally, JavaScript comes of age and gains the respect of professional developers! While you might say it's all due to having a solid standard, like ECMAScript 5, which is now implemented in all modern browsers, it's really Google that pushed JavaScript usage into the professional limelight, when in 2005 they released Google Maps and showed the world what could really be done with JavaScript to create dynamic web pages.

With all the new attention, many of the best programming language minds focused on improving JavaScript's interpreters and made vast improvements to its runtime performance. Today, JavaScript stands with only a few changes from the early days, and despite its rushed birth into the world, is showing itself to be a powerful and expressive language.

1995

2000

2012

Look how easy it is to write JavaScript



You don't know JavaScript yet, but we bet you can make some good guesses about how JavaScript code works. Take a look at each line of code below and see if you can guess what it does. Write in your answers below. We've done one for you to get you started. If you get stuck, the answers are on the next page.

```
var price = 28.99;
var discount = 10;
var total =
    price - (price * (discount / 100));
if (total > 25) {
    freeShipping();
}

var count = 10;
while (count > 0) {
    juggle();
    count = count - 1;
}

var dog = {name: "Rover", weight: 35};
if (dog.weight > 30) {
    alert("WOOF WOOF");
} else {
    alert("woof woof");
}

var circleRadius = 20;
var circleArea =
    Math.PI * (circleRadius * circleRadius);
```

Create a variable named price, and assign the value 28.99 to it.



Look how easy it is to write JavaScript

```

var price = 28.99;
var discount = 10;
var total =
    price - (price * (discount / 100));
if (total > 25) {
    freeShipping();
}

var count = 10;
while (count > 0) {
    juggle();
    count = count - 1;
}

var dog = {name: "Rover", weight: 35};

if (dog.weight > 30) {
    alert("WOOF WOOF");
} else {
    alert("woof woof");
}

var circleRadius = 20;
var circleArea =
    Math.PI * (circleRadius * circleRadius);
  
```

You don't know JavaScript yet, but we bet you can make some good guesses about how JavaScript code works. Take a look at each line of code below and see if you can guess what it does. Write in your answers below. We've done one for you to get you started. Here are our answers.

| |
|--|
| Create a variable named <code>price</code> , and assign the value <code>28.99</code> to it. |
| Create a variable named <code>discount</code> , and assign the value <code>10</code> to it. |
| Compute a new price by applying a discount and then assign it to the variable <code>total</code> . |
| Compare the value in the variable <code>total</code> to <code>25</code> . If it's greater... |
| ...then do something with <code>freeShipping</code> . |
| End the <code>if</code> statement |
| |
| Create a variable named <code>count</code> , and assign the value <code>10</code> to it. |
| As long as the variable <code>count</code> is greater than <code>0</code> ... |
| ...do some juggling, and... |
| ...reduce the value of <code>count</code> by <code>1</code> each time. |
| End the <code>while</code> loop |
| |
| Create a dog with a name and weight. |
| If the dog's weight is greater than <code>30</code> ... |
| ...alert "WOOF WOOF" to the browser's web page |
| Otherwise... |
| ...alert "woof woof" to the browser's web page |
| End the <code>if/else</code> statement |
| |
| Create a variable, <code>circleRadius</code> , and assign the value <code>20</code> to it. |
| Create a variable named <code>circleArea</code> ... |
| ...and assign the result of this expression to it (<code>1256.6370614359173</code>) |



It's True.

With HTML and CSS you can create some great looking pages. But once you know JavaScript, you can really expand on the kinds of pages you can create. So much so, in fact, you might actually start thinking of your pages as applications (or even experiences!) rather than mere pages.

Now, you might be saying, “Sure, I know that. Why do you think I’m reading this book?” Well, we actually wanted to use this opportunity to have a little chat about learning JavaScript. If you already have a programming language or scripting language under your belt, then you have some idea of what lies ahead. However, if you’ve mostly been using HTML & CSS to date, you should know that there is something fundamentally different about learning a programming language.

With HTML & CSS what you’re doing is largely declarative—for instance, you’re declaring, say, that some text is a paragraph or that all elements in the “sale” class should be colored red. With JavaScript you’re adding *behavior* to the page, and to do that you need to describe computation. You need to be able to describe things like, “compute the user’s score by summing up all the correct answers” or “do this action ten times” or “when the user clicks on that button play the you-have-won sound” or even “go off and get my latest tweet, and put it in this page.”

To do those things you need a language that is quite different from HTML or CSS. Let’s see how...

←
And usually
increase the
size of your
paycheck too!

How to make a statement

When you create HTML you usually **mark up** text to give it structure; to do that you add elements, attributes and values to the text:

```
<h1 class="drink">Mocha Caffe Latte</h1>  
<p>Espresso, steamed milk and chocolate syrup,  
just the way you like it.</p>
```

With HTML we mark up text to create structure. Like, "I need a large heading called Mocha Cafe Latte; it's a heading for a drink. And I need a paragraph after that."

CSS is a bit different. With CSS you're writing a set of **rules**, where each rule selects elements in the page, and then specifies a set of styles for those elements:

```
h1.drink {  
    color: brown;  
}  
  
p {  
    font-family: sans-serif;  
}
```

With CSS we write rules that use **selectors**, like `h1.drink` and `p`, to determine what parts of the HTML the style is applied to.

Let's make sure all drink headings are colored brown...

...and we want all the paragraphs to have a sans-serif type font.

With JavaScript you write **statements**. Each statement specifies a small part of a computation, and together, all the statements create the behavior of the page:

```
var age = 25;  
var name = "Owen";  
  
if (age > 14) {  
    alert("Sorry this page is for kids only!");  
} else {  
    alert("Welcome " + name + "!");  
}
```

A set of statements. Each statement does a little bit of work, like declaring some variables to contain values for us.

Here we create a variable to contain an age of 25, and we also need a variable to contain the value "Owen".

Or making decisions, such as: Is the age of the user greater than 14? And if so alerting the user they are too old for this page.

Otherwise, we welcome the user by name, like this: "Welcome Owen!" (but since Owen is 25, we don't do that in this case.)

Variables and values

You might have noticed that JavaScript statements usually involve variables. Variables are used to store values. What kinds of values? Here are a few examples:

```
var winners = 2;           ← This statement declares a
                           variable named winners and
                           assigns a numeric value of 2 to it.
```



```
var name = "Duke";        ← This one assigns a string of
                           characters to the variable name
                           (we call those "strings," for short).
```



```
var isEligible = false;    ← And this statement assigns
                           the value false to the
                           variable isEligible. We call
                           true/false values "booleans." ←
                           Pronounced "boo-lee-ans."
```

There are other values that variables can hold beyond numbers, strings and booleans, and we'll get to those soon enough, but, no matter what a variable contains, we create all variables the same way. Let's take a little closer look at how to declare a variable:

We always start with the var keyword when declaring a variable. ← NO EXCEPTIONS! Even if JavaScript doesn't complain when you leave off the var. We'll tell you why later... Next we give the variable a name. ← We always end an assignment statement with a semicolon. ←

```
var winners = 2;
```

And, optionally, we assign a value to the variable by adding an equals sign followed by the value.

We say optionally, because if you want, you can create a variable without an initial value, and then assign it a value later. To create a variable without an initial value, just leave off the assignment part, like this:

```
var losers;
```

By leaving off the equals sign and value you're just declaring the variable for later use.



Notice we don't put quotes around boolean values.



Back away from that keyboard!

You know variables have a name, and you know they have a value.

You also know some of the things a variable can hold are numbers, strings and boolean values.

But what can you call your variables? Is any name okay? Well no, but the rules around creating variable names are simple: just follow the two rules below to create valid variable names:

- 1 Start your variables with a letter, an underscore or a dollar sign.**
- 2 After that, use as many letters, numeric digits, underscores or dollar signs as you like.**

Oh, and one more thing; we really don't want to confuse JavaScript by using any of the built-in *keywords*, like **var** or **function** or **false**, so consider those off limits for your own variable names. We'll get to some of these keywords and what they mean throughout the rest of the book, but here's a list to take a quick look at:

| | | | | | |
|----------|---------|------------|-----------|--------|-------|
| break | delete | for | let | super | void |
| case | do | function | new | switch | while |
| catch | else | if | package | this | with |
| class | enum | implements | private | throw | yield |
| const | export | import | protected | true | |
| continue | extends | in | public | try | |
| debugger | false | instanceof | return | typeof | |
| default | finally | interface | static | var | |



*there are no
Dumb Questions*

Q: What's a keyword?

A: A keyword is a reserved word in JavaScript. JavaScript uses these reserved words for its own purposes, and it would be confusing to you and the browser if you started using them for your variables.

Q: What if I used a keyword as part of my variable name? For instance, can I have a variable named `ifOnly` (that is, a variable that contains the keyword `if`)?

A: You sure can, just don't match the keyword exactly. It's also good to write clear code, so in general you wouldn't want to use something like `elze`, which might be confused with `else`.

Q: Is JavaScript case sensitive? In other words, are `myvariable` and `MyVariable` the same thing?

A: If you're used to HTML markup you might be used to case insensitive languages; after all, `<head>` and `<HEAD>` are treated the same by the browser. With JavaScript however, case matters for variables, keywords, function names and pretty much everything else, too. So pay attention to your use of upper- and lowercase.



WEBVILLE TIMES

How to avoid those embarrassing naming mistakes

You've got a lot of flexibility in choosing your variable names, so here are a few Webville tips to make your naming easier:

Choose names that mean something.
Variable names like _m, \$, r and foo might mean something to you but they are generally frowned upon in Webville. Not only are you likely to forget them over time, your code will be much more readable with names like angle, currentPressure and passedExam.

Use "camel case" when creating multiword variable names.

At some point you're going to have to decide how you name a variable that represents, say, a two-headed dragon with fire. How? Just use camel case, in which you capitalize the first letter of each word (other than the first): twoHeadedDragonWithFire. Camel case is easy to form, widely spoken in Webville and gives you enough flexibility to create as specific a variable name as you need. There are other schemes too, but this is one of the more commonly used (even beyond JavaScript).

Use variables that begin with _ and \$ only with very good reason.

Variables that begin with \$ are usually reserved for JavaScript libraries and while some authors use variables beginning with _ for various conventions, we recommend you stay away from both unless you have very good reason (you'll know if you do).

Be safe.

Be safe in your variable naming; we'll cover a few more tips for staying safe later in the book, but for now be clear in your naming, avoid keywords, and always use var when declaring a variable.



Syntax Fun

- Each statement ends in a semicolon.
`x = x + 1;`
- A single line comment begins with two forward slashes. Comments are just notes to you or other developers about the code. They aren't executed.
`// I'm a comment`
- Whitespace doesn't matter (almost everywhere).
`x = 2233;`
- Surround strings of characters with double quotes (or single, both work, just be consistent).
`"You rule!"`
`'And so do you!'`
- Don't use quotes around the boolean values true and false.
`rockin = true;`
- Variables don't have to be given a value when they are declared:
`var width;`
- JavaScript, unlike HTML markup, is case sensitive, meaning upper- and lowercase matters. The variable `counter` is different from the variable `Counter`.

A

```
// Test for jokes  
  
var joke = "JavaScript walked into a bar....";  
var toldJoke = "false";  
var $punchline =  
    "Better watch out for those semi-colons."  
var %entage = 20;  
var result  
  
if (toldJoke == true) {  
    Alert($punchline);  
} else  
    alert(joke);  
}
```



BE the Browser

Below, you'll find JavaScript code with some mistakes in it. Your job is to play like you're the browser and find the errors in the code. After you've done the exercise look at the end of the chapter to see if you found them all.

Don't worry too much about what this JavaScript does for now; just focus on looking for errors in variables and syntax.



B

```
\\" Movie Night  
  
var zip code = 98104;  
var joe'sFavoriteMovie = Forbidden Planet;  
var movieTicket$      =      9;  
  
if (movieTicket$ >= 9) {  
    alert("Too much!");  
} else {  
    alert("We're going to see " + joe'sFavoriteMovie);  
}
```

Express yourself



To truly express yourself in JavaScript you need **expressions**.

Expressions evaluate to values. You've already seen a few in passing in our code examples. Take the expression in this statement for instance:

Here's a JavaScript statement that assigns the result of evaluating an expression to the variable total.

```
var total = price - (price * (discount / 100));
```

Here's our variable total. And the assignment. And this whole thing is an expression.

We use * for multiply and / for divide.

This expression evaluates to a price reduced by a discount that is a percent of the price. So if your price is 10 and the discount is 20, we get 8 as a result.

If you've ever taken a math class, balanced your checkbook or done your taxes, we're sure these kinds of numeric expressions are nothing new.

There are also string expressions; here are a few:

```
"Dear " + "Reader" + ","
"super" + "cali" + youKnowTheRest
phoneNumber.substring(0, 3)
```

This adds together, or concatenates, these strings to form a new string "Dear Reader".

Same here, except we have a variable that contains a string as part of the expression. This evaluates to "supercalifragilisticexpialidocious".*

Just another example of an expression that results in a string. We'll get to exactly how this works later, but this returns the area code of a US phone number string.

We also have expressions that evaluate to **true** or **false**, otherwise known as boolean expressions. Work through each of these to see how you get true or false from them:

```
age < 14
```

If a person's age is less than 14 this is true, otherwise it is false. We could use this to test if someone is a child or not.

```
cost >= 3.99
```

If the cost is 3.99 or greater, this is true. Otherwise it's false. Get ready to buy on sale when it's false!

```
animal == "bear"
```

This is true when animal contains the string "bear". If it does, beware!

And expressions can evaluate to a few other types; we'll get to these later in the book. For now, the important thing is to realize all these expressions evaluate to something: a value that is a number, a string or a boolean. Let's keep moving and see what that gets you!

* Of course, that is assuming the variable youKnowTheRest is "fragilisticexpialidocious".



Sharpen your pencil

Get out your pencil and put some expressions through their paces. For each expression below, compute its value and write in your answer. Yes, WRITE IN... forget what your Mom told you about writing in books and scribble your answer right in this book! Be sure to check your answers at the end of the chapter.

Can you say "Celsius to Fahrenheit calculator"?

`(9 / 5) * temp + 32`

What is the result when temp is 10? _____

This is a boolean expression. The `==` operator tests if two values are equal to each other.

`color == "orange"`

Is this expression true or false when color has the value "pink"? _____

Or has the value "orange"? _____

`name + ", " + "you've won!"`

What value does this compute to when name is "Martha"? _____

`yourLevel > 5`

This tests if the first value is greater than the second. You can also use `>=` to test if the first value is greater than or equal to the second.

`(level * points) + bonus`

When `yourLevel` is 2, what does this evaluate to? _____

When `yourLevel` is 5, what does this evaluate to? _____

When `yourLevel` is 7, what does this evaluate to? _____

`color != "orange"`

Okay, level is 5, points is 30,000 and bonus is 3300. What does this evaluate to? _____

The `!=` operator tests if two values are NOT equal to each other.

Extra CREDIT!

`1000 + "108"`

Are there a few possible answers?
Only one is correct. Which would you choose? _____

Is this expression true or false when color has the value "pink"? _____



Serious Coding

Did you notice that the `=` operator is used in assignments, while the `==` operator tests for equality? That is, we use one equal sign to assign values to variables. We use two equal signs to test if two values are equal to each other. Substituting one for the other is a common coding mistake.



```
while (juggling) {
    keepBallsInAir();
}
```



Doing things more than once

You do a lot of things more than once:

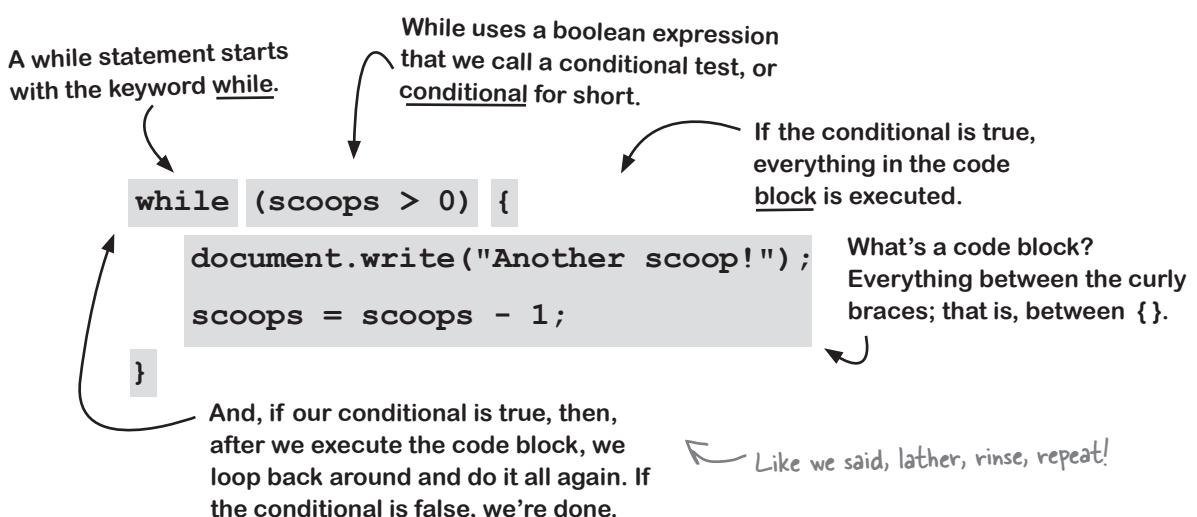
Lather, rinse, repeat...

Wax on, wax off...

Eat candies from the bowl until they're all gone.

Of course you'll often need to do things in code more than once, and JavaScript gives you a few ways to repeatedly execute code in a loop: **while**, **for**, **for in** and **forEach**. Eventually, we'll look at all these ways of looping, but let's focus on **while** for now.

We just talked about expressions that evaluate to boolean values, like `scoops > 0`, and these kinds of expressions are the key to the `while` statement. Here's how:



How the while loop works

Seeing as this is your first while loop, let's trace through a round of its execution to see exactly how it works. Notice we've added a declaration for scoops to declare it, and initialize it to the value 5.

Now let's start executing this code. First we set scoops to five.



```
var scoops = 5;
while (scoops > 0) {
    document.write("Another scoop!<br>");
    scoops = scoops - 1;
}
document.write("Life without ice cream isn't the same");
```

After that we hit the while statement. When we evaluate a while statement the first thing we do is evaluate the conditional to see if it's true or false.

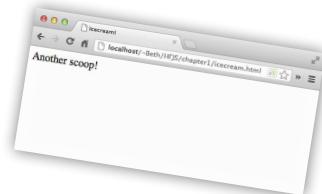
```
var scoops = 5;
while (scoops > 0) {
    document.write("Another scoop!<br>");
    scoops = scoops - 1;
}
document.write("Life without ice cream isn't the same");
```

Is scoops greater than zero? Looks like it to us!



Because the conditional is true, we start executing the block of code. The first statement in the body writes the string "Another scoop!
" to the browser.

```
var scoops = 5;
while (scoops > 0) {
    document.write("Another scoop!<br>");
    scoops = scoops - 1;
}
document.write("Life without ice cream isn't the same");
```



The next statement subtracts one from the number of scoops and then sets scoops to that new value, four.

```
var scoops = 5;
while (scoops > 0) {
  document.write("Another scoop!<br>");
  scoops = scoops - 1;
}
document.write("Life without ice cream isn't the same");
```



That's the last statement in the block, so we loop back up to the conditional and start over again.

```
var scoops = 5;
while (scoops > 0) {
  document.write("Another scoop!<br>");
  scoops = scoops - 1;
}
document.write("Life without ice cream isn't the same");
```

Evaluating our conditional again, this time scoops is four. But that's still more than zero.

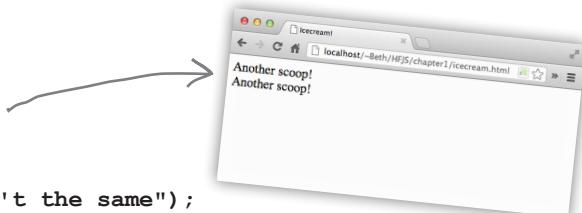
```
var scoops = 5;
while (scoops > 0) {
  document.write("Another scoop!<br>");
  scoops = scoops - 1;
}
document.write("Life without ice cream isn't the same");
```

Still plenty left!



Once again we write the string "Another scoop!
" to the browser.

```
var scoops = 5;
while (scoops > 0) {
  document.write("Another scoop!<br>");
  scoops = scoops - 1;
}
document.write("Life without ice cream isn't the same");
```



javascript while loop

The next statement subtracts one from the number of scoops and sets scoops to that new value, which is three.

```
var scoops = 5;
while (scoops > 0) {
    document.write("Another scoop!<br>");
    scoops = scoops - 1;
}
document.write("Life without ice cream isn't the same");
```



That's the last statement in the block, so we loop back up to the conditional and start over again.

```
var scoops = 5;
while (scoops > 0) {
    document.write("Another scoop!<br>");
    scoops = scoops - 1;
}
document.write("Life without ice cream isn't the same");
```

Evaluating our conditional again, this time scoops is three. But that's still more than zero.

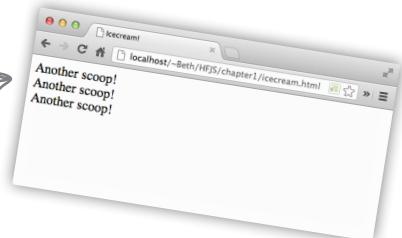
```
var scoops = 5;
while (scoops > 0) {
    document.write("Another scoop!<br>");
    scoops = scoops - 1;
}
document.write("Life without ice cream isn't the same");
```

Still plenty left!



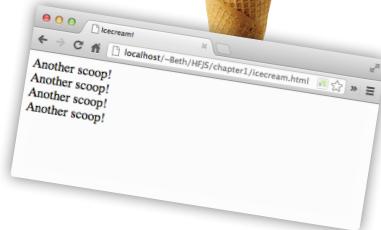
Once again we write the string “Another scoop!
” to the browser.

```
var scoops = 5;
while (scoops > 0) {
    document.write("Another scoop!<br>");
    scoops = scoops - 1;
}
document.write("Life without ice cream isn't the same");
```



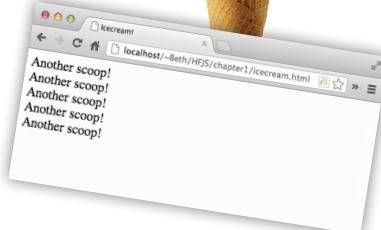
And as you can see, this continues... each time we loop, we decrement (reduce scoops by 1), write another string to the browser, and keep going.

```
var scoops = 5;
while (scoops > 0) {
    document.write("Another scoop!<br>");
    scoops = scoops - 1;
}
document.write("Life without ice cream isn't the same");
```



And continues...

```
var scoops = 5;
while (scoops > 0) {
    document.write("Another scoop!<br>");
    scoops = scoops - 1;
}
document.write("Life without ice cream isn't the same");
```



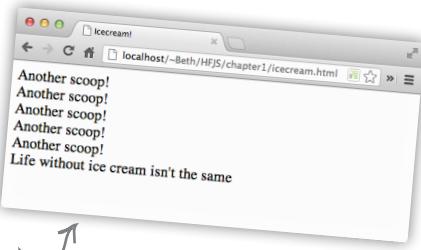
Until the last time... this time something's different. Scoops is zero, and so our conditional returns false. That's it folks; we're not going to go through the loop anymore, we're not going to execute the block. This time, we bypass the block and execute the statement that follows it.

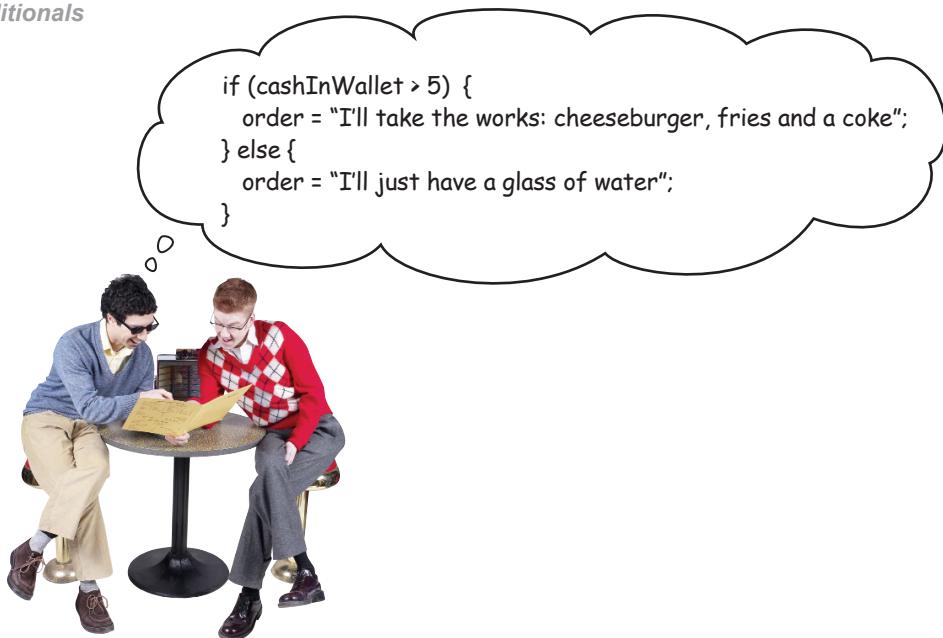
```
var scoops = 5;
while (scoops > 0) {
    document.write("Another scoop!<br>");
    scoops = scoops - 1;
}
document.write("Life without ice cream isn't the same");
```



Now we execute the other `document.write`, and write the string "Life without ice cream isn't the same". We're done!

```
var scoops = 5;
while (scoops > 0) {
    document.write("Another scoop!<br>");
    scoops = scoops - 1;
}
document.write("Life without ice cream isn't the same");
```





Making decisions with JavaScript

You've just seen how you use a conditional to decide whether to continue looping in a `while` statement. You can also use boolean expressions to make decisions in JavaScript with the `if` statement. The `if` statement executes its code block only if a conditional test is true. Here's an example:

Here's the `if` keyword, followed by a conditional and a block of code.

```
if (scoops < 3) {
  alert("Ice cream is running low!");
}
```

This conditional tests to see if we're down to fewer than three scoops.

And if we've got fewer than three left, then we execute the `if` statement's code block.

alert takes a string and displays it in a popup dialog in your browser. Give it a try!

The browser window shows the URL `http://localhost` and the alert message "Ice cream is running low!". There is an "OK" button at the bottom right of the dialog.

With an `if` statement we can also string together multiple tests by adding one or more `else if`'s, like this:

```
if (scoops >= 5) {
  alert("Eat faster, the ice cream is going to melt!");
} else if (scoops < 3) {
  alert("Ice cream is running low!");
}
```

We can have one test, and then another test with `if/else if`

Add as many tests with "`else if`" as you need, each with its own associated code block that will be executed when the condition is true.

And, when you need to make LOTS of decisions

You can string together as many `if/else` statements as you need, and if you want one, even a final catch-all `else`, so that if all conditions fail, you can handle it. Like this:

```

if (scoops >= 5) { ←
    alert("Eat faster, the ice cream is going to melt!");
} else if (scoops == 3) { ← ...or if there are precisely three left...
    alert("Ice cream is running low!");
} else if (scoops == 2) {
    alert("Going once!");
} else if (scoops == 1) {
    alert("Going twice!");
} else if (scoops == 0) {
    alert("Gone!");
} else {
    alert("Still lots of ice cream left, come and get it.");
}

In this code we check to see if there are five or more scoops left...
...or if there are precisely three left...
...or if there are 2, 1 or 0, and then we provide the appropriate alert.
And if none of the conditions above are true, then this code is executed.

```



there are no Dumb Questions

Q: What exactly is a block of code?

A: Syntactically, a block of code (which we usually just call a block) is a set of statements, which could be one statement, or as many as you like, grouped together between curly braces. Once you've got a block of code, all the statements in that block are treated as a group to be executed together. For instance, all the statements within the block in a `while` statement are executed if the condition of the `while` is true. The same holds for a block in an `if` or `else if`.

Q: I've seen code where the conditional is just a variable that is sometimes a string, not a boolean. How does that work?

A: We'll be covering that a little later, but the short answer is JavaScript is quite flexible in what it thinks is a true or false value. For instance, any variable that holds a (non-empty) string is considered true, but a variable that hasn't been set to a value is considered false. We'll get into these details soon enough.

Q: You've said that expressions can result in things other than numbers, strings and booleans. Like what?

A: Right now we're concentrating on what are known as the *primitive types*, that is, numbers, strings and booleans. Later we'll take a look at more complex types, like arrays, which are collections of values, objects and functions.

Q: Where does the name boolean come from?

A: Booleans are named after George Boole, an English mathematician who invented Boolean logic. You'll often see boolean written "Boolean," to signify that these types of variables are named after George.



Code Magnets

A JavaScript program is all scrambled up on the fridge. Can you put the magnets back in the right places to make a working JavaScript program to produce the output shown below?. Check your answer at the end of the chapter before you go on.

Arrange these magnets to make a
working JavaScript program.

```
document.write("Happy Birthday dear " + name + ",<br>");
```

```
document.write("Happy Birthday to you.<br>");
```

```
var i = 0;
```

```
var name = "Joe";
```

```
i = i + 1;
```

```
}
```

```
document.write("Happy Birthday to you.<br>");
```

```
while (i < 2) {
```

↓ Your unscrambled program
should produce this output.

The screenshot shows a web browser window with the title "Happy Birthday". The address bar displays "localhost/~Beth/HFJS/chapter1/birthday.html". The main content area of the browser shows the following text:
Happy Birthday to you.
Happy Birthday to you.
Happy Birthday dear Joe,
Happy Birthday to you.



↑
Use this space for your
re-arranged magnets.

Reach out and communicate with your user

We've been talking about making your pages more interactive, and to do that you need to be able to communicate with your user. As it turns out there are a few ways to do that, and you've already seen some of them. Let's get a quick overview and then we'll dive into these in more detail throughout the book:

Create an alert.

As you've seen, the browser gives you a quick way to alert your users through the `alert` function. Just call `alert` with a string containing your alert message, and the browser will give your user the message in a nice dialog box. A small confession though: we've been overusing this because it's easy; `alert` really should be used only when you truly want to stop everything and let the user know something.

Write directly into your document.

Think of your web page as a document (that's what the browser calls it). You can use a function `document.write` to write arbitrary HTML and content into your page at any point. In general, this is considered bad form, although you'll see it used here and there. We've used it a bit in this chapter too because it's an easy way to get started.

Use the console.

Every JavaScript environment also has a console that can log messages from your code. To write a message to the console's log you use the function `console.log` and hand it a string that you'd like printed to the log (more details on using `console.log` in a second). You can view `console.log` as a great tool for troubleshooting your code, but typically your users will never see your console log, so it's not a very effective way to communicate with them.

Directly manipulate your document.

This is the big leagues; this is the way you want to be interacting with your page and users—using JavaScript you can access your actual web page, read & change its content, and even alter its structure and style! This all happens by making use of your browser's *document object model* (more on that later). As you'll see, this is the best way to communicate with your user. But, using the document object model requires knowledge of how your page is structured and of the programming interface that is used to read and write to the page. We'll be getting there soon enough. But first, we've got some more JavaScript to learn.



← We're using these three methods in this chapter.

← The console is a really handy way to help find errors in your code! If you've made a typing mistake, like missing a quote, JavaScript will usually give you an error in the console to help you track it down.

← This is what we're working towards. When you get there you'll be able to read, alter and manipulate your page in any number of ways.

* WHO DOES WHAT? *

All our methods of communication have come to the party with masks on. Can you help us unmask each one? Match the descriptions on the right to the names on the left. We've done one for you.

document.write

I'll stop your user in his tracks and deliver a short message. The user has to click on "ok" to go further.

console.log

I can insert a little HTML and text into a document. I'm not the most elegant way to get a message to your users, but I work on every browser.

alert

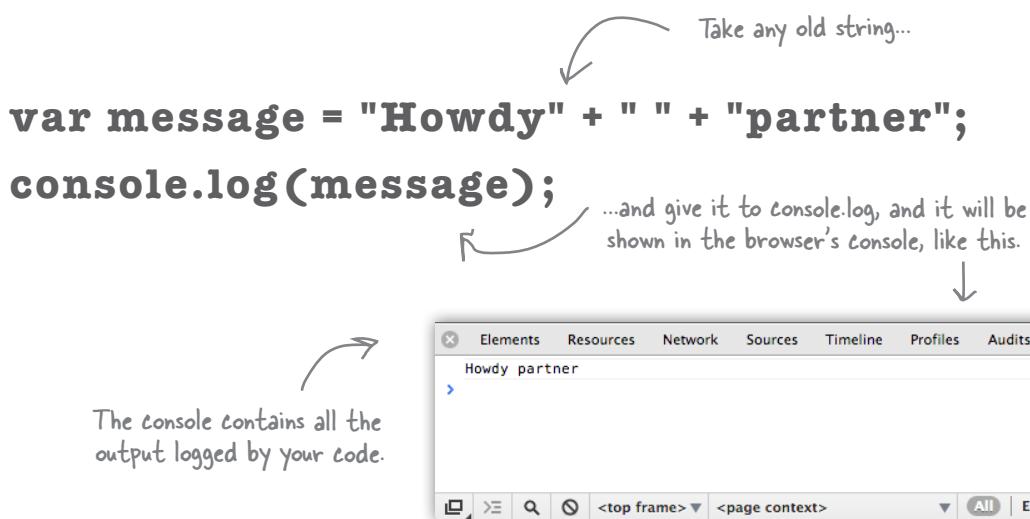
Using me you can totally control a web page: get values that a user typed in, alter the HTML or the style, or update the content of your page.

document object model

I'm just here for simple debugging purposes. Use me and I can write out information to a special developer's console.

A closer look at `console.log`

Let's take a closer look at how `console.log` works so we can use it in this chapter to see the output from our code, and throughout the book to inspect the output of our code and debug it. Remember though, the console is not a browser feature most casual users of the web will encounter, so you won't want to use it in the final version of your web page. Writing to the console log is typically done to troubleshoot as you develop your page. That said, it's a great way to see what your code is doing while you're learning the basics of JavaScript. Here's how it works:



there are no Dumb Questions

Q: I get that `console.log` can be used to output strings, but what exactly is it? I mean why are the "console" and the "log" separated by a period?

A: Ah, good point. We're jumping ahead a bit, but think of the console as an object that does things, console-like things. One of those things is logging, and to tell the console to log for us, we use the syntax "`console.log`" and pass it our output in between parentheses. Keep that in the

back of your mind; we're coming back to talk a lot more about objects a little later in the book. For now, you've got enough to use `console.log`.

Q: Can the console do anything other than just log?

A: Yes, but typically people just use it to log. There are a few more advanced ways to use log (and console), but they tend to be browser-specific. Note that console is

something all modern browsers supply, but it isn't part of any formal specification.

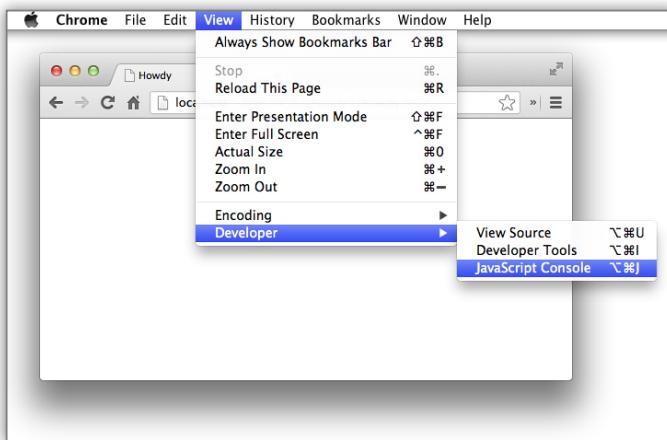
Q: Uh, console looks great, but where do I find it? I'm using it in my code and I don't see any output!

A: In most browsers you have to explicitly open the console window. Check out the next page for details.

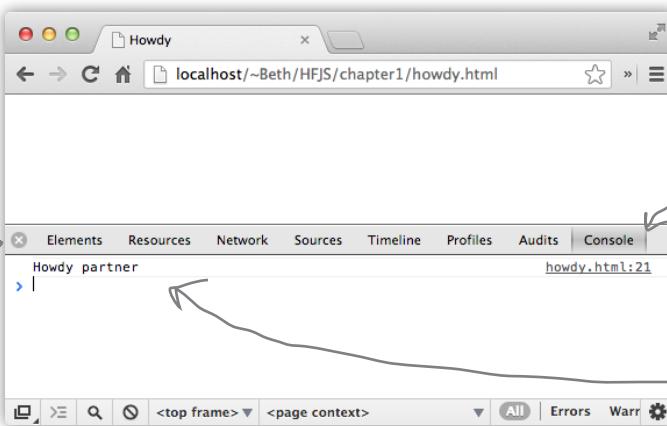
Opening the console

Every browser has a slightly different implementation of the console. And, to make things even more complicated, the way that browsers implement the console changes fairly frequently—not in a huge way, but enough so that by the time you read this, your browser's console might look a bit different from what we're showing here.

So, we're going to show you how to access the console in the Chrome browser (version 25) on the Mac, and we'll put instructions on how to access the console in all the major browsers online at <http://wickedlysmart.com/hfsconsole>. Once you get the hang of the console in one browser, it's fairly easy to figure out how to use it in other browsers too, and we encourage you to try using the console in at least two browsers so you're familiar with them.



To access the console in Chrome (on the Mac), use the View > Developer > JavaScript Console menu.



The console will appear in the bottom part of your browser window.

Make sure the Console tab is selected in the tab bar along the top of the console.

You should see any messages you give to `console.log` in your code displayed in the window here.

Don't worry about what these other tabs are for. They're useful, but the most important one now is Console, so we can see `console.log` messages from our code.

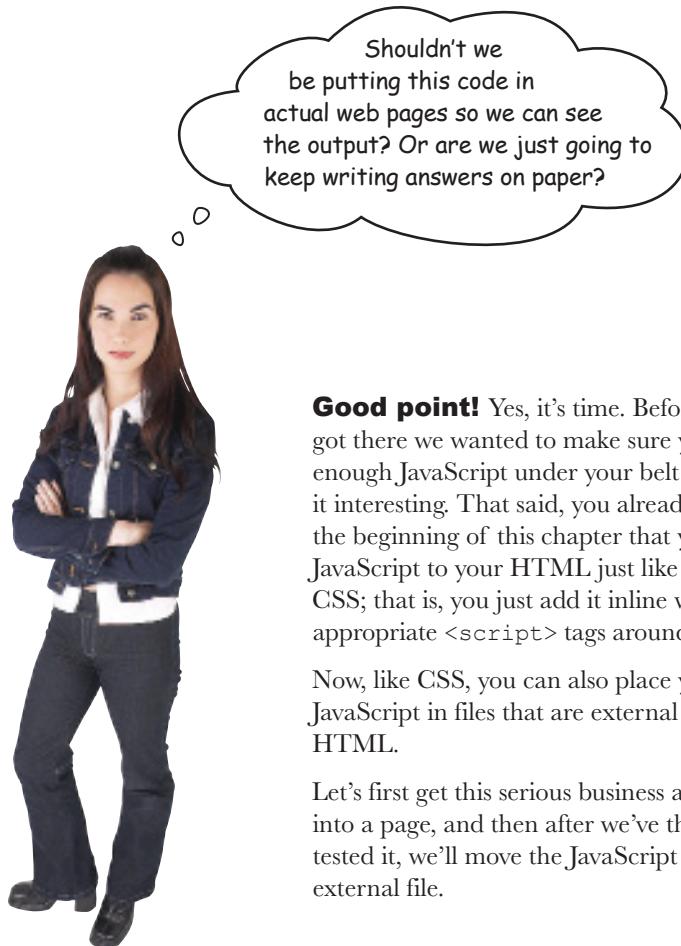
Coding a Serious JavaScript Application

Let's put all these new JavaScript skills and `console.log` to good use with something practical. We need some variables, a `while` statement, some `if` statements with `elses`. Add a little more polish and we'll have a super-serious business application before you know it. But, before you look at the code, think to yourself how you'd code that classic favorite, "99 bottles of beer."

```
var word = "bottles";
var count = 99;
while (count > 0) {
    console.log(count + " " + word + " of beer on the wall");
    console.log(count + " " + word + " of beer,");
    console.log("Take one down, pass it around,");
    count = count - 1;
    if (count > 0) {
        console.log(count + " " + word + " of beer on the wall.");
    } else {
        console.log("No more " + word + " of beer on the wall.");
    }
}
```



There's still a little flaw in our code. It runs correctly, but the output isn't 100% perfect. See if you can find the flaw, and fix it.



Good point! Yes, it's time. Before we got there we wanted to make sure you had enough JavaScript under your belt to make it interesting. That said, you already saw in the beginning of this chapter that you add JavaScript to your HTML just like you add CSS; that is, you just add it inline with the appropriate `<script>` tags around it.

Now, like CSS, you can also place your JavaScript in files that are external to your HTML.

Let's first get this serious business application into a page, and then after we've thoroughly tested it, we'll move the JavaScript out to an external file.



A Test Drive

Okay, let's get some code in the browser... follow the instructions below and get your serious business app launched! You'll see our result below:

↑ To download all the code and sample files for this book, please visit <http://wickedlysmart.com/hfjs>.

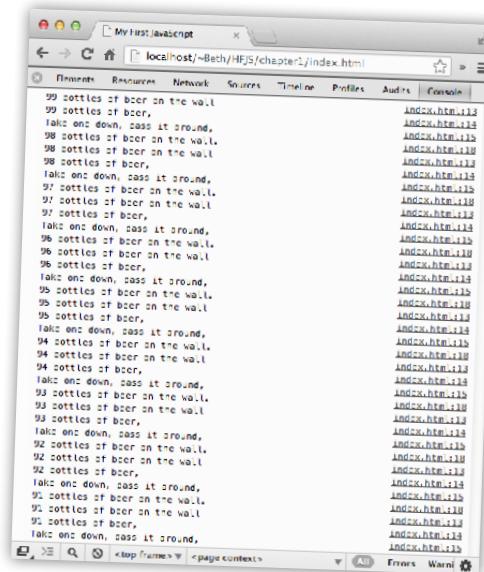
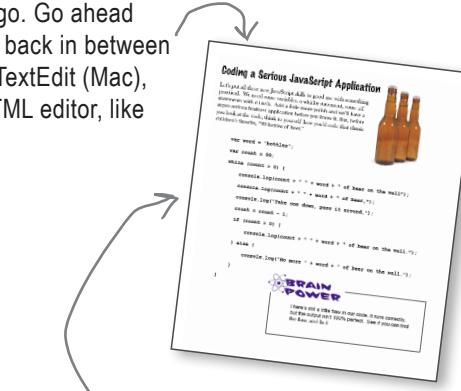
- Check out the HTML below; that's where your JavaScript's going to go. Go ahead and type in the HTML and then place the JavaScript from two pages back in between the `<script>` tags. You can use an editor like Notepad (Windows) or TextEdit (Mac), making sure you are in plain text mode. Or, if you have a favorite HTML editor, like Dreamweaver, Coda or WebStorm, you can use that too.

```
<!doctype html> ← Type this in.
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>My First JavaScript</title>
  </head>
  <body>
    <script>
    </script>
  </body>
</html>
```

Here are the `<script>` tags. At this point you know that's where you should put your code.

- Save the file as "index.html".
- Load the file into your browser. You can either drag the file right on top of your browser window, or use the File > Open (or File > Open File) menu option in your favorite browser.
- You won't see anything in the web page itself because we're logging all the output to the console, using `console.log`. So open up the browser's console, and congratulate yourself on your serious business application.

Here's our test run of this code. The code creates the entire lyrics for the 99 bottles of beer song and logs the text to the browser's console.



How do I add code to my page? (let me count the ways)

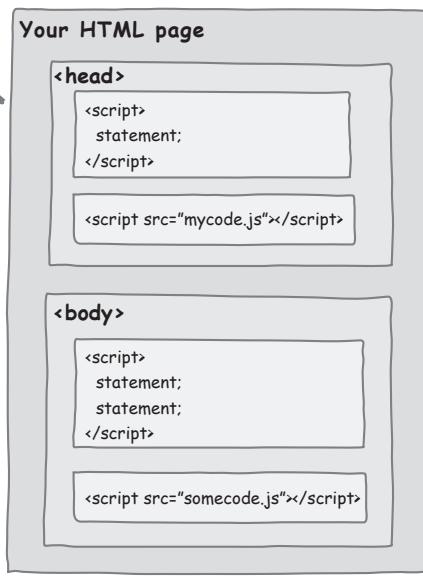
You already know you can add the `<script>` element with your JavaScript code to the `<head>` or `<body>` of your page, but there are a couple of other ways to add your code to a page. Let's check out all the places you can put JavaScript (and why you might want to put it one place over another):

You can place your code inline, in the `<head>` element. The most common way to add code to your pages is to put a `<script>` element in the `<head>`. Sure, it makes your code easy to find and seems to be a logical place for your code, but it's not always the best place. Why? Read on...

Or, you can add your code inline in the body of the document. To do this, enclose your JavaScript code in the `<script>` element and place it in the `<body>` of your page (typically at the end of the body).

This is a little better. Why? When your browser loads a page, it loads everything in your page's `<head>` before it loads the `<body>`. So, if your code is in the `<head>`, users might have to wait a while to see the page. If the code is loaded after the HTML in the `<body>`, users will get to see the page content while they wait for the code to load.

Still, is there a better way? Read on...



Or, put your code in its own file and link to it from the `<head>`.

This is just like linking to a CSS file. The only difference is that you use the `src` attribute of the `<script>` tag to specify the URL to your JavaScript file.

When your code is in an external file, it's easier to maintain (separately from the HTML) and can be used across multiple pages. But this method still has the drawback that all the code needs to be loaded before the body of the page. Is there a better way? Read on...

Finally, you can link to an external file in the body of your page. Ahhh, the best of both worlds. We have a nice, maintainable JavaScript file that can be included in any page, and it's referenced from the bottom of the body of the page, so it's only loaded after the body of the page. Not bad.

Despite evidence to the contrary, I still think the `<head>` is a great place for code.



We're going to have to separate you two

Going separate ways hurts, but we know we have to do it. It's time to take your JavaScript and move it into its own file. Here's how you do that...

- ① Open index.html and select all the code; that is, everything between the `<script>` tags. Your selection should look like this:

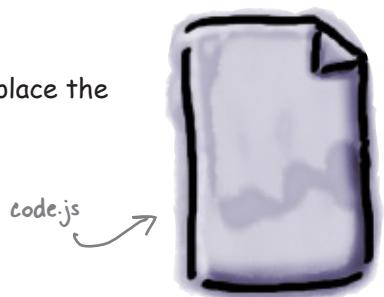
```

<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>My First JavaScript</title>
  </head>
  <body>
    <script>
      var word = "bottles";
      var count = 99;
      while (count > 0) {
        console.log(count + " " + word + " of beer on the wall");
        console.log(count + " " + word + " of beer,");
        console.log("Take one down, pass it around,");
        count = count - 1;
        if (count > 0) {
          console.log(count + " " + word + " of beer on the wall.");
        } else {
          console.log("No more " + word + " of beer on the wall.");
        }
      }
    </script>
  </body>
</html>

```

Select just the code, not the `<script>` tags;
you won't need those where you're going...

- ② Now create a new file named "code.js" in your editor, and place the code into it. Then save "code.js".



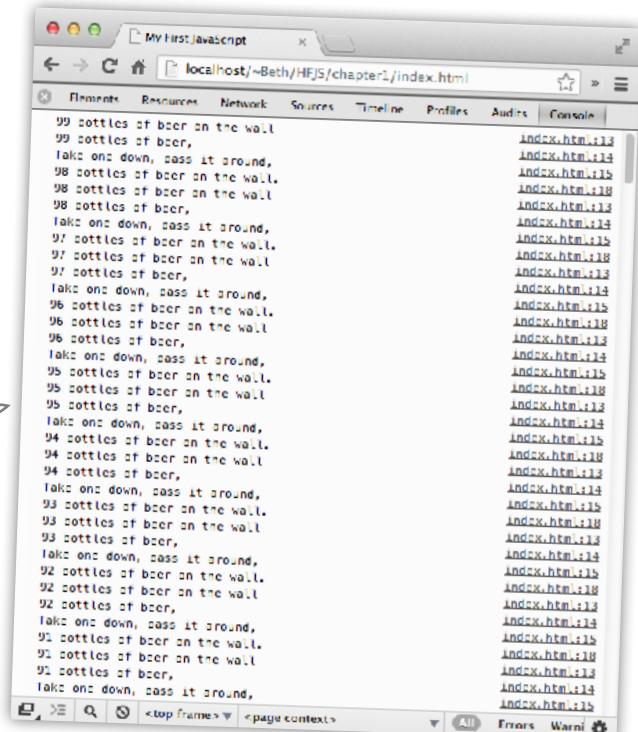
- ③ Now we need to place a reference to the "code.js" file in "index.html" so that it's retrieved and loaded when the page loads. To do that, delete the JavaScript code from "index.html", but leave the `<script>` tags. Then add a `src` attribute to your opening `<script>` tag to reference "code.js".

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>My First JavaScript</title>
  </head>
  <body>
    <script src="code.js">
      </script> Where your code was.
    </body>
  </html> Believe it or not we still need the ending <script> tag, even if
         there is no code between the two tags.
```

Use the `src` attribute of the `<script>` element to link to your JavaScript file.

- ④ That's it, the surgery is complete. Now you need to test it. Reload your "index.html" page and you should see exactly the same result as before. Note that by using `src="code.js"`, we're assuming that the code file is in the same directory as the HTML file.

You should get the same result as before. But now your HTML and JavaScript are in separate files. Doesn't that just feel cleaner, more manageable, more stress-free already?





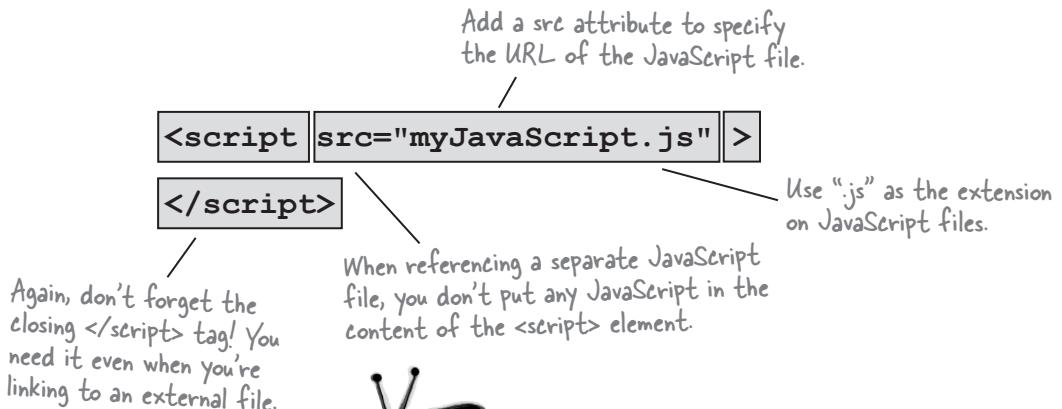
Anatomy of a Script Element

You know how to use the `<script>` element to add code to your page, but just to really nail down the topic, let's review the `<script>` element to make sure we have every detail covered:

The `type` attribute tells the browser you're writing JavaScript. The thing is, browsers assume you're using JavaScript if you leave it off. So, we recommend you leave it off, and so do the people who write the standards.



And when you are referencing a separate JavaScript file from your HTML, you'll use the `<script>` element like this:



Watch it!

You can't use inline and external together.

If you try throwing some quick code in between those `<script>` tags when you're already using a `src` attribute, it won't work. You'll need two separate `<script>` elements.

```
<script src="goodies.js">
    var = "quick hack";
</script>
```

WRONG



JavaScript Exposed

This week's interview:
Getting to know JavaScript

Head First: Welcome JavaScript. We know you're super busy out there, working on all those web pages, so we're glad you could take time out to talk to us.

JavaScript: No problem. And, I *am* busier than ever these days; people are using JavaScript on just about every page on the Web nowadays, for everything from simple menu effects to full blown games. It's nuts!

Head First: That's amazing given that just a few years ago, someone said that you were just a "half-baked, wimpy scripting language" and now you're everywhere.

JavaScript: Don't remind me. I've come a long way since then, and many great minds have been hard at work making me better.

Head First: Better how? Seems like your basic language features are about the same...

JavaScript: Well, I'm better in a couple of ways. First of all, I'm lightning fast these days. While I'm considered a scripting language, now my performance is close to that of native compiled languages.

Head First: And second?

JavaScript: My ability to do things in the browser has expanded dramatically. Using the JavaScript libraries available in all modern browsers you can find out your location, play video and audio, paint graphics on your web page and a lot more. But if you wanna do all that you have to know JavaScript.

Head First: But back to those criticisms of you, the language. I've heard some not so kind words... I believe the phrase was "hacked up language."

JavaScript: I'll stand on my record. I'm pretty much one of, if not *the* most widely used languages in the world. I've also fought off many competitors and won. Remember Java in the browser? Ha, what a joke. VBScript? Ha. JScript? Flash?! Silverlight? I could go on and on. So, tell me, how bad could I be?

Head First: You've been criticized as, well, "simplistic."

JavaScript: Honestly, it's my greatest strength. The fact that you can fire up a browser, type in a few lines of JavaScript and be off and running, that's powerful. And it's great for beginners too. I've heard some say there's no better beginning language than JavaScript.

Head First: But simplicity comes at a cost, no?

JavaScript: Well that's the great thing, I'm simple in the sense you can get a quick start. But I'm deep and full of all the latest modern programming constructs.

Head First: Oh, like what?

JavaScript: Well, for example, can you say dynamic types, first-class functions and closures?

Head First: I can say it but I don't know what they are.

JavaScript: Figures... that's okay, if you stay with the book you will get to know them.

Head First: Well, give us the gist.

JavaScript: Let me just say this, JavaScript was built to live in a dynamic web environment, an exciting environment where users interact with a page, where data is coming in on the fly, where many types of events happen, and the language reflects that style of programming. You'll get it a little more a bit later in the book when you understand JavaScript more.

Head First: Okay, to hear you tell it, you're the perfect language. Is that right?

JavaScript tears up...

JavaScript: You know, I didn't grow up within the ivy-covered walls of academia like most languages. I was born into the real world and had to sink or swim very fast in my life. Given that, I'm not perfect; I certainly have a few "bad parts."

Head First with a slight Barbara Walters smile:

We've seen a new side of you today. I think this merits another interview in the future. Any parting thoughts?

JavaScript: Don't judge me by my bad parts, learn the good stuff and stick with that!



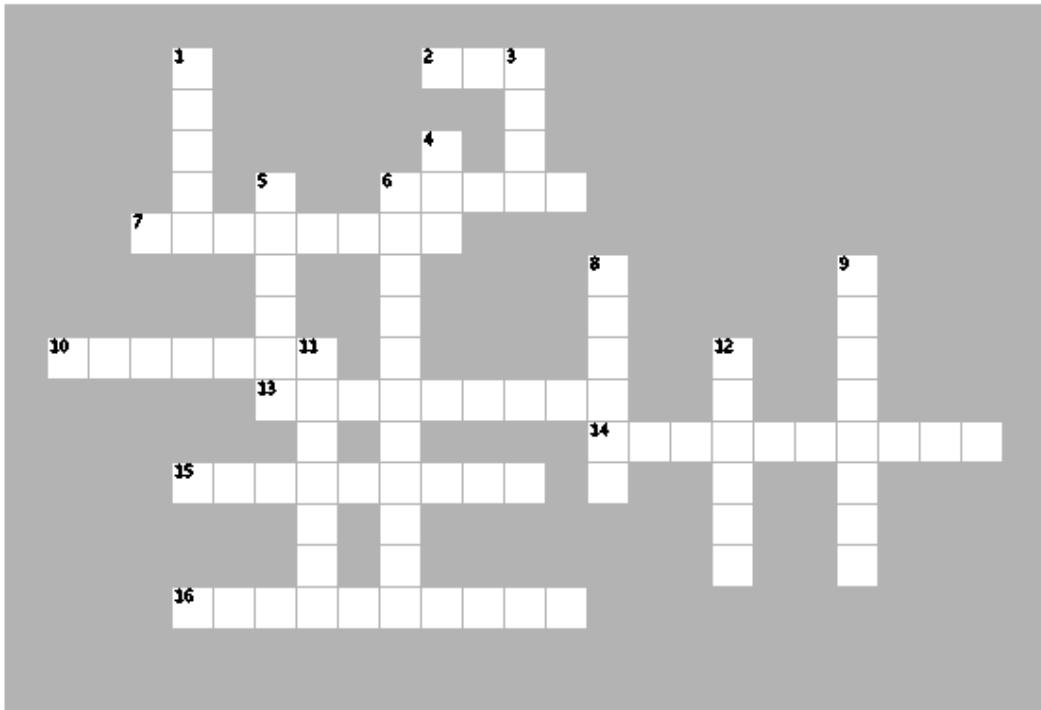
BULLET POINTS

- JavaScript is used to add **behavior** to web pages.
- Browser engines are much faster at executing JavaScript than they were just a few years ago.
- Browsers begin executing JavaScript code as soon as they encounter the code in the page.
- Add JavaScript to your page with the `<script>` element.
- You can put your JavaScript inline in the web page, or link to a separate file containing your JavaScript from your HTML.
- Use the `src` attribute in the `<script>` tag to link to a separate JavaScript file.
- HTML **declares** the structure and content of your page; JavaScript **computes** values and adds behavior to your page.
- JavaScript programs are made up of a series of **statements**.
- One of the most common JavaScript statements is a variable declaration, which uses the `var` keyword to declare a new variable and the assignment operator, `=`, to assign a value to it.
- There are just a few rules and guidelines for naming JavaScript variables, and it's important that you follow them.
- Remember to avoid JavaScript keywords when naming variables.
- JavaScript expressions compute values.
- Three common types of expressions are **numeric**, **string** and **boolean** expressions.
- **if/else** statements allow you to make decisions in your code.
- **while/for** statements allow you to execute code many times by looping.
- Use `console.log` instead of `alert` to display messages to the Console.
- Console messages should be used primarily for troubleshooting as users will most likely never see console messages.
- JavaScript is most commonly found adding behavior to web pages, but is also used to script applications like Adobe Photoshop, OpenOffice and Google Apps, and is even used as a server-side programming language.



JavaScript cross

Time to stretch your dendrites with a puzzle to help it all sink in.



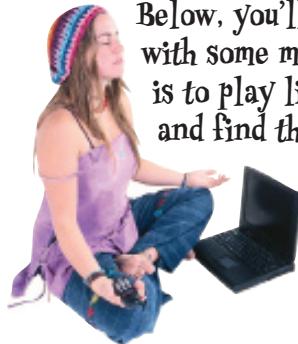
ACROSS

2. To link to an external JavaScript file from HTML, you need the _____ attribute for your <script> element.
6. To avoid embarrassing naming mistakes, use _____ case.
7. JavaScript adds _____ to your web pages.
10. There are 99 _____ of beer on the wall.
13. Each line of JavaScript code is called a _____.
14. $3 + 4$ is an example of an _____.
15. All JavaScript statements end with a _____.
16. Use _____ to troubleshoot your code.

DOWN

2. Do things more than once in a JavaScript program with the _____ loop.
3. JavaScript variable names are _____ sensitive.
4. To declare a variable, use this keyword.
5. Variables are used to store these.
6. Each time through a loop, we evaluate a _____ expression.
8. Today's JavaScript runs a lot _____ than it used to.
9. The if/else statement is used to make a _____.
11. You can concatenate _____ together with the + operator.
12. You put your JavaScript inside a _____ element.

BE the Browser Solution



Below, you'll find JavaScript code with some mistakes in it. Your job is to play like you're the browser and find the errors in the code.

After you've done the exercise look at the end of the chapter to see if you found them all. Here's our solution.

A

```
// Test for jokes
var joke = "JavaScript walked into a bar....";
var toldJoke = "false";
var $punchline = It's okay, but not recommended, to begin a variable with a $.
    "Better watch out for those semi-colons." Don't forget to end
var %entage = 20; Can't use % in variable names. statements with a semi-colon!
var result Another missing semi-colon.

if (toldJoke == true) {
    Alert($punchline); Should be alert, not Alert.
} else
    alert(joke); We're missing an
} opening brace here.
```

B

```
\\" Movie Night Comments should begin with // not \\".
var zip code = 98104; No spaces allowed in variable names.
var joe'sFavoriteMovie = Forbidden Planet; No quotes allowed
var movieTicket$      =      9; in variable names. But we do need quotes
                                around the string
                                "Forbidden Planet".
if (movieTicket$ >= 9) {
    alert("Too much!");
} else {
    alert("We're going to see " + joe'sFavoriteMovie);
}
```

This if/else doesn't work because of the invalid variable name here.

Delimit your strings with two double quotes ("") or two single quotes (''). Don't mix!

Don't put quotes around boolean values unless you really want a string.

It's okay, but not recommended, to begin a variable with a \$.

Don't forget to end statements with a semi-colon!

Another missing semi-colon.

Should be alert, not Alert.
JavaScript is case-sensitive.

We're missing an
opening brace here.

Comments should begin with // not \\".

No spaces allowed in variable names.

No quotes allowed in variable names.

"Forbidden Planet".



Sharpen your pencil

Solution

Get out your pencil and let's put some expressions through their paces. For each expression below, compute its value and write in your answer. Yes, WRITE IN... forget what your Mom told you about writing in books and scribble your answer right in this book! Here's our solution.

Can you say "Celsius to Fahrenheit calculator"?

`(9 / 5) * temp + 32`

This is a boolean expression. The `==` operator tests if two values are equal to each other.

`color == "orange"`

`name + ", " + "you've won!"`

`yourLevel > 5`

This tests if the first value is greater than the second. You can also use `>=` to test if the first value is greater than or equal to the second.

`(level * points) + bonus`

`color != "orange"`

The `!=` operator tests if two values are NOT equal to each other.

Extra CREDIT!

`1000 + "108"`

Are there a few possible answers? Only one is correct. Which would you choose? "1000108"

What is the result when temp is 10? 50

Is this expression true or false when color has the value "pink"? false

Or, has the value "orange"? true

What value does this compute to when name is "Martha"?
"Martha, you've won!"

When `yourLevel` is 2, what does this evaluate to? false

When `yourLevel` is 5, what does this evaluate to? false

When `yourLevel` is 7, what does this evaluate to? true

Okay, `level` is 5, `points` is 30,000 and `bonus` is 3300. What does this evaluate to? 153300

Is this expression true or false when color has the value "pink"? true



Serious Coding

Did you notice that the `=` operator is used in assignments, while the `==` operator tests for equality? That is, we use one equal sign to assign values to variables. We use two equal signs to test if two values are equal to each other. Substituting one for the other is a common coding mistake.



Code Magnets Solution

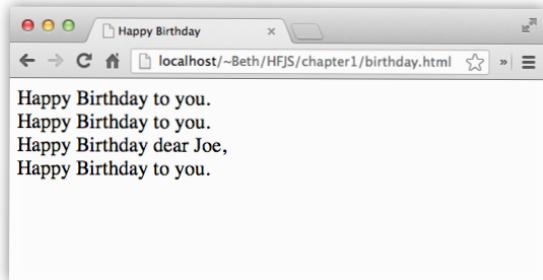
A JavaScript program is all scrambled up on the fridge. Can you put the magnets back in the right places to make a working JavaScript program to produce the output shown below?. Here's our solution.

Here are the unscrambled magnets!

```

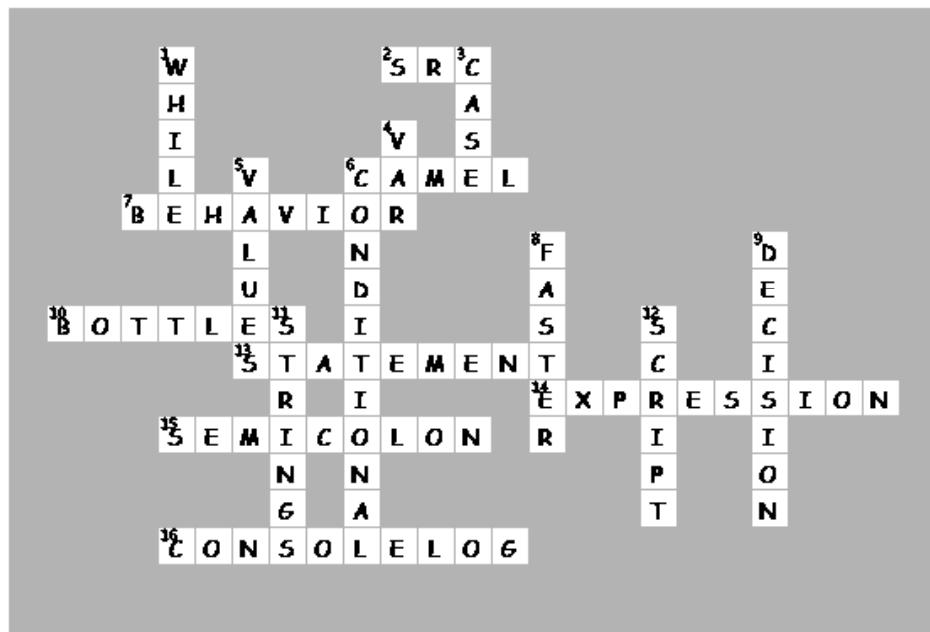
var name = "Joe";
var i = 0;
while (i < 2) {
    document.write("Happy Birthday to you.<br>");
    i = i + 1;
}
document.write("Happy Birthday dear " + name + ",<br>");
document.write("Happy Birthday to you.<br>");
```

Your unscrambled program
should produce this output.





JavaScript Cross Solution



WHO DOES WHAT? SOLUTION

All our methods of communication have come to the party with masks on. Can you help us unmask each one? Match the descriptions on the right to the names on the left. Here's our solution:

`document.write`

I'll stop your user in his tracks and deliver a short message. The user has to click "ok" to go further.

`console.log`

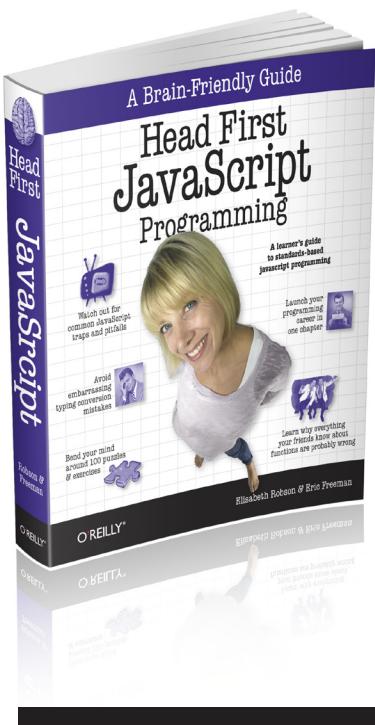
I can insert a little HTML and text into a document. I'm not the most elegant way to get a message to your users, but I work on every browser.

`alert`

Using me you can totally control a web page: get values that a user typed in, alter the HTML or the style, or update the content of your page.

`document object model`

I'm just here for simple debugging purposes. Use me and I can write out information to a special developer's console.



O'REILLY®

Head First JavaScript Programming

O'Reilly Media spreads the knowledge of innovators through its books, online services, magazines, research, and conferences. Get the information you need from the experts you trust; visit oreilly.com to purchase this book.

Get 40% off of the print book by entering discount code HFJSP.

Head First JavaScript Programming

By Eric T. Freeman, Elisabeth Robson

Mar 2014, ISBN 978-1-4493-4013-1

704 pages, \$49.99

Spreading the knowledge of innovators

oreilly.com