O'REILLY®

Head First JavaScript Programming

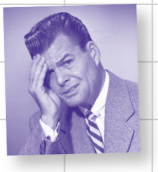
A Brain-Friendly Guide

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

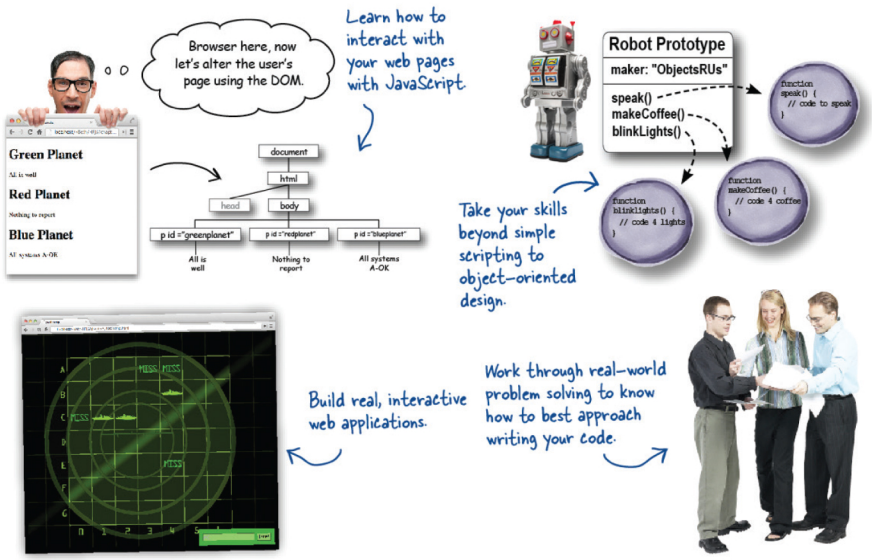
Eric Freeman & Elisabeth Robson

O'REILLY®

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

Eric T. Freeman
Elisabeth Robson

O'REILLY®

Beijing • Cambridge • Köln • Sebastopol • Tokyo

Head First JavaScript Programming

by Eric T. Freeman and Elisabeth Robson

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.

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

Elisabeth Robson

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 vistor 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 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)

Table of Contents (the real thing)

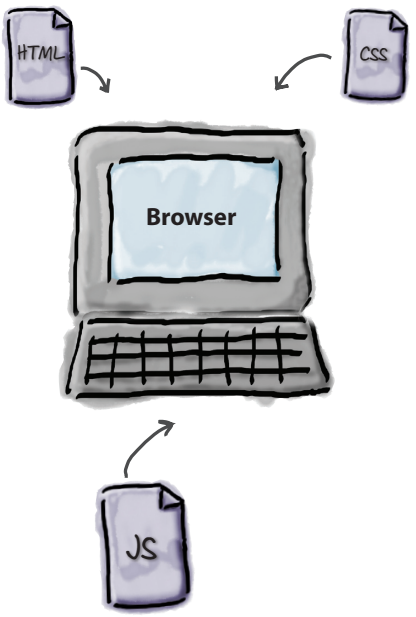
Intro

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?

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.

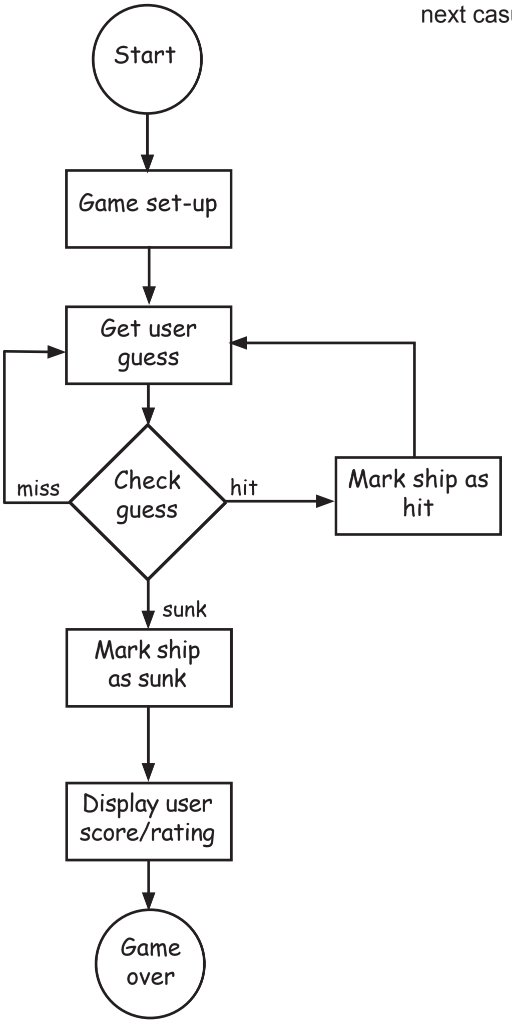


2

writing real code

Going further

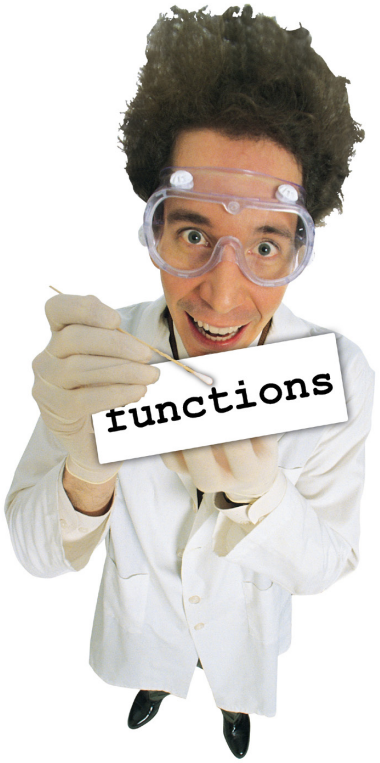
You already know about variables, types, expressions... we could go on. The point is, you already know a few things about JavaScript. In fact, you know enough to write some **real code**. Some code that does something interesting, some code that someone would want to use. What you're lacking is the **real 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.

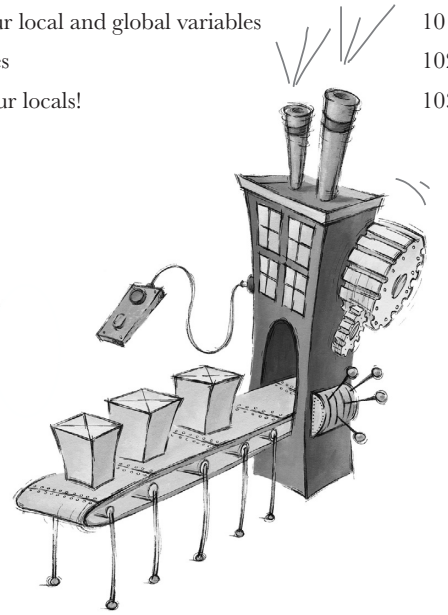


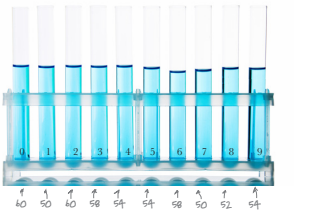
introducing functions

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*.







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.

undestanding objects

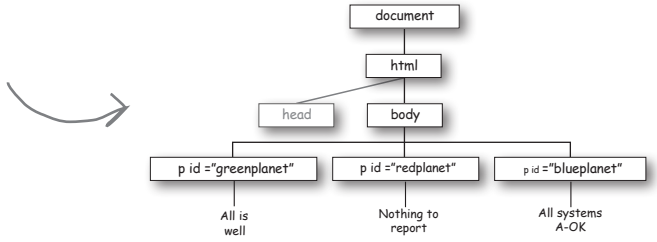
5 A trip to Objectville

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.

interacting with your web page

6 Getting to know the DOM

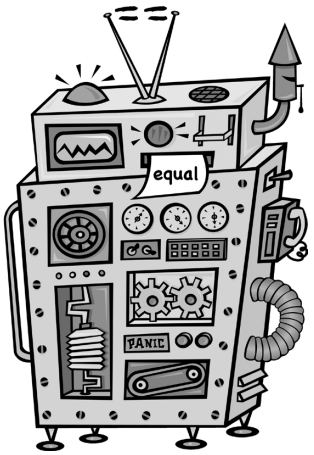
You've come a long way with JavaScript. In fact you've evolved from a newbie to a scripter to, well, a **programmer**. But, there's something missing. To really begin leveraging your JavaScript skills you need to know how to interact with the web page your code lives in. Only by doing that are you going 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.



types, equality, conversion, and all that jazz

7 Serious types

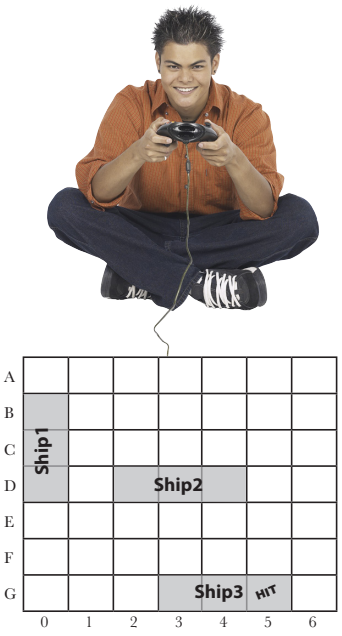
It's time to get serious about our types. One of the great things about JavaScript is you can get a long way without knowing a lot of details of the language. But to truly **master the language**, get that promotion and get on to the things you really want to do in life, you have to rock at **types**. 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.



bringing it all together

8 Building an app

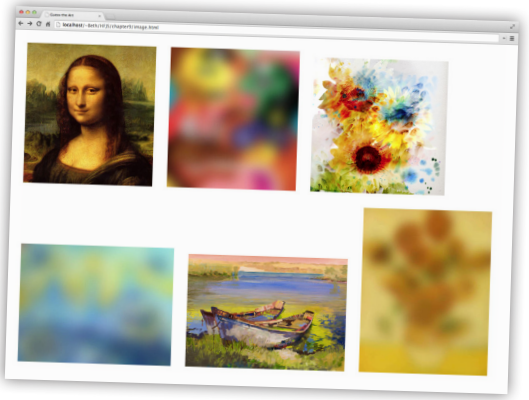
Put on your toolbelt. That is, the toolbelt with all your new coding skills, your knowledge of the DOM, and even some HTML & CSS. We're going to bring everything together in this chapter to create our first true **web application**. No more **silly toy games** with one battleship and a single row of 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.



asynchronous coding

9 Handling events

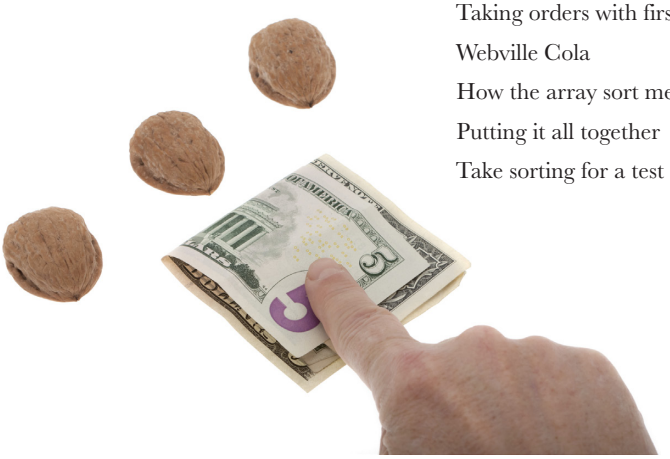
After this chapter you're going to realize you aren't in Kansas anymore. Up until now, you've been writing code that typically executes from top to bottom—sure, your code might be a little more complex than that, and make use of a few functions, objects and methods, but at some point the code just runs 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.



first class functions

10 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.



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 JavasScript 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.

advanced object construction

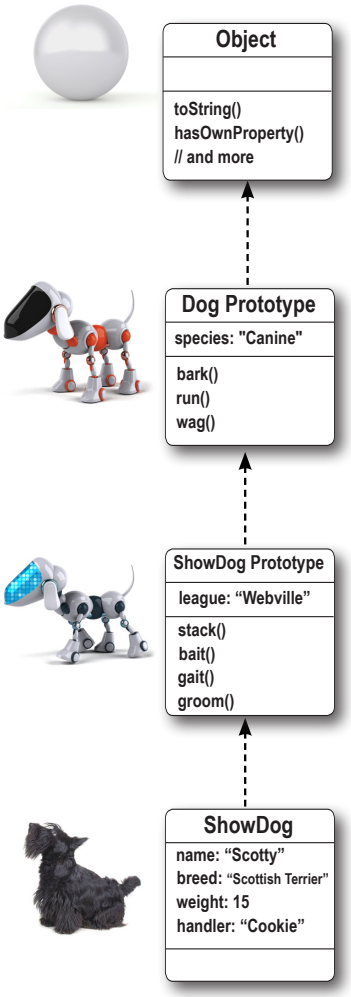
12 Creating objects

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.

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...



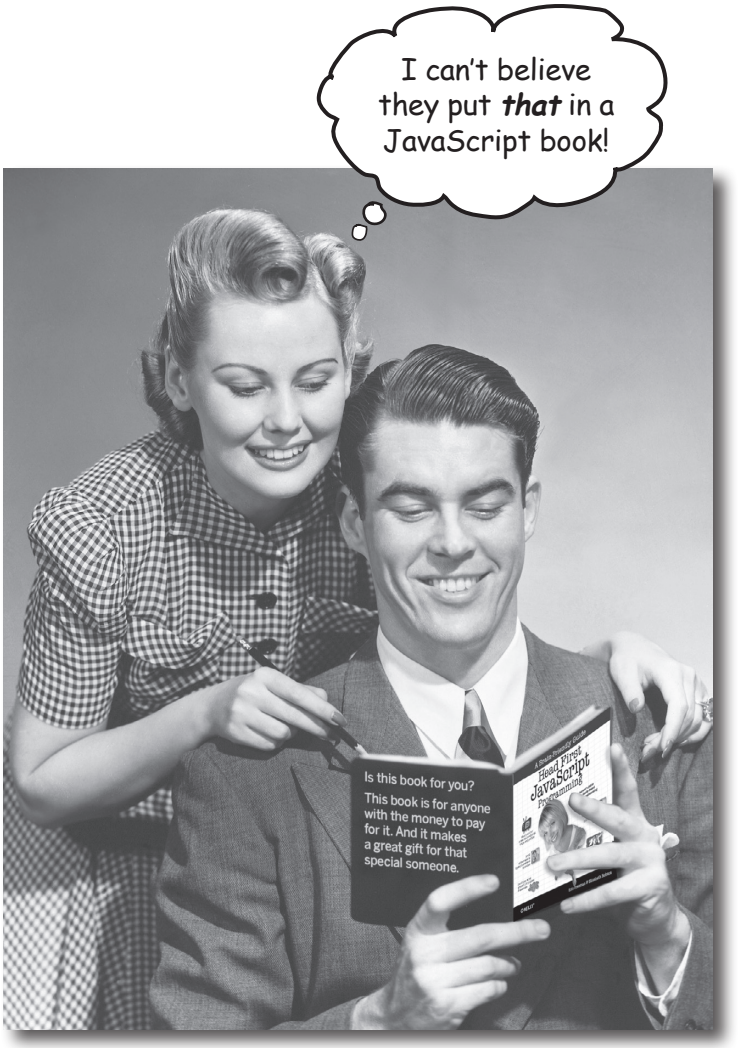
Appendix: Leftovers

14 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!).

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?**

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

this book is for you.

[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.

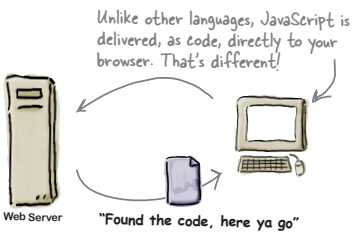
And there's no simple way to tell your brain, "Hey brain, thank you very much, but no matter how dull this book is, and how little I'm registering on the emotional Richter scale right now, I really *do* want you to keep this stuff around."

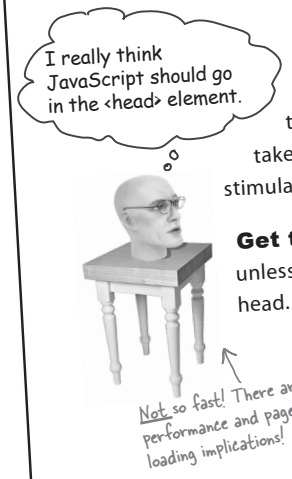
We think of a "Head First" reader as a learner.

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?

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?

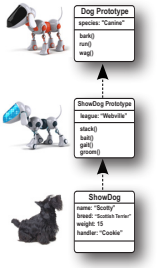
There's the slow, tedious way, or the faster, more effective way. The slow way is about sheer repetition. You obviously know that you *are* able to learn and remember even the dullest of topics, if you keep pounding on the same thing. With enough repetition, your brain says, "This doesn't *feel* important 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.***

Be the Browser

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.

BULLET POINTS

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.

Puzzles

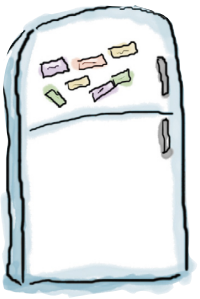
We include content for ***both sides of your brain***, because the more of your brain you engage, the more likely you are to learn and remember, and the longer you can stay focused. Since working one side of the brain often means giving the other side a chance to rest, you can be more productive at learning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



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

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

Jeff Straw

Ismaël Martin "Bing" Demiddel

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**. 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.

Esteemed Reviewer, David Powers

Don't let the smile fool you, this guy is hard core (technically of course).

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!

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

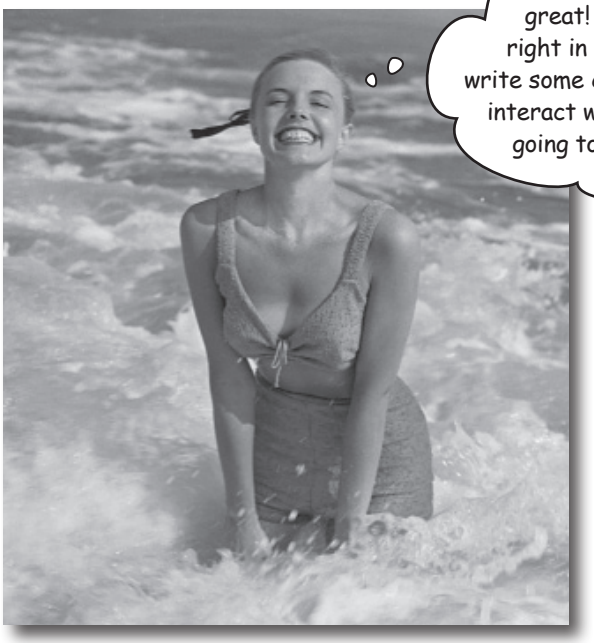
*The large number of acknowledgments is because we're testing the theory that everyone mentioned in a book acknowledgment will buy at least one copy, probably more, what with relatives and everything. If you'd like to be in the acknowledgment of our *next* book, and you have a large family, write to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

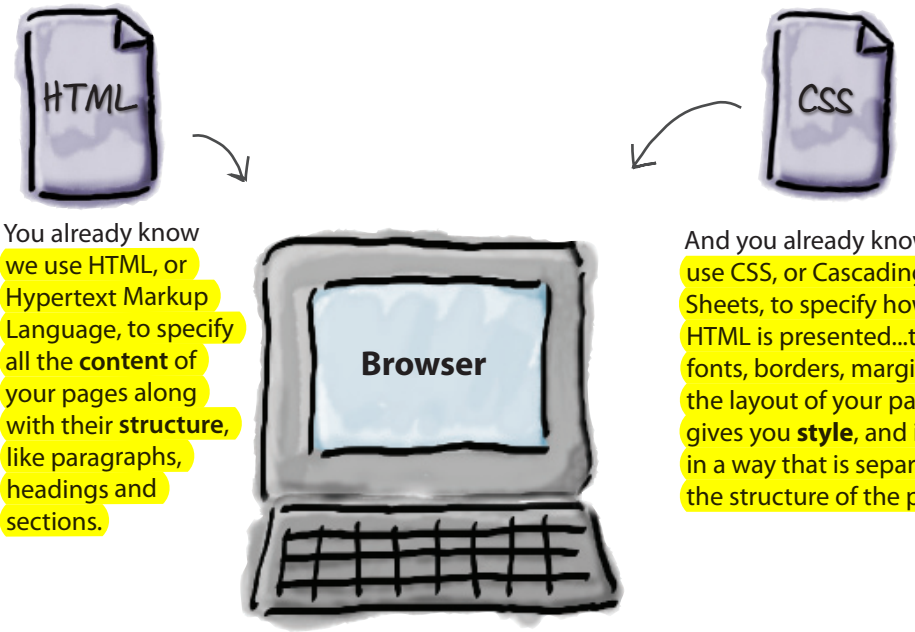


JavaScript gives you superpowers. The **true programming language** of the web, JavaScript lets you **add behavior** to your web pages. No more dry, boring, static pages that just sit there looking at you—with JavaScript you're going to be able to reach out and touch your users, react to interesting 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!

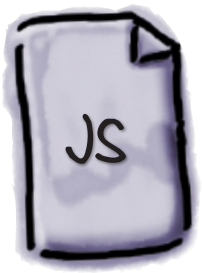
The way JavaScript works

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.

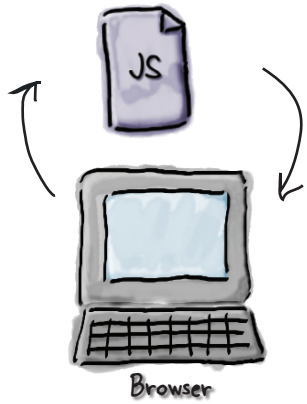


So let's introduce JavaScript, HTML & CSS's computational cousin. JavaScript lets you create **behavior** in your web pages. Need to react when a user clicks on your "On Sale for the next 30 seconds!" button? Double check your user's form input on the fly? Grab some tweets from Twitter and display them? 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:

```
<html>
<head>
<title>Icecream</title>
<script>
 var x = 49;
</script>
<body>
<h1>Icecream Flavors</h1>
<h2><em>49 flavors</em></h2>
<p>All your favorite
flavors!</p>
</body>
</html>
```



Writing

1

You create your page just like you always do, with HTML content and CSS style. And you also include JavaScript in your page. As you'll see, just like HTML and CSS, you can put everything together in one file, or you can place JavaScript in its own file, to be included in your page.

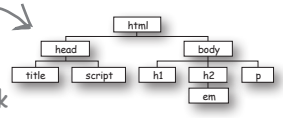
We'll talk about the best way in a bit...

Loading

2

Point your browser to your page, just as you always do. When the browser sees code, it starts parsing it immediately, getting ready to execute it. Note that like HTML and CSS, if the browser sees errors in your code, it will do its best to keep moving and reading more JavaScript, HTML and CSS. The last thing it wants to do is not be able to give the user a page to see.

For future reference, the browser also builds an "object model" of your HTML page that JavaScript can make use of. Put that in the back of your brain, we'll come back to it later...



Executing

3

The browser starts executing your code as soon as it encounters it in your page, and continues executing it for the lifetime of your page. Unlike early versions of JavaScript, today's JavaScript is a powerhouse, using advanced compilation techniques to execute your code at nearly the same speed as many native programming languages.

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.

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.

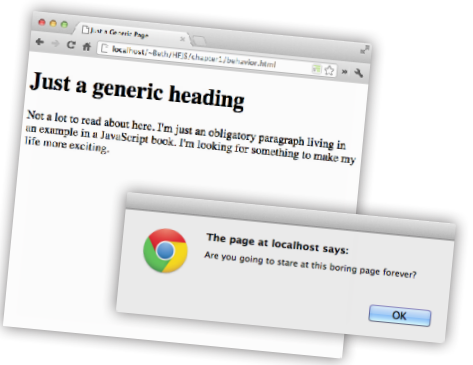
And we've written some JavaScript code inside it.

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.

```
<!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>
```

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:

```
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.

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

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.

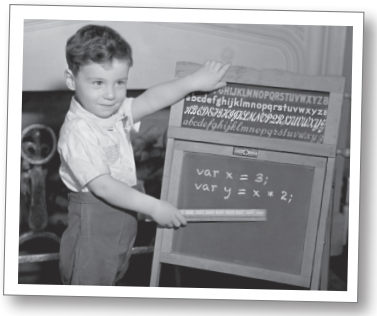
Q: My friend is using JavaScript inside Photoshop, or at least he says he is. Is that possible?

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?

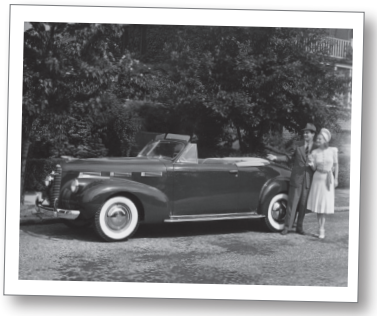
Did we mention Microsoft? They created their own scripting language soon after Netscape did, named, um, JScript, and it was, um, quite similar to JavaScript. And so began the browser wars.

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.

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

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.

`var price = 28.99;`	Create a variable named price, and assign the value 28.99 to it.
`var discount = 10;`	Create a variable named discount, and assign the value 10 to it.
`var total = price - (price * (discount / 100));`	Compute a new price by applying a discount and then assign it to the variable total.
`if (total > 25) {`	Compare the value in the variable total to 25. If it's greater...
`freeShipping();`	...then do something with freeShipping.
`}`	End the if statement
`var count = 10;`	Create a variable named count, and assign the value 10 to it.
`while (count > 0) {`	As long as the variable count is greater than 0...
`juggle();`	...do some juggling, and...
`count = count - 1;`	...reduce the value of count by 1 each time.
`}`	End the while loop
`var dog = {name: "Rover", weight: 35};`	Create a dog with a name and weight.
`if (dog.weight > 30) {`	If the dog's weight is greater than 30...
`alert("WOOF WOOF");`	...alert "WOOF WOOF" to the browser's web page
`} else {`	Otherwise...
`alert("woof woof");`	...alert "woof woof" to the browser's web page
`}`	End the if/else statement
`var circleRadius = 20;`	Create a variable, circleRadius, and assign the value 20 to it.
`var circleArea =`	Create a variable named circleArea...
`Math.PI * (circleRadius * circleRadius);`	...and assign the result of this expression to it (1256.637061435917 3)

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.

And usually increase the size of your paycheck too!

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…

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:

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.

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

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:

A set of statements.

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

```
var age = 25;
var name = "Owen";

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

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."

Notice we don't put quotes around boolean values.

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.

```
var winners = 2;
```

We always end an assignment statement with a semicolon.

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.

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 embarassing 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`.

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.

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);
}
```

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.

We use * for multiply and / for divide.

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

Here's our variable total.

And the assignment.

And this whole thing is an expression.

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" + ","
```

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

```
"super" + "cali" + youKnowTheRest
```

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

```
phoneNumber.substring(0,3)
```

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"?
_____

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.

```
yourLevel > 5
```

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? _____

```
(level * points) + bonus
```

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

```
color != "orange"
```

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

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? _____

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:

A while statement starts with the keyword while.

While uses a boolean expression that we call a conditional test, or conditional for short.

If the conditional is true, everything in the code block is executed.

```
while (scoops > 0) {
    document.write("Another scoop!");
    scoops = scoops - 1;
}
```

What's a code block? Everything between the curly braces; that is, between {}.

And, if our conditional is true, then, after we execute the code block, we loop back around and do it all again. If the conditional is false, we're done.

Like we said, lather, rinse, repeat!

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");
```

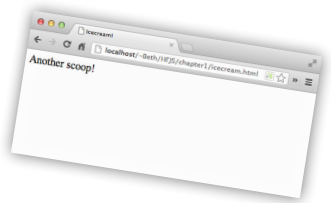
After that we hit the while statement. When we evaluate a while statement the first thing we do is evaluate the conditional to see if it's true or false.

```
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.

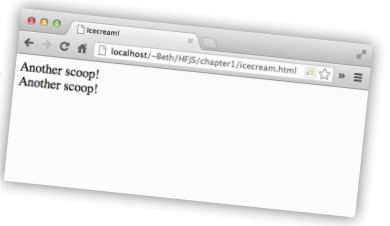
```
var scoops = 5;
while (scoops > 0) {
   document.write("Another scoop!<br>");
   scoops = scoops - 1;
}
document.write("Life without ice cream isn't the same");
```

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");
```

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");
```



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");
```

2 scoops gone, 3 left!

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");
```

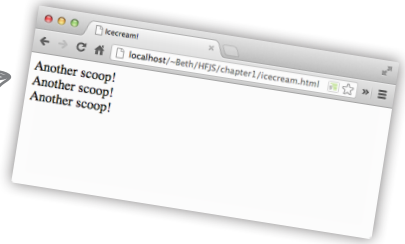
Evaluating our conditional again, this time scoops is three. But that's still more than zero.

```
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!

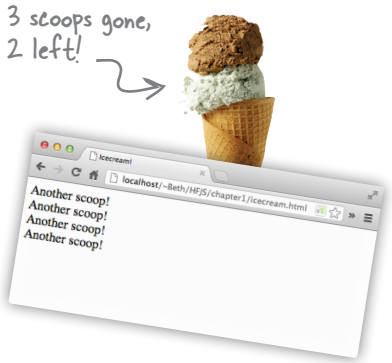
Once again we write the string "Another scoop!
" to the browser.

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



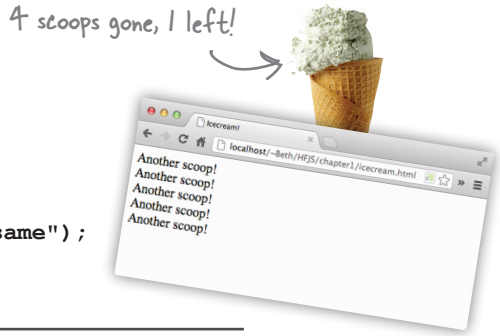
And as you can see, this continues... each time we loop, we decrement (reduce scoops by 1), write another string to the browser, and keep going.

```
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");
```

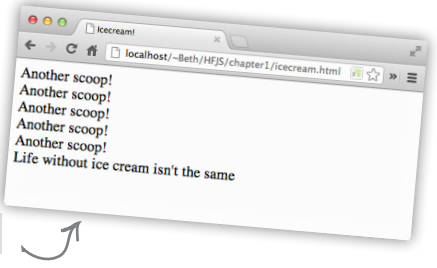


Until the last time... this time something's different. Scoops is zero, and so our conditional returns false. That's it folks; we're not going to go through the loop anymore, we're not going to execute the block. This time, we bypass the block and execute the statement that follows it.

```
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");
```



```
if (cashInWallet > 5) {
  order = "I'll take the works: cheeseburger, fries and a coke";
} else {
  order = "I'll just have a glass of water";
}
```

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:

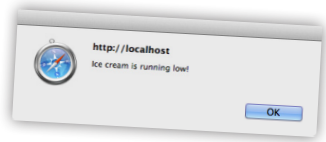
Here's the if keyword, followed by a conditional and a block of code.

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

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

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!



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

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

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

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) {
   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.

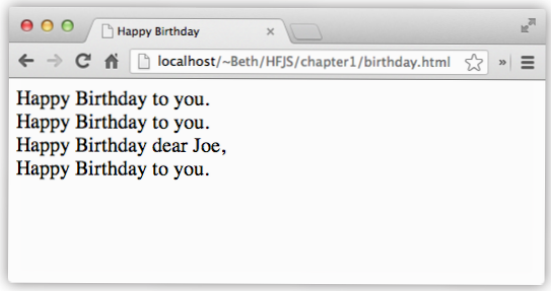
Code Magnets

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.



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.

We're using these three methods in this chapter.

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.

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.

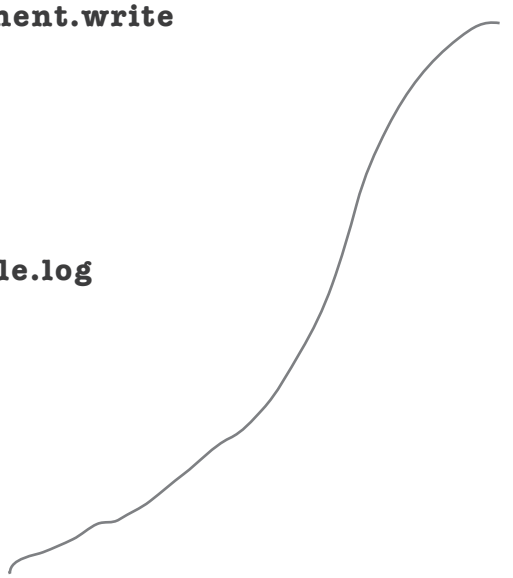
Directly manipulate your document.

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.

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

console.log

alert

document object model

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

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.

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.

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

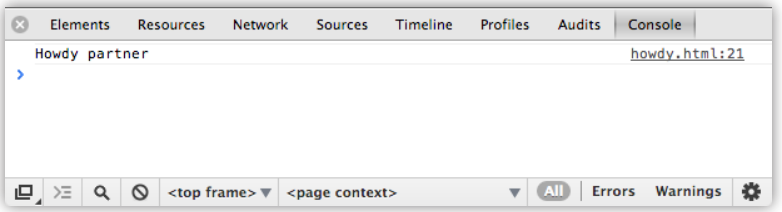
A closer look at console.log

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:

Take any old string...

```
var message = "Howdy" + " " + "partner";
console.log(message);
```

...and give it to console.log, and it will be shown in the browser's console, like this.



The console contains all the output logged by your code.

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" seperated 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.

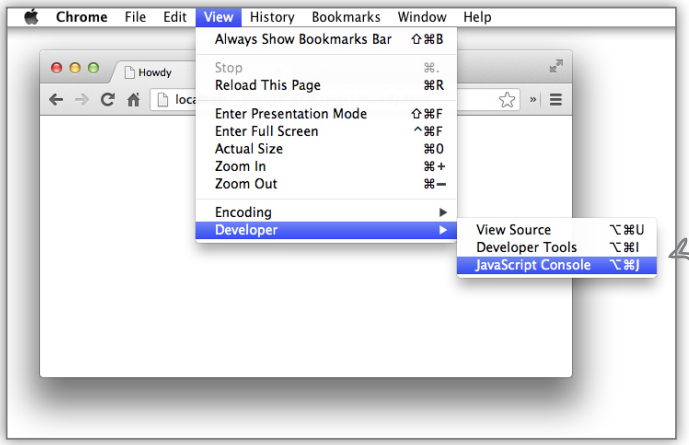
Q: Uh, console looks great, but where do I find it? I'm using it in my code and I don't see any output!

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

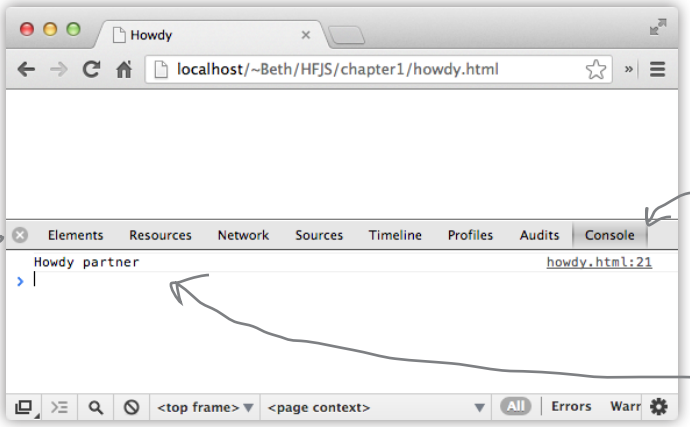
Opening the console

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/hfjsconsole. 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 `else`s. 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.

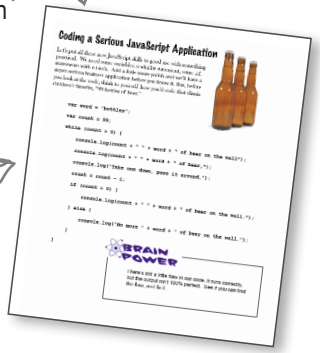
Let's first get this serious business application into a page, and then after we've thoroughly tested it, we'll move the JavaScript out to an external file.

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.

1. 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>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>My First JavaScript</title>
  </head>
  <body>
    <script>


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

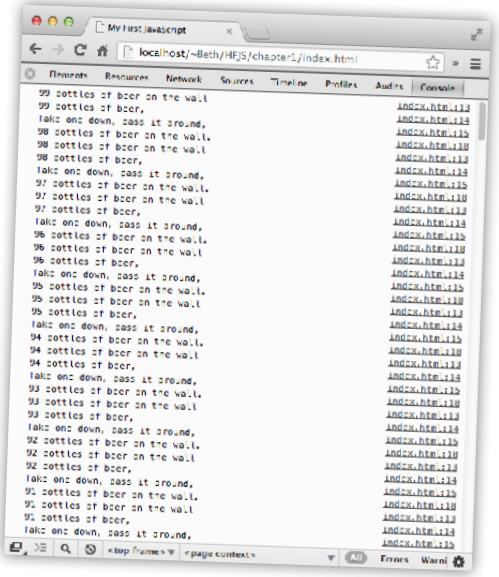
Type this in.

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

2. Save the file as "index.html".

3. 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.

4. 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)

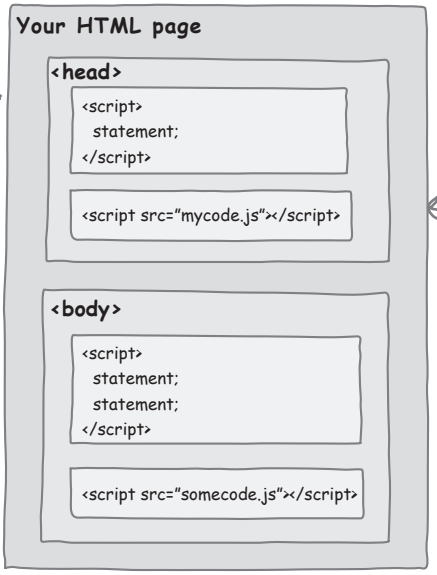
You already know you can add the `<script>` element with your JavaScript code to the `<head>` or `<body>` of your page, but there are a couple of other ways to add your code to a page. Let's check out all the places you can put JavaScript (and why you might want to put it one place over another):

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.

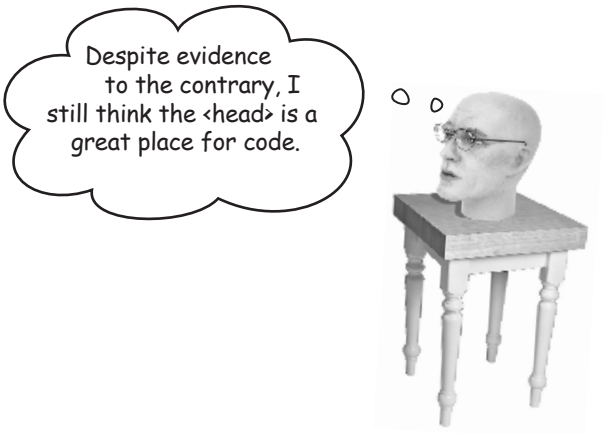
Still, is there a better way? Read on...



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.



We're going to have to separate you two

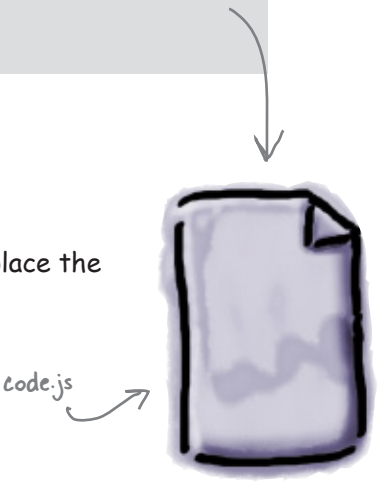
Going separate ways hurts, but we know we have to do it. It's time to take your JavaScript and move it into its own file. Here's how you do that…

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".

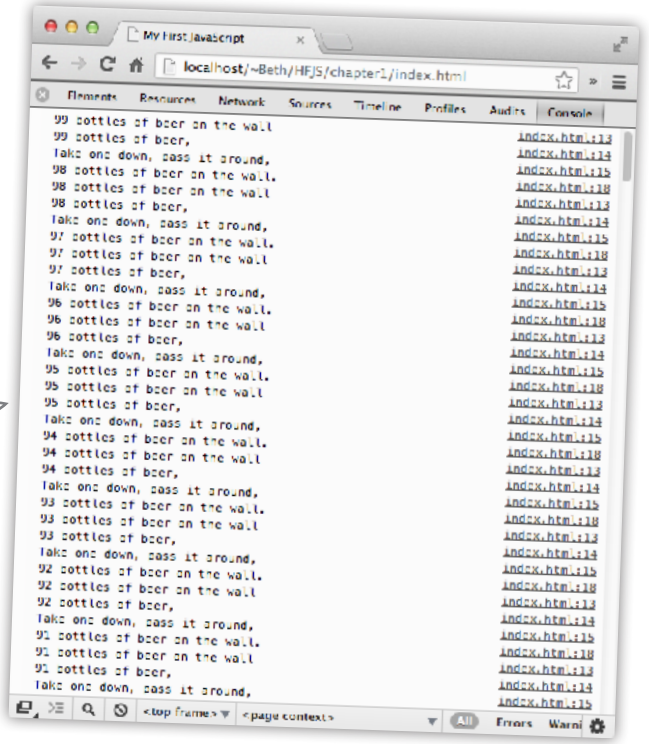
```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>My First JavaScript</title>
  </head>
  <body>
    <script src="code.js">

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

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

Where your code was.

Believe it or not we still need the ending <script> tag, even if there is no code between the two tags.

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 a 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.

The <script> opening tag.

Don't forget the right bracket on the opening tag.

```
<script type="text/javascript">
  alert("Hello world!");
</script>
```

Everything between the script tags must be valid JavaScript.

You must end the script with a closing </script> tag, always!

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

Add a src attribute to specify the URL of the JavaScript file.

```
<script src="myJavaScript.js">
</script>
```

Use ".js" as the extension on JavaScript files.

When referencing a separate JavaScript file, you don't put any JavaScript in the content of the <script> element.

Again, don't forget the closing </script> tag! You need it even when you're linking to an external file.

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?

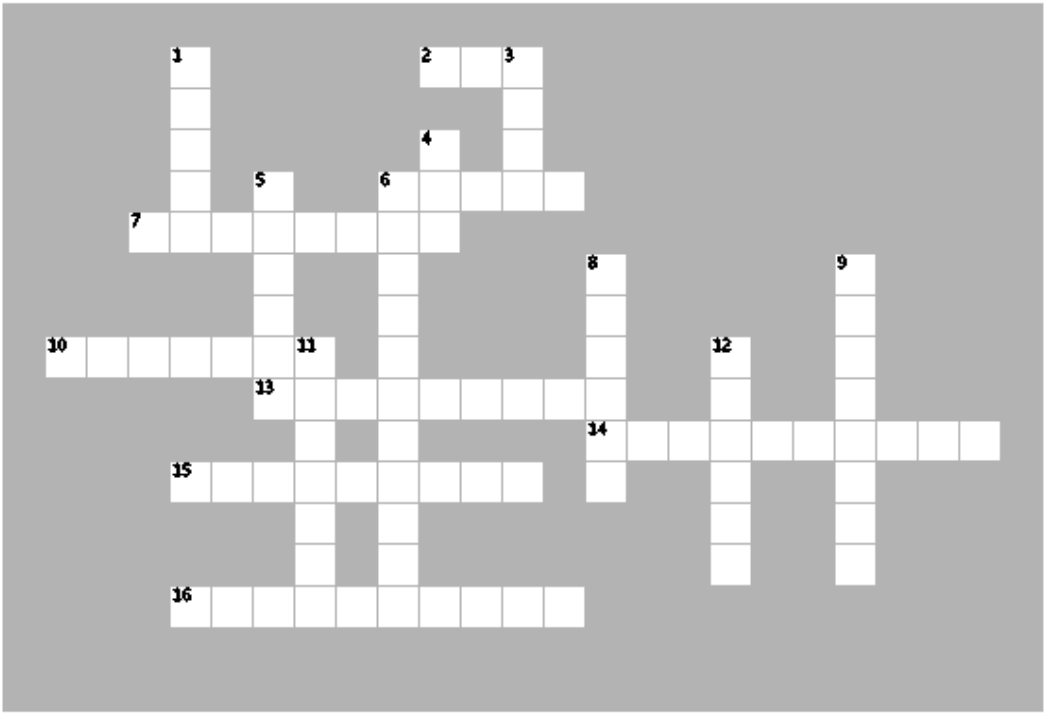
JavaScript: Don't judge me by my bad parts, learn the good stuff and stick with that!

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

1. 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 =
  "Better watch out for those semi-colons."
var %entage = 20;
var result

if (toldJoke == true) {
    Alert($punchline);
} else
    alert(joke);
}
```

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!

Can't use % in variable names.

Another missing semi-colon.

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

We're missing an opening brace here.

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);
}
```

Comments should begin with // not \\.

No spaces allowed in variable names.

No quotes allowed in variable names.

But we do need quotes around the string "Forbidden Planet".

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

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
```

What is the result when temp is 10? 50

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"? false
Or, has the value "orange"? true

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

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

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.

```
yourLevel > 5
```

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

```
(level * points) + bonus
```

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

```
color != "orange"
```

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

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"

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.

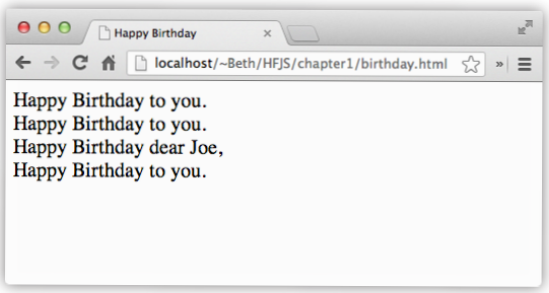
Code Magnets Solution

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.

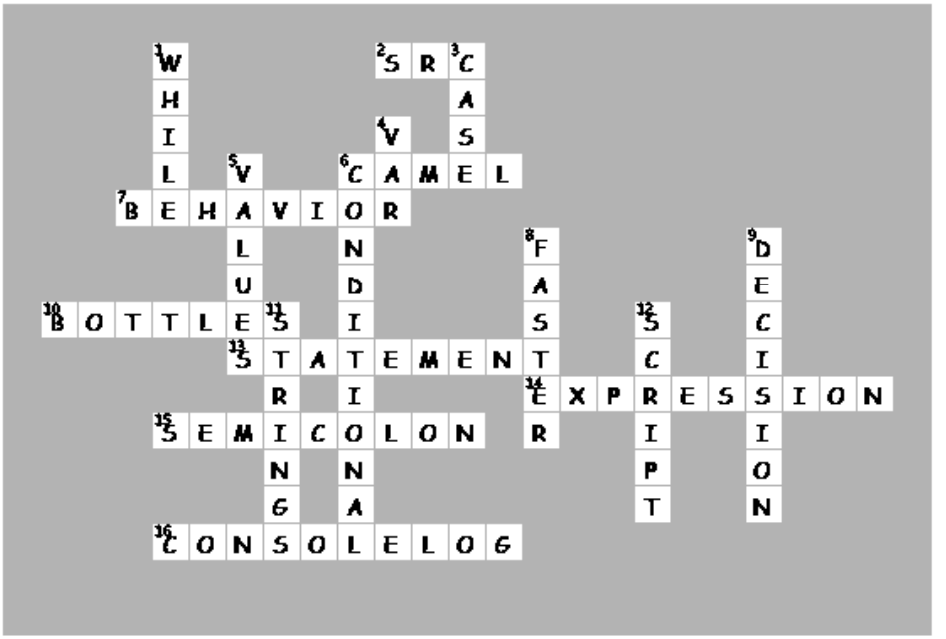
Here are the unscrambled magnets!

```
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

console.log

alert

document object model

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

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.

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.

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

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.

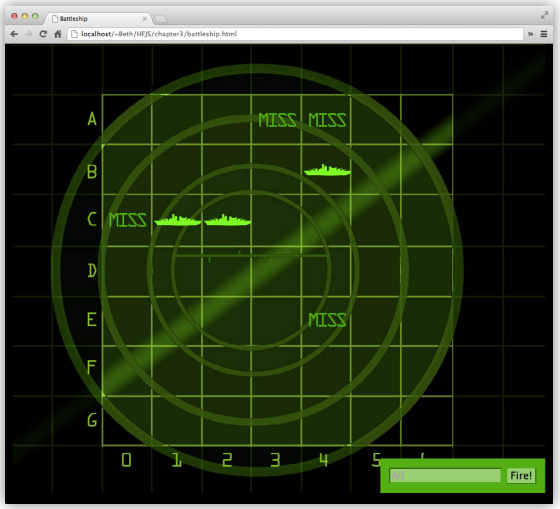
Let's build a Battleship game

It's you against the browser: the browser hides ships and your job is to seek them out and destroy them. Of course, unlike the real Battleship game, in this one you don't place any ships of your own. Instead, your job is to sink the computer's ships in the fewest number of guesses.

Goal: Sink the browser's ships in the fewest number of guesses. You're given a rating, based on how well you perform.

Setup: When the game program is launched, the computer places ships on a virtual grid. When that's done, the game asks for your first guess.

How you play: The browser will prompt you to enter a guess and you'll type in a grid location. In response to your guess, you'll see a result of "Hit", "Miss", or "You sank my battleship!" When you sink all the ships, the game ends by displaying your rating.



Here's what we're shooting for: a nice 7x7 grid with three ships to hunt down. Right now we're going to start a little simpler, but once you know a bit more JavaScript we'll complete the implementation so it looks just like this, complete with graphics and everything...we'll leave the sound to you as extra credit.

Our first attempt...

... a simplified Battleship

For our first attempt we're going to start simpler than the full-blown 7x7 graphical version with three ships. Instead we're going to start with a nice 1-D grid with seven locations and one ship to find. It will be crude, but our focus is on designing the basic code for the game, not the look and feel (at least for now).

Don't worry; by starting with a simplified version of the game, you get a big head start on building the full game later. This also gives us a nice chunk to bite off for your first real JavaScript program (not counting the Serious Business Application from Chapter 1, of course). So, we'll build the simple version of the game in this chapter, and get to the deluxe version later in the book after you've learned a bit more about JavaScript.

Instead of a 7x7 grid, like the one above, we're going to start with just a 1x7 grid. And, we'll worry about just one ship for now.

Notice that each ship takes up three grid locations (similar to the real board game).

First, a high-level design

We know we'll need variables, and some numbers and strings, and if statements, and conditional tests, and loops... but where and how many? And how do we put it all together? To answer these questions, we need more information about what the game should do.

First, we need to figure out the general flow of the game. Here's the basic idea:

1 User starts the game

- **A** Game places a battleship at a random location on the grid.

2 Game play begins

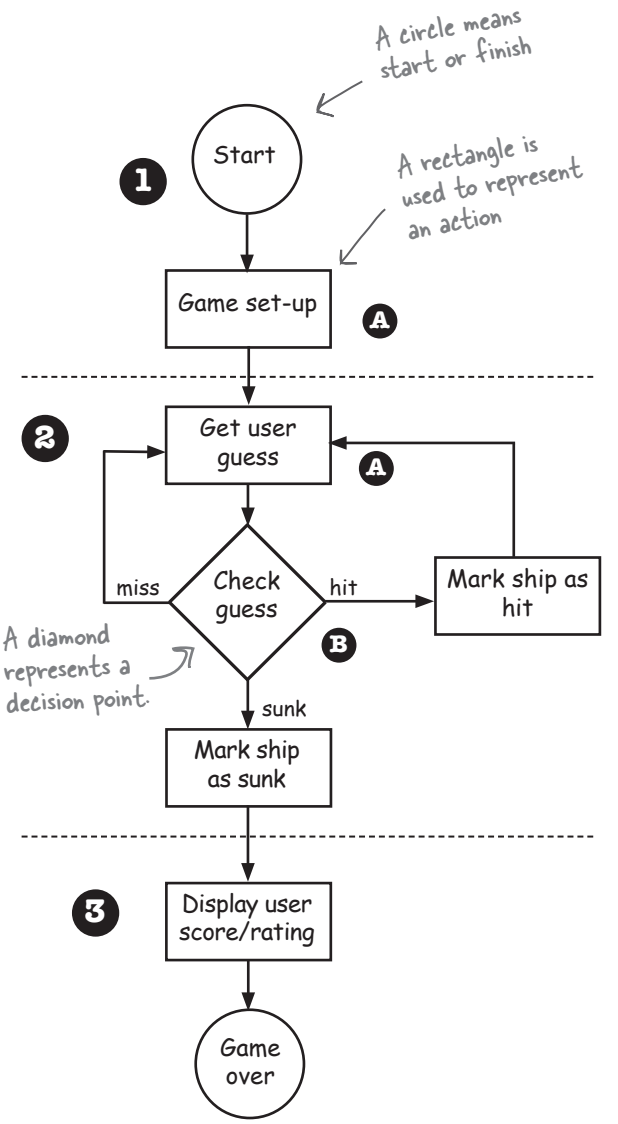
Repeat the following until the battleship is sunk:

- **A** Prompt user for a guess ("2", "0", etc.)
- **B** Check the user's guess against the battleship to look for a hit, miss or sink.

3 Game finishes

Give the user a rating based on the number of guesses.

Now we have a high-level idea of the kinds of things the program needs to do. Next we'll figure out a few more details for the steps.



Whoa. A real flowchart.

A few more details...

We have a pretty good idea about how this game is going to work from the high-level design and professional looking flowchart, but let's nail down just a few more of the details before we begin writing the code.

Representing the ships

For one thing, we can start by figuring out how to represent a ship in our grid. Keep in mind that the virtual grid is… well, *virtual*. In other words, it doesn't exist anywhere in the program. As long as both the game and the user know that the battleship is hidden in three consecutive cells out of a possible seven (starting at zero), the row itself doesn't have to be represented in code. You might be tempted to build something that holds all seven locations and then to try to place the ship in those locations. But, we don't need to. We just need to know the cells where the ship is located, say, at cells 1, 2 and 3.

Getting user input

What about getting user input? We can do that with the `prompt` function. Whenever we need to get a new location from the user, we'll use `prompt` to display a message and get the input, which is just a number between 0 and 6, from the user.

Displaying the results

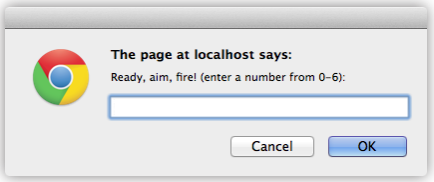
What about output? For now, we'll continue to use `alert` to show the output of the game. It's a bit clunky, but it'll work. (For the real game, later in the book, we'll be updating the web page instead, but we've got a way to go before we get there.)

1 **Game starts**, and creates one battleship and gives it a location on three cells in the single row of seven cells.

The locations are just integers; for example, 1,2,3 are the cell locations in this picture:

2 **Game play begins**. Prompt user for a guess:

A



B Check to see if user's input hit any of the ship's three cells. Keep track of how many hits there are in a variable.

3 **Game finishes** when all three cells have been hit and our number of hits variable value is 3. We tell the user how many guesses it took to sink the ship.

Sample game interaction

Working through the Pseudocode

We need an approach to planning and writing our code. We're going to start by writing *pseudocode*. Pseudocode is halfway between real JavaScript code and a plain English description of the program, and as you'll see, it will help us think through how the program is going to work without fully having to develop the *real code*.

In this pseudocode for Simple Battleship, we've included a section that describes the variables we'll need, and a section describing the logic of the program. The variables will tell us what we need to keep track of in our code, and the logic describes what the code has to faithfully implement to create the game.

The variables we need.

```
DECLARE three variables to hold the location of each cell of the ship. Let's call them
location1, location2 and location3.
DECLARE a variable to hold the user's current guess. Let's call it guess.
DECLARE a variable to hold the number of hits. We'll call it hits and set it to 0.
DECLARE a variable to hold the number of guesses. We'll call it guesses and set it to 0.
DECLARE a variable to keep track of whether the ship is sunk or not. Let's call it isSunk
and set it to false.
```

And here's the logic.

```
LOOP: while the ship is not sunk
    GET the user's guess
    COMPARE the user's input to valid input values
    IF the user's guess is invalid,
        TELL user to enter a valid number
    ELSE
        ADD one to guesses
        IF the user's guess matches a location
            ADD one to the number of hits
            IF number of hits is 3
                SET isSunk to true
                TELL user "You sank my battleship!"
            END IF
        END IF
    END ELSE
END LOOP
TELL user stats
```

It's not JavaScript, but you can probably already see how to begin implementing this logic in code.

Notice that we're using indentation to help make the pseudocode easier to read. We'll be doing that in the real code too.

Sharpen your pencil

Let's say our virtual grid looks like this:

And we've represented the ship locations using our location variables, like this:

```
location1 = 3;
location2 = 4;
location3 = 5;
```

Use the following sequence as your test user input:

```
1, 4, 2, 3, 5
```

Now, using the pseudocode on the previous page, walk through each step of code and see how this works given the user input. Put your notes below. We've begun the exercise for you below. If this is your first time walking through pseudocode, take your time and see how it all works.

If you need a hint, take a quick peek at our answer at the end of the chapter.

location1	location2	location3	guess	guesses	hits	isSunk
3	4	5	—	0	0	false
3	4	5	1	1	0	false

The first row shows the initial values of the variables, before the user enters their first guess. We're not initializing the variable guess, so its value is undefined.

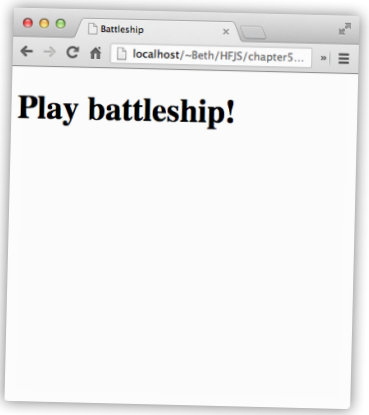
Oh, before we go any further, don't forget the HTML!

You're not going to get very far without some HTML to link to your code. Go ahead and type the markup below into a new file named "battleship.html". After you've done that we'll get back to writing code.

The HTML for the Battleship game is super simple; we just need a page that links to the JavaScript code, and that's where all the action happens.

```
<!doctype html>
<html lang="en">
  <head>
    <title>Battleship</title>
    <meta charset="utf-8">
  </head>
  <body>
    <h1>Play battleship!</h1>
    <script src="battleship.js"></script>
  </body>
</html>
```

We're linking to the JavaScript at the bottom of the <body> of the page, so the page is loaded by the time the browser starts executing the code in "battleship.js".



Here's what you'll see when you load the page. We need to write some code to get the game going!

BRAIN POWER

Flex those dendrites.

This is thinking ahead a bit, but what kind of code do you think it would take to generate a random location for the ship each time you load the page? What factors would you have to take into account in the code to correctly place a ship? Feel free to scribble some ideas here.

Writing the Simple Battleship code

We're going to use the pseudocode as a blueprint for our real JavaScript code. First, let's tackle all the variables we need. Take another look at our pseudocode to check out the variables we need:

We need three variables to hold the ship's location.

DECLARE three *variables* to hold the location of each cell of the ship. Let's call them `location1`, `location2` and `location3`.

DECLARE a *variable* to hold the user's current guess. Let's call it `guess`.

DECLARE a *variable* to hold the number of hits. We'll call it `hits` and *set* it to 0.

DECLARE a *variable* to hold the number of guesses. We'll call it `guesses` and *set* it to 0.

And three more (guess, hits and guesses) to deal with the user's guess.

DECLARE a *variable* to keep track of whether the ship is sunk or not. Let's call it `isSunk` and *set* it to `false`.

And another to track whether or not the ship is sunk.

Let's get these variables into a JavaScript file. Create a new file named "battleship.js" and type in your variable declarations like this:

```
var location1 = 3;
var location2 = 4;
var location3 = 5;

var guess;
var hits = 0;
var guesses = 0;

var isSunk = false;
```

Here are our three location variables. We'll go ahead and set up a ship at locations 3, 4 and 5, just for now.

We'll come back later and write some code to generate a random location for the ship to make it harder for the user.

The variable guess won't have a value until the user makes a guess. Until then it will have the value undefined.

We'll assign initial values of 0 to both hits and guesses.

Finally, the isSunk variable gets a value of false. We'll set this to true when we've sunk the ship.

Serious Coding

If you don't provide an initial value for a variable, then JavaScript gives it a default value of `undefined`. Think of the value `undefined` as JavaScript's way of saying "this variable hasn't been given a value yet." We'll be talking more about undefined and some other strange values a little later.

Now let's write the game logic

We've got the variables out of the way, so let's dig into the actual pseudocode that implements the game. We'll break this into a few pieces. The first thing you're going to want to do is implement the loop: it needs to keep looping while the ship isn't sunk. From there we'll take care of getting the guess from the user and validating it—you know, making sure it really is a number between 0 and 6—and then we'll write the logic to check for a hit on a ship and to see if the ship is sunk. Last, we'll create a little report for the user with the number of guesses it took to sink the ship.

```
LOOP: while the ship is not sunk
    GET the user's guess
    COMPARE the user's input to valid input values
  IF the user's guess is invalid,
      TELL user to enter a valid number
  ELSE
      ADD one to guesses
    IF the user's guess matches a location
        ADD one to the number of hits
      IF number of hits is 3
          SET isSunk to true
          TELL user "You sank my battleship!"
      END IF
    END IF
  END ELSE
END LOOP
TELL user stats
```

STEP1: Set up the loop, get the user's input and validate it.

STEP2: Check the guess. Is it a hit? A miss?

STEP3: Check to see if the ship is sunk.

STEP4: Handle the final message to the user.

To Do:

- [ ] Create loop and get user guess
- [ ] Check user guess
- [ ] Check if ship has been sunk
- [ ] Display stats to user

Step One: setting up the loop, getting some input

- [ ] Create loop and get user guess
- [ ] Check user guess
- [ ] Check if ship has been sunk
- [ ] Display stats to user

Now we're going to begin to translate the logic of our game into actual JavaScript code. There isn't a perfect mapping from pseudocode to JavaScript, so you'll see a few adjustments here and there. The pseudocode gives us a good idea of *what* the code needs to do, and now we have to write the JavaScript code that can do the *how*.

Let's start with all the code we have so far and then we'll zero in on just the parts we're adding (to save a few trees here and there, or electrons if you're reading the digital version):

DECLARE variables

```
var location1 = 3;
var location2 = 4;
var location3 = 5;
var guess;
var hits = 0;
var guesses = 0;
var isSunk = false;
```

We've already covered these, but we're including them here for completeness.

Here's the start of the loop. While the ship isn't sunk, we're still in the game, so keep looping.

Remember, while uses a conditional test to determine whether to keep looping. In this case we're testing to make sure that isSunk is still false. We'll set it to true as soon as the ship is sunk.

LOOP: *while the ship is not sunk*

```
while (isSunk == false) {
```

GET the user's guess

```
    guess = prompt("Ready, aim, fire! (enter a number 0-6):");

}
```

Each time we go through the while loop we're going to ask the user for a guess. To do that we use the prompt built-in function. More on that on the next page...

How prompt works

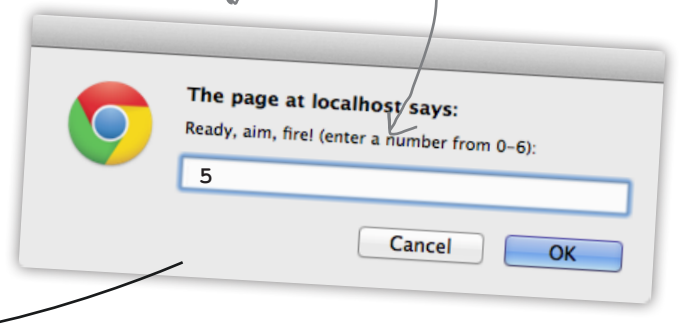
The browser provides a built-in function you can use to get input from the user, named `prompt`. The `prompt` function is a lot like the `alert` function you've already used—`prompt` causes a dialog to be displayed with a string that you provide, just like `alert`—but it also provides the user with a place to type a response. That response, in the form of a string, is then returned as a result of calling the function. Now, if the user cancels the dialog or doesn't enter anything , then `null` is returned instead.

Here we're assigning the result of the prompt function to the guess variable.

```
guess = prompt("Ready, aim, fire! (enter a number 0-6):");
```

You provide prompt with a string, which is used as instructions to your user in the dialog box.

The prompt function's job is to get input from the user. Depending on your device, that usually happens in a dialog box.



"5"

Once the prompt function obtains input from the user, it returns that input to your code. In this case the input, in the form of a string, is assigned to the variable guess.

Watch it!

You might be tempted to try this code now...

...but don't. If you do, your browser will start an *infinite loop* of asking you for a guess, and then asking you for a guess, and so on, without any means of stopping the loop (other than using your operating system to force the browser process to stop).

Checking the user's guess

- [x] Create loop and get user guess
- [ ] Check user guess
- [ ] Check if ship has been sunk
- [ ] Display stats to user

If you look at the pseudocode, to check the user's guess we need to first make sure the user has entered a valid input. If so, then we also check to see if the guess was a hit or miss. We'll also want to make sure we appropriately update the `guesses` and `hits` variables. Let's get started by checking the validity of the user's input, and if the input is valid, we'll increment the `guesses` variable. After that we'll write the code to see if the user has a hit or miss.

```
// Variable declarations go here


while (isSunk == false) {
    guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
    if (guess < 0 || guess > 6) {
       alert("Please enter a valid cell number!");
    } else {
       guesses = guesses + 1;
    }
}
```

We check validity by making sure the guess is between zero and six.

If the guess isn't valid, we'll tell the user with an alert.

And if the guess is valid, go ahead and add one to guesses so we can keep track of how many times the user has guessed.

Let's look a little more closely at the validity test. You know we're checking to see that the guess is between zero and six, but how exactly does this conditional test that? Let's break it down:

Try to read this like it's English: this conditional is true if the user's guess is less than zero OR the user's guess is greater than six. If either is true, then the input is invalid.

```
if (guess < 0 || guess > 6) {
```

This is really just two small tests put together. The first test checks if guess is less than zero.

And this one checks to see if guess is greater than six.

And this, which we call the OR operator, combines the two tests so that if either test is true, then the entire conditional is true. If both tests are false, then the statement is false, and the guess is between 0 and 6, which means it's valid.

there are no Dumb Questions

Q: **I noticed there is a cancel button on the prompt dialog box. What gets returned from the prompt function if the user hits cancel?**

A: If you click cancel in the prompt dialog box then prompt returns the value null rather than a string. Remember that null means "no value", which is appropriate in this case because you've cancelled without entering a value. We can use the fact that the value returned from prompt is null to check to see if the user clicked cancel, and if they did, then we could, say, end the game. We're not doing that in our code, but keep this idea in the back of your mind as we might use it later in the book.

Q: **You said that prompt always returns a string. So how can we compare a string value, like "0" or "6", to numbers, like 0 and 6?**

A: In this situation, JavaScript tries to convert the string in guess to a number in order to do the comparisons, guess < 0 and guess > 6. As long as you enter only a number, like 4, JavaScript knows how to convert the string "4" to the number 4 when it needs to. We'll come back to the topic of type conversion in more detail later.

Q: **What happens if the user enters something that isn't a number into the prompt? Like "six" or "quit"?**

A: In that case, JavaScript won't be able to convert the string to a number for the comparison. So, you'd be comparing "six" to 6 or "quit" to 6, and that kind of comparison will return false, which will lead to a MISS. In a more robust version of battleship, we'll check the user input more carefully and make sure they've entered a number first.

Q: **With the OR operator, is it true if only one or the other is true, or can both be true?**

A: Yes, both can be true. The result of the OR operator (||) is true if either of the tests is true, or if both are true. If both are false, then the result is false.

Q: **Is there an AND operator?**

A: Yes! The AND operator (&&) works similarly to OR, except that the result of AND is true only if both tests are true.

Q: **What's an infinite loop?**

A: Great question. An infinite loop is one of the many problems that plague programmers. Remember that a loop requires a conditional test, and the loop will continue as long as that conditional test is true. If your code never does anything to change things so that the conditional test is false at some point, the loop will continue forever. And ever. Until you kill your browser or reboot.

Two-minute Guide to Boolean Operators

A boolean operator is used in a boolean expression, which results in a true or false value. There are two kinds of boolean operators: comparison operators and logical operators.

Comparison Operators

Comparison operators compare two values. Here are some common comparison operators:

- `<` means "less than"
- `>` means "greater than"
- `==` means "equal to"
- `===` means "exactly equal to" (we'll come back to this one later!)
- `<=` means "less than or equal to"
- `>=` means "greater than or equal to"
- `!=` means "not equal to"

Logical Operators

Logical operators combine two boolean expressions to create one boolean result (true or false). Here are two logical operators:

- `||` means OR. Results in true if *either* of the two expressions is true.
- `&&` means AND. Results in true if *both* of the two expressions are true.

Another logical operator is NOT, which acts on one boolean expression (rather than two):

- `!` means NOT. Results in true if the expression is false.

So, do we have a hit?

- [x] Create loop and get user guess
- [ ] Check user guess
- [ ] Check if ship has been sunk
- [ ] Display stats to user

This is where things get interesting—the user's taken a guess at the ship's location and we need to write the code to determine if that guess has hit the ship. More specifically, we need to see if the guess matches one of the locations of the ship. If it does, then we'll increment the `hits` variable.

Here's a first stab at writing the code for the hit detection; let's step through it:

```
if (guess == location1) {
     hits = hits + 1;
} else  if (guess == location2) {
     hits = hits + 1;
} else if (guess == location3) {
     hits = hits + 1;
}
```

If the guess is at location1, then we hit the ship, so increment the hits variable by one.

Otherwise, if the guess is location2, then do the same thing.

Finally, if the guess is location3, then we need to increment the hits variable.

And if none of these are true, then the hits variable is never incremented.

Notice we're using indentation for the code in each if/else block. This makes your code easier to read, especially when you've got lots of blocks nested inside blocks.

Sharpen your pencil

What do you think of this first attempt to write the code to detect when a ship is hit? Does it look more complex than it needs to be? Are we repeating code in a way that seems a bit, well, redundant? Could we simplify it? Using what you know of the || operator (that is, the boolean OR operator), can you simplify this code? *Make sure you check your answer at the end of the chapter before moving on.*

Adding the hit detection code

- [x] Create loop and get user guess
- [x] Check user guess
- [ ] Check if ship has been sunk
- [ ] Display stats to user

Let's put everything together from the previous couple of pages:

```
// Variable declarations go here

while (isSunk == false) {
    guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
    if (guess < 0 || guess > 6) {
      alert("Please enter a valid cell number!");
    } else {
      guesses = guesses + 1;
    }
    if (guess == location1 || guess == location2 || guess == location3) {
        hits = hits + 1;
    }
  }
}
```

LOOP: *while the ship is not sunk*

GET the user's guess

← Check the user's guess...

ADD one to guesses

The user's guess looks valid, so let's increase the number of guesses by one.

IF the user's guess matches a location

ADD one to the number of hits

If the guess matches one of the ship's locations we increment the hits counter.

We've combined the three conditionals into one if statement using || (OR). So read it like this: "If guess is equal to location1 OR guess is equal to location2 OR guess is equal to location3, increment hits."

Hey, you sank my battleship!

- [x] Create loop and get user guess
- [x] Check user guess
- [x] Check if ship has been sunk
- [ ] Display stats to user

We're almost there; we've almost got this game logic nailed down. Looking at the pseudocode again, what we need to do now is test to see if we have three hits. If we do, then we've sunk a battleship. And, if we've sunk a battleship then we need to set `isSunk` to true and also tell the user they've destroyed a ship. Let's sketch out the code again before adding it in:

First check to see if there are three hits.

Take another look at the while loop above. What happens when isSunk is true?

```
if (hits == 3) {
    isSunk = true;
    alert("You sank my battleship!");
}
```

And if so, set isSunk to true.

And also let the user know!

Provide some post-game analysis

- [x] Create loop and get user guess
- [x] Check user guess
- [x] Check if ship has been sunk
- [x] Display stats to user

After `isSunk` is set to true, the while loop is going to stop looping. That's right, this program we've come to know so well is going to stop executing the body of the while loop, and before you know it the game's going to be over. But, we still owe the user some stats on how they did. Here's some code that does that:

```
var stats = "You took " + guesses + " guesses to sink the battleship, " +
            "which means your shooting accuracy was " + (3/guesses);
alert(stats);
```

Here we're creating a string that contains a message to the user including the number of guesses they took, along with the accuracy of their shots. Notice that we're splitting up the string into pieces (to insert the variable guesses, and also to fit the string into multiple lines) using the concatenation operator, +. For now just type this as is, and we'll explain more about this later.

Now let's add this and the sunk ship detection into the rest of the code:

```
// Variable declarations go here

while (isSunk == false) {
    guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
    if (guess < 0 || guess > 6) {
        alert("Please enter a valid cell number!");
    } else {
        guesses = guesses + 1;

        if (guess == location1 || guess == location2 || guess == location3) {
            hits = hits + 1;

            if (hits == 3) {

                isSunk = true;

                alert("You sank my battleship!");
            }
        }
    }
}
var stats = "You took " + guesses + " guesses to sink the battleship, " +
            "which means your shooting accuracy was " + (3/guesses);
alert(stats);
```

LOOP: *while the ship is not sunk*

GET the user's guess

ADD one to guesses

IF the user's guess matches a location

ADD one to the number of hits

IF number of hits is 3

SET isSunk to true

TELL user "You sank my battleship!"

TELL user stats

Remember we said pseudocode often isn't perfect? Well we actually left something out of our original pseudocode: we're not telling the user if her guess is a HIT or a MISS. Can you insert these pieces of code in the proper place to correct this?

alert("HIT!");

else {

}

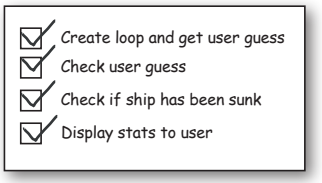
alert("MISS");

Here's the code you'll need to insert.

```
// Variable declarations go here

while (isSunk == false) {
    guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
    if (guess < 0 || guess > 6) {
        alert("Please enter a valid cell number!");
    } else {
        guesses = guesses + 1;
        if (guess == location1 || guess == location2 || guess == location3) {
            hits = hits + 1;
            if (hits == 3) {
              isSunk = true;
              alert("You sank my battleship!");
            }
        }
    }
}
var stats = "You took " + guesses + " guesses to sink the battleship, " +
               "which means your shooting accuracy was " + (3/guesses);
alert(stats);
```

This is a lot of curly braces to match. If you're having trouble matching them, just draw lines right in the book, to match them up.

And that completes the logic!



Alright! We've now fully translated the pseudocode to actual JavaScript code. We even discovered something we left out of the pseudocode and we've got that accounted for too. Below you'll find the code in its entirety. Make sure you have this typed in and saved in "battleship.js":

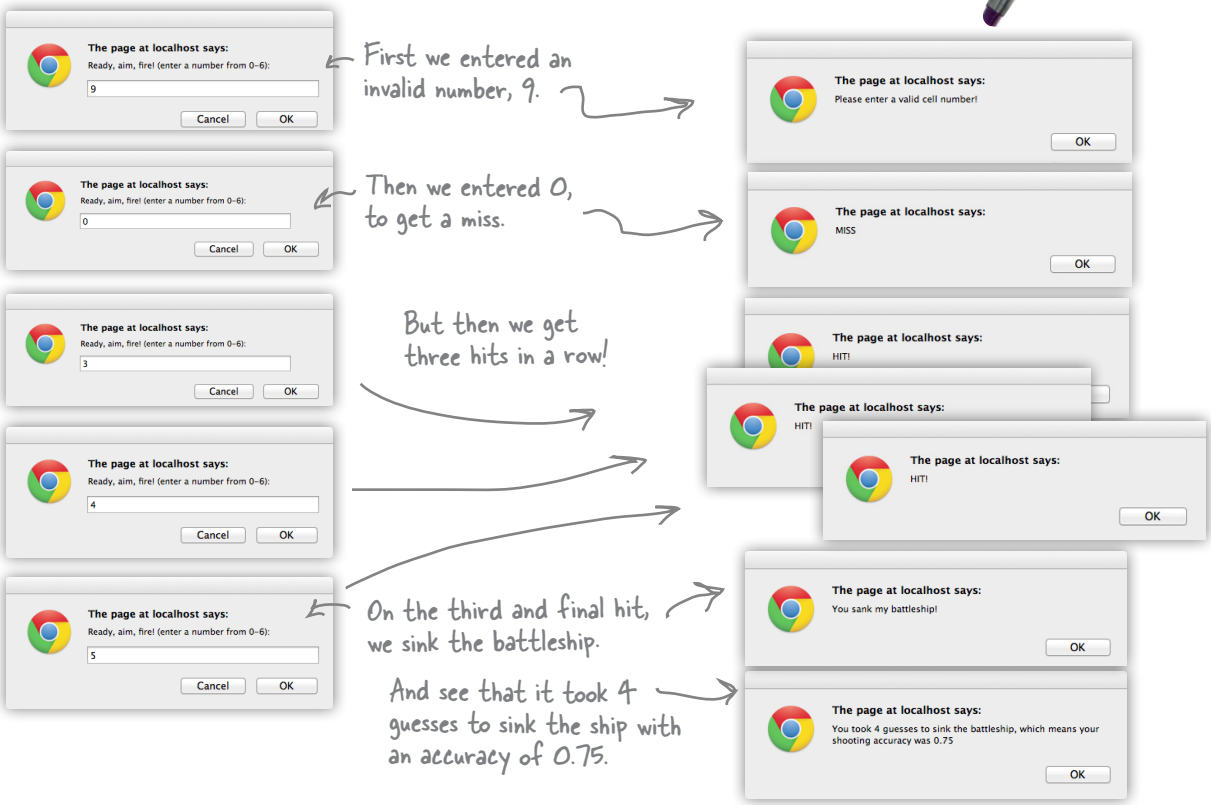
```
var location1 = 3;
var location2 = 4;
var location3 = 5;
var guess;
var hits = 0;
var guesses = 0;
var isSunk = false;

while (isSunk == false) {
    guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
    if (guess < 0 || guess > 6) {
        alert("Please enter a valid cell number!");
    } else {
        guesses = guesses + 1;

        if (guess == location1 || guess == location2 || guess == location3) {
            alert("HIT!");
            hits = hits + 1;
            if (hits == 3) {
                isSunk = true;
                alert("You sank my battleship!");
            }
        } else {
           alert("MISS");
        }
    }
}
var stats = "You took " + guesses + " guesses to sink the battleship, " +
               "which means your shooting accuracy was " + (3/guesses);
alert(stats);
```

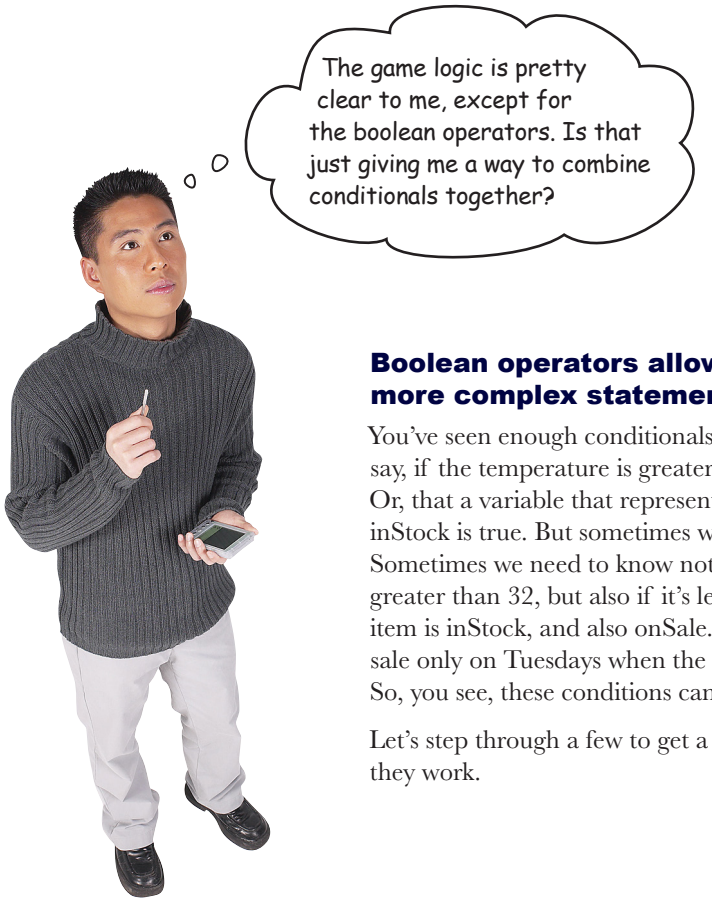
Doing a little Quality Assurance

QA, or quality assurance, is the process of testing software to find defects. So we're going to do a little QA on this code. When you're ready, load "battleship.html" in your browser and start playing. Try some different things. Is it working perfectly? Or did you find some issues? If so list them here. You can see our test run on this page too.

Jot down anything that doesn't work the way it should, or that could be improved.

Here's what our game interaction looked like.





Boolean operators allow you to write more complex statements of logic.

You've seen enough conditionals to know how to test, say, if the temperature is greater than 32 degrees. Or, that a variable that represents whether an item is inStock is true. But sometimes we need to test more. Sometimes we need to know not only if a value is greater than 32, but also if it's less than 100. Or, if an item is inStock, and also onSale. Or that an item is on sale only on Tuesdays when the user is a VIP member. So, you see, these conditions can get complex.

Let's step through a few to get a better idea of how they work.

Say we need to test that an item is inStock AND onSale. We could do that like this:

```
if (inStock == true) {
    if (onSale == true) {
        // sounds like a bargain!
        alert("buy buy buy!");
    }
}
```

First, see if the item is in stock...

And, if so, then see if it is on sale.

And if so, then take some action, like buy a few!

Notice this code is executed only if both conditionals are true!

We can simplify this code by combining these two conditionals together. Unlike in Simple Battleship, where we tested if guess < 0 OR guess > 6, here we want to know if inStock is true AND onSale is true. Let's see how to do that...

Here's our AND operator. With AND this combined conditional is true only if the first part AND the second part are true.

```
if (inStock == true  &&  onSale == true) {
        // sounds like a bargain!
        alert("buy buy buy!");
}
```

Not only is this code more concise, it's also more readable. Compare this code with the code on the previous page to see.

We don't have to stop there; we can combine boolean operators in multiple ways:

Now we're using both AND and OR in the same conditional expression. This one says: If an item is in stock AND it's either on sale, OR the price is less than 60, then buy.

```
if (inStock == true  &&  (onSale == true || price < 60)) {
        // sounds like a bargain!
        alert("buy buy buy!");
}
```

Notice we're using parentheses to group the conditions together so we get the result of the OR first, and then use that result to compute the result of the AND.

Sharpen your pencil

We've got a whole bunch of boolean expressions that need evaluating below. Fill in the blanks, and then check your answers at the end of the chapter before you go on.

```
var temp = 81;
var willRain = true;
var humid = (temp > 80 && willRain == true);
```

What's the value of **humid**? _____

```
var guess = 6;
var isValid = (guess >= 0 && guess <= 6);
```

What's the value of **isValid**? _____

```
var kB = 1287;
var tooBig = (kB > 1000);
var urgent = true;
var sendFile =
    (urgent == true || tooBig == false);
```

What's the value of **sendFile**? _____

```
var keyPressed = "N";
var points = 142;
var level;
if (keyPressed == "Y" ||
    (points > 100 && points < 200)) {
   level = 2;
} else {
   level = 1;
}
```

What's the value of **level**? _____

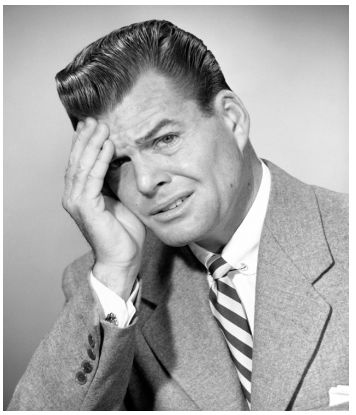
Exercise

Bob and Bill, both from accounting, are working on a new price checker application for their company's web site. They've both written if/else statements using boolean expressions. Both are sure they've written the correct code. Which accountant is right? Should these accountants even be writing code? Check your answer at the end of the chapter before you go on.

```
if (price < 200 || price > 600) {
   alert("Price is too low or too high! Don't buy the gadget.");
} else {
   alert("Price is right! Buy the gadget.");
}
```


```
if (price >= 200 || price <= 600) {
   alert("Price is right! Buy the gadget.");
} else {
   alert("Price is too low or too high! Don't buy the gadget.");
}
```

Can we talk about your verbosity...



We don't know how to bring this up, but you've been a little verbose in specifying your conditionals. What do we mean? Take this condition for instance:

We often compare our boolean variables to true or false to form our conditional.

```
if (inStock == true) {
  ...
}
```

And, inStock is a variable that holds a boolean value of true or false.

As it turns out, that's a bit of overkill. The whole point of a conditional is that it evaluates to either true or false, but our boolean variable `inStock` already *is* one of those values. So, we don't need to compare the variable to anything; it can just stand on its own. That is, we can just write this instead:

```
if (inStock) {
  ...
}
```

If we just use the boolean variable by itself, then if that variable is true, the conditional test is true, and the block is executed.

And if inStock is false, then the conditional test fails and the code block is skipped.

Now, while some might claim our original, verbose version was clearer in its intent, it's more common to see the more succinct version in practice. And, you'll find the less verbose version easier to read as well.

Exercise

We've got two statements below that use the onSale and inStock variables in conditionals to figure out the value of the variable buyIt. Work through each possible value of inStock and onSale for both statements. Which version is the biggest spender?

```
var buyIt = (inStock || onSale);
```

onSale	inStock	buyIt	buyIt
true	true		
true	false		
false	true		
false	false		

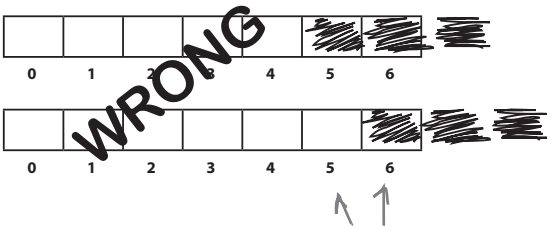
```
var buyIt = (inStock && onSale);
```

Finishing the Simple Battleship game

Yes, we still have one little matter to take care of because right now you've hard coded the location of the ship—no matter how many times you play the game, the ship is always at locations 3, 4 and 5. That actually works out well for testing, but we really need to randomly place the ship to make it a little more interesting to the user.

Let's step back and think about the right way to place a ship on the 1-D grid of seven cells. We need a starting location that allows us to place three consecutive positions on the grid. That means we need a starting location from zero to four.

We can start in locations 0, 1, 2, 3 or 4 and still have room to place the ship in the next three positions.



But, starting at position 5 or 6 won't work.

How to assign random locations

Now, once we have a starting location (between zero and four), we simply use it and the following two locations to hold the ship.

```
var location1 = randomLoc;
var location2 = location1 + 1;
var location3 = location2 + 1;
```

Take the random location along with the next two consecutive locations.

Okay, but how do we generate a random number? That's where we turn to JavaScript and its built-in functions. More specifically, JavaScript comes with a bunch of built-in math-related functions, including a couple that can be used to generate random numbers. Now we're going to get deeper into built-in functions, and functions in general a little later in the book. For now, we're just going to make use of these functions to get our job done.

The world-famous recipe for generating a random number

We're going to start with the `Math.random` function. By calling this function we'll get back a random decimal number:

Our variable randomLoc. We want to assign a number from 0 to 4 to this variable.

Math.random is part of standard JavaScript and returns a random number.

```
var randomLoc = Math.random();
```

The only problem is it returns numbers like 0.128, 0.830, 0.9, 0.42. These numbers are between 0 and 1 (not including exactly 1). So we need a way to use this to generate random numbers 0–4.

What we need is an integer between 0 and 4—that is, 0, 1, 2, 3 or 4—not a decimal number, like 0.34. To start, we could multiply the number returned by `Math.random` by 5 to get a little closer; here's what we mean…

First, if we multiply the random number by 5, then we get a number between 0 and 5, but not including 5. Like 0.13983, 4.231, 2.3451, or say 4.999.

```
var randomLoc = Math.random() * 5;
```

Remember, * means multiplication.

That's closer! Now all we need to do is clip off the end of the number to give us an integer number. To do that we can use another built-in Math function, Math.floor:

We can use Math.floor to round down all these numbers to their nearest integer value.

```
var randomLoc = Math.floor(Math.random() * 5);
```

So, for instance, 0.13983 becomes 0, 2.34 becomes 2 and 4.999 becomes 4.

there are no Dumb Questions

Q: If we're trying to generate a number between 0 and 4, why does the code have a 5 in it, as in `Math.floor(Math.random() * 5)`?

A: Good question. First, Math.random generates a number between 0 and 1, but not including 1. The maximum number you can get from Math.random is 0.999.... When you multiply that number by 5, the highest number you'll get is 4.999…
Math.floor always rounds a number down, so 1.2 becomes 1, but so does 1.9999. If we generate a number from 0 to 4.999… then everything will be rounded down to 0 to 4. This is not the only way to do it, and in other languages it's often done differently, but this is how you'll see it done in most JavaScript code.

Q: So if I wanted a random number between 0 and 100 (including 100), I'd write `Math.floor(Math.random() * 101)`?

A: That's right! Multiplying by 101, and using Math.floor to round down, ensures that your result will be at most 100.

Q: What are the parentheses for in Math.random()?

A: We use parentheses whenever we "call" a function. Sometimes we need to hand a value to a function, like we do when we use alert to display a message, and sometimes we don't, like when we use Math.random. But whenever you're calling a function (whether it's built-in or not), you'll need to use parentheses. Don't worry about this right now; we'll get into all these details in the next chapter.

Q: I can't get my battleship game to work. I'm not seeing anything in my web page except the "Play battleship" heading. How can I figure out what I did wrong?

A: This is where using the console can come in handy. If you've made an error like forgetting a quote on a string, then JavaScript will typically complain about the syntax of your program not being right, and may even show you the line number where your error is. Sometimes errors are more subtle, however. For instance, if you mistakenly write isSunk = false instead of isSunk == false, you won't see a JavaScript error, but your code won't behave as you expect it to. For this kind of error, try using console.log to display the values of your variables at various points in your code to see if you can track down the error.

Back to do a little more QA

That's all we need. Let's put this code together (we've already done that below) and replace your existing location code with it. When you're finished, give it a few test runs to see how fast you can sink the enemy.

```
var randomLoc = Math.floor(Math.random() * 5);
var location1 = randomLoc;
var location2 = location1 + 1;
var location3 = location2 + 1;
var guess;
var hits = 0;
var guesses = 0;
var isSunk = false;

while (isSunk == false) {
    guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
    if (guess < 0 || guess > 6) {
      // the rest of your code goes here....
```

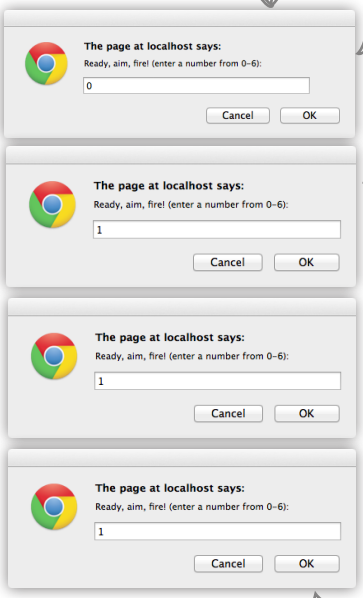
Go ahead and replace your location variable declarations with these new statements.

Here's one of our test sessions. The game's a little more interesting now that we've got random locations for the ship. But we still managed to get a pretty good score…

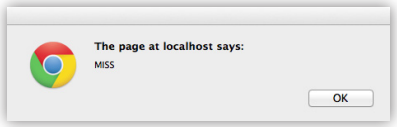
Exercise

Wait a sec, we noticed something that looks wrong. Hint: when we enter 0, 1, 1, 1 things don't look right! Can you figure out what's happening?

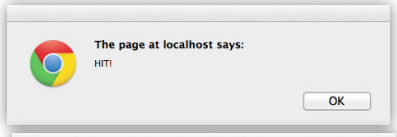
Here are our guesses...



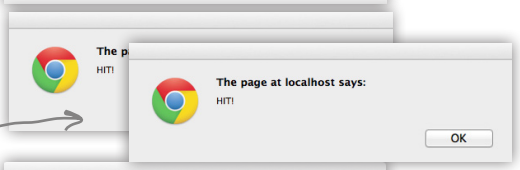
We miss on our first guess.



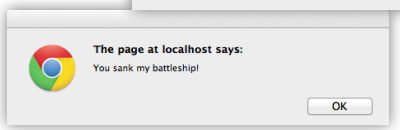
On our second guess we find a location of the ship.

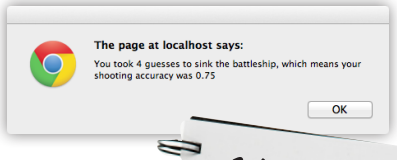


And then we keep entering that same location, and keep getting hits!

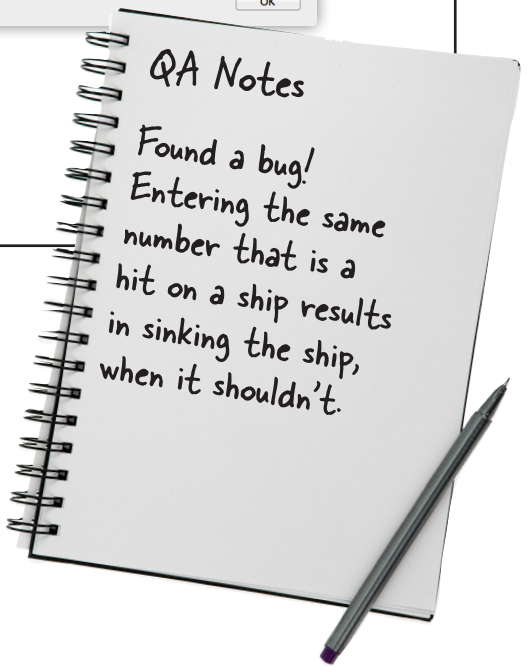


On the third hit, we see that we sank the battleship! But **something's wrong**. We shouldn't be able to sink it by hitting the same location three times.





We entered 0, 1, 1, 1, and the ship is at 1, 2, 3.



It's a cliff-hanger!

Will we ***find*** the bug?

Will we ***fix*** the bug?

Stay tuned for a much improved version of Battleship a little later in the book...

And in the meantime, see if you can come up with ideas for how you might fix the bug.

Congrats on your first true JavaScript program, and a short word about reusing code

You've probably noticed that we made use of a few *built-in functions* like `alert`, `prompt`, `console.log` and `Math.random`. With very little effort, these functions have given you the ability to pop up dialog boxes, log output to the console and generate random numbers, almost like magic. But, these built-in functions are just packaged up code that's already been written for you, and as you can see their power is that you can use and reuse them just by making a call to them when you need them.

Now there's a lot to learn about functions, how to call them, what kinds of values you can pass them, and so on, and we're going to start getting into all that in the next chapter where you learn to create your own functions.

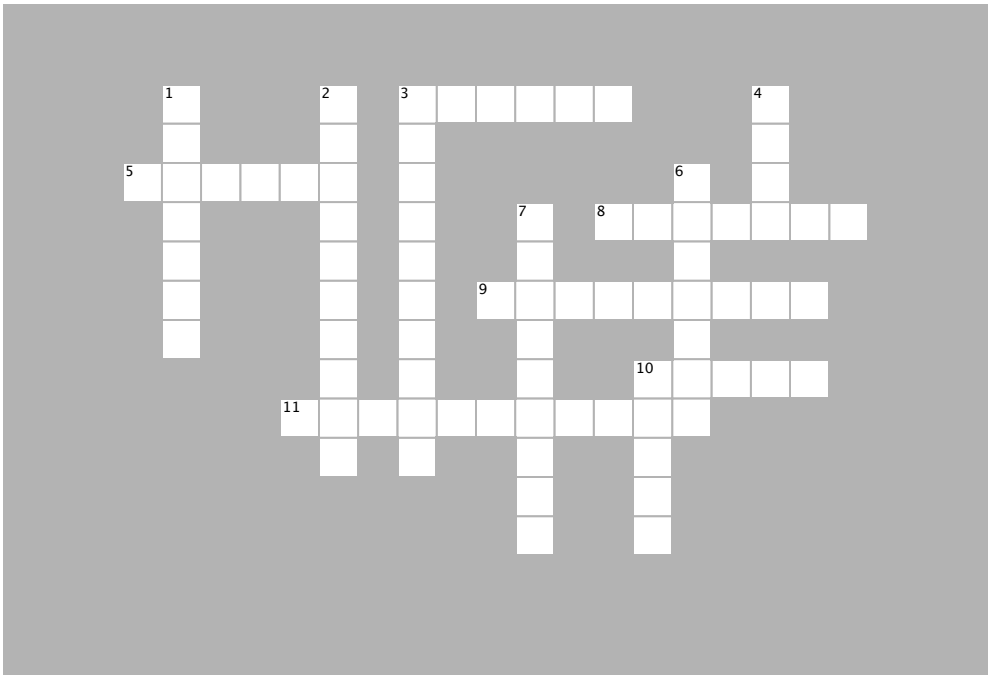
But before you get there you've got the bullet points to review, a crossword puzzle to complete… oh, and a good night's sleep to let everything sink in.

BULLET POINTS

- You can use a flowchart to outline the logic of a JavaScript program, showing decision points and actions.
- Before you begin writing a program, it's a good idea to sketch out what your program needs to do with pseudocode.
- **Pseudocode** is an approximation of what your real code should do.
- There are two kinds of boolean operators: comparison operators and logical operators. When used in an expression, boolean operators result in a true or false value.
- **Comparison** operators compare two values and result in true or false. For example, we can use the boolean comparison operator < ("less than") like this: 3 < 6. This expression results in true.
- **Logical** operators combine two boolean values. For example true || false results in true; true && false results in false.
- You can generate a random number between 0 and 1 (including 0, but not including 1) using the **Math.random** function.
- The **Math.floor** function rounds down a decimal number to the nearest integer.
- Make sure you use Math with an uppercase M, and not m, when using Math.random and Math.floor.
- The JavaScript function **prompt** shows a dialog with message and a space for the user to enter a value.
- In this chapter, we used prompt to get input from the user, and alert to display the results of the battleship game in the browser.

JavaScript cross

How does a crossword puzzle help you learn JavaScript? The mental twists and turns burn the JavaScript right into your brain!



ACROSS

3. To get input from a user, you can use the _____ function.

5. To randomly choose a position for a ship, use Math._____.

8. We keep track of whether a ship is sunk or not with a _____ variable.

9. If you don't initialize a variable, the value is _____.

10. Boolean operators always result in true or _____.

11. Both while and if statements use _____ tests.

DOWN

1. If you're good at testing programs, you might want to become a _____ Assurance specialist.

2. == is a _____ operator you can use to test to see if two values are the same.

3. This helps you think about how a program is going to work.

4. To get a true value from an AND operator (&&), both parts of the conditional must be _____.

6. OR (||) and AND (&&) are _____ operators.

7. JavaScript has many built-in _____ like alert and prompt.

10. To get a false value from an OR operator (||), both parts of the conditional must be _____.

Sharpen your pencil Solution

Let's say our virtual row looks like this:

And we've represented that by setting:

```
location1 = 3;
location2 = 4;
location3 = 5;
```

Assume the following user input:

```
1, 4, 2, 3, 5
```

Now, using the pseudocode on the previous page, trace through each step of code, and see how this works. Put your notes below. We've started the trace for you below. Here's our solution.

location1	location2	location 3	guess	guesses	hits	isSunk
3	4	5	—	0	0	false
3	4	5	1	1	0	false
3	4	5	4	2	1	false
3	4	5	2	3	1	false
3	4	5	3	4	2	false
3	4	5	5	5	3	true

Exercise Solution

We've got two statements below that use the onSale and inStock variables in conditionals to figure out the value of the variable buyIt. Work through each possible value of inStock and onSale for both statements. Which version is the biggest spender? The OR (||) operator!

```
var buyIt = (inStock || onSale);
```

onSale	inStock	buyIt	buyIt
true	true	true	true
true	false	true	false
false	true	true	false
false	false	false	false

```
var buyIt = (inStock && onSale);
```

Sharpen your pencil Solution

We've got a whole bunch of boolean expressions that need evaluating below. Fill in the blanks. Here's our solution:

```
var temp = 81;
var willRain = true;
var humid = (temp > 80 && willRain == true);
```

What's the value of **humid**? true

```
var guess = 6;
var isValid = (guess >= 0 && guess <= 6);
```

What's the value of **isValid**? true

```
var kB = 1287;
var tooBig = (kB > 1000);
var urgent = true;
var sendFile =
    (urgent == true || tooBig == false);
```

What's the value of **sendFile**? true

```
var keyPressed = "N";
var points = 142;
var level;
if (keyPressed == "Y" ||
    (points > 100 && points < 200)) {
   level = 2;
} else {
   level = 1;
}
```

What's the value of **level**? 2

Exercise Solution

Bob and Bill, both from accounting, are working on a new price checker application for their company's web site. They've both written if/else statements using boolean expressions. Both are sure they've written the correct code. Which accountant is right? Should these accountants even be writing code? Here's our solution.

Bob

```
if (price < 200 || price > 600) {
   alert("Price is too low or too high! Don't buy the gadget.");
} else {
   alert("Price is right! Buy the gadget.");
}
```

Bill

```
if (price >= 200 || price <= 600) {
   alert("Price is right! Buy the gadget.");
} else {
   alert("Price is too low or too high! Don't buy the gadget.");
}
```

Bob's the better coder (and possibly, a better accountant, too). Bob's solution works, but Bill's doesn't. To see why, let's try three different prices (too low, too high and just right) with Bob's and Bill's conditionals and see what results we get:

price	Bob's	Bill's
100	true alert: Don't buy!	true alert: Buy!
700	true alert: Don't buy!	true alert: Buy!
400	false alert: Buy!	true alert: Buy!

If price is 100, then 100 is less than 200, so Bob's conditional is true (remember, with OR, you only need one of the expressions to be true for the whole thing to be true), and we alert NOT to buy.

But Bill's conditional is also true, because price is <= 600! So the result of the entire expression is true, and we alert the user to buy, even though the price is too low.

Turns out Bill's conditional is always true, no matter what the price is, so his code tells us to Buy! every time. Bill should stick with accounting.

Remember we said pseudocode often isn't perfect? Well we actually left something out of our original pseudocode: we're not telling the user if her guess is a HIT or a MISS. Can you insert these pieces of code in the proper place to correct this? Here's our solution:

```
// Variable declarations go here


while (isSunk == false) {
    guess = prompt("Ready, aim, fire! (enter a number from 0-6):");
    if (guess < 0 || guess > 6) {
        alert("Please enter a valid cell number!");
    } else {
        guesses = guesses + 1;
        if (guess == location1 || guess == location2 || guess == location3) {
            alert("HIT!");
            hits = hits + 1;
            if (hits == 3) {
              isSunk = true;
              alert("You sank my battleship!");
            }
        } else {
            alert("MISS");
        }
    }
}
var stats = "You took " + guesses + " guesses to sink the battleship, " +
               "which means your shooting accuracy was " + (3/guesses);
alert(stats);
```

Sharpen your pencil Solution

What do you think of this first attempt to write the code to detect when a ship is hit? Does it look more complex than it needs to be, or are we repeating code in a way that seems a bit, well, redundant? Could we simplify it? Using what you know of the || operator (that is, the boolean OR operator), can you simplify this code? *Here's our solution.*

```
if (guess == location1) {
     hits = hits + 1;
} else  if (guess == location2) {
     hits = hits + 1;
} else if (guess == location3) {
     hits = hits + 1;
}
```

We're using the same code over and over here.

If we ever have to change how hits are updated, we've got three places to change our code. Changes like this are often a source of bugs and issues in code.

Not only that, this code is just way more complex than it needs to be. It's harder to read than it should be, and it took a lot more thought and typing than needed.

But, with the boolean OR operator we can combine the tests so that if location matches any of location1, location2 or location3, then the if conditional will be true, and the hits variable will be updated.

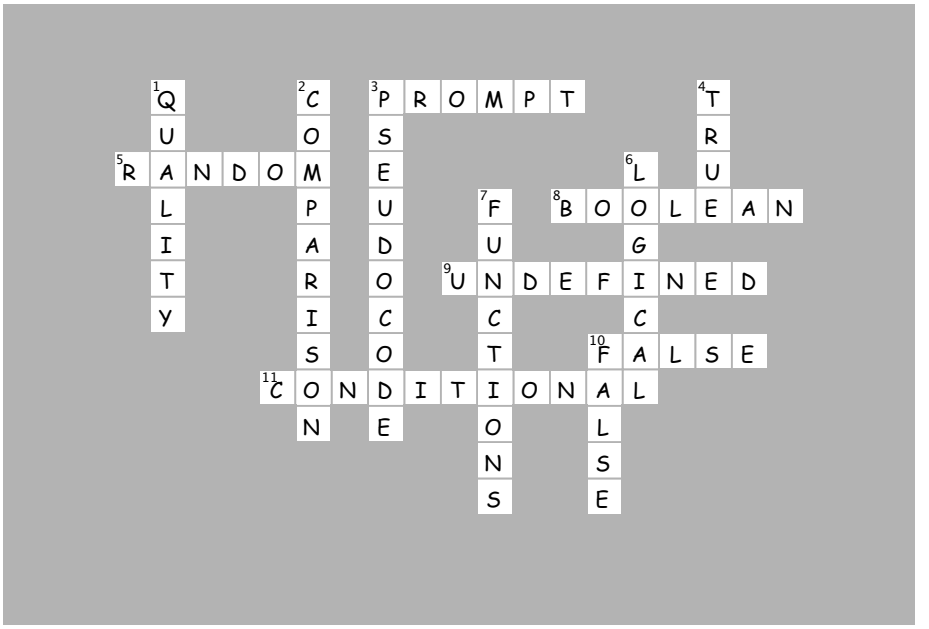
```
if (guess == location1 || guess == location2 || guess == location3) {
     hits = hits + 1;
}
```

Isn't that much easier on the eye? Not to mention easier to understand.

And if we ever have to change how hits is updated, well, then we only have one place to do it, which is much less error prone.

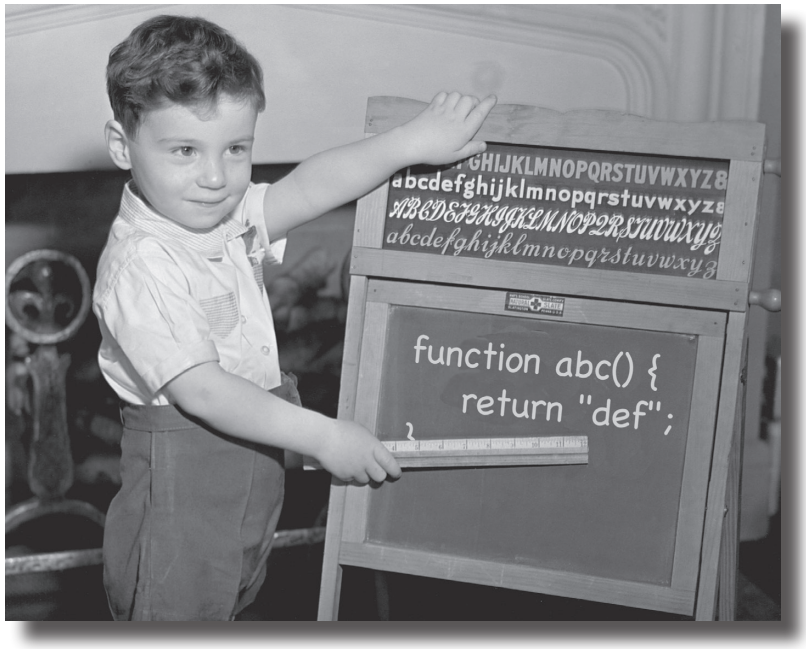
JavaScript cross Solution

How does a crossword puzzle help you learn JavaScript? The mental twists and turns burn the JavaScript right into your brain! Here's our solution.



3 introducing functions

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*.

More on this as we progress through the book.

Sharpen your pencil

Do a little analysis of the code below. How does it look? Choose as many of the options below as you like, or write in your own analysis:

```
var dogName = "rover";
var dogWeight = 23;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
dogName = "spot";
dogWeight = 13;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
dogName = "spike";
dogWeight = 53;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
dogName = "lady";
dogWeight = 17;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
```

_____

- ☐ A. The code seems very redundant.
- ☐ B. If we want to change the display of the output, or add another weight for dogs, this is going to require a lot of reworking.
- ☐ C. Looks tedious to type in!
- ☐ D. Not the most readable code I've ever seen.
- ☐ E. _____
  _____

What's wrong with the code anyway?

We just looked at some code that got used *over and over*. What's wrong with that? Well, at face value, nothing. After all, it works, right? Let's have a closer look at the code in question:

```
var dogName = "rover";
var dogWeight = 23;
if (dogWeight > 20) {
    console.log(dogName + " says WOOF WOOF");
} else {
    console.log(dogName + " says woof woof");
}
 ...
dogName = "lady";
dogWeight = 17;
if (dogWeight > 20) {
    console.log(dogName + " says WOOF WOOF");
} else {
    console.log(dogName + " says woof woof");
}
```

What we're doing here is comparing the dog's weight to 20, and if it's greater than 20, we're outputting a big WOOF WOOF. If it's less than 20, we're outputting a smaller woof woof.

And this code is... d'oh! It's doing EXACTLY the same thing. And so on, many times over in the rest of the code.

```
dogName = "spike";
dogWeight = 53;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
dogName = "lady";
dogWeight = 17;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
```

Sure, this code looks innocent enough, but it's tedious to write, a pain to read and will be problematic if your code needs to change, over time. That last point will ring true more and more as you gain experience in programming—all code changes over time and the code above is a nightmare waiting to happen because we've got the same logic repeated over and over, and if you need to change that logic, you'll have to change it in multiple places. And the bigger the program gets, the more changes you'll have to make, leading to more opportunities for mistakes. What we really want is a way to take redundant code like this and to put it in one place where it can be easily re-used whenever we need it.

How can we improve this code? Take a few minutes to think of a few possibilities. Does JavaScript have something that could help?


If only I could find a way to **reuse** code so that anytime I needed it, I could just **use** it rather than **retyping** it. And a way to give it a nice memorable **name** so that I could remember it. And a way to make changes in just **one** place instead of many if something changes. That would be dreamy. But I know it's just a fantasy...

By the way, did we happen to mention FUNCTIONS?

Meet *functions*. JavaScript functions allow you to take a bit of code, give it a name, and then refer to it over and over whenever we need it. That sounds like just the medicine we need.

Say you're writing some code that does a lot of "barking." If your code is dealing with a big dog then the bark is a big "WOOF WOOF". And if it's a small dog, the bark is a tiny "woof woof". You're going to need to use this barking functionality many times in your code. Let's write a `bark` function you can use over and over:

The function keyword begins a function definition.

Next we give the function a name, like bark.

And we're going to hand it two things when we get around to using it: a dog name and a dog weight.

We call these the parameters of the function. We put these in parentheses after the function name.

```
function bark(name, weight) {

}
```

Next we're going to write some code that gets executed when we use the function.

We'll call this the body of the function. It's everything inside the { and the }.

Now we need to write the code for the function; our code will check the weight and output the appropriate sized bark.

First we need to check the weight, and...

Notice the variable names used in the code match the parameters of the function.

```
function bark(name, weight) {
    if (weight > 20) {
        console.log(name + " says WOOF WOOF");
    } else {
        console.log(name + " says woof woof");
    }
}
```

...then output the dog's name with WOOF WOOF or woof woof.

Now you have a function you can use in your code. Let's see how that works next...

Okay, but how does it actually work?

First, let's rework our code using the new function `bark`:

```
function bark(name, weight) {
   if (weight > 20) {
      console.log(name + " says WOOF WOOF");
   } else {
      console.log(name + " says woof woof");
   }
}

bark("rover", 23);
bark("spot", 13);
bark("spike", 53);
bark("lady", 17);
```

Ahh, this is nice, all the logic of the code is here in one place.

Now all that code becomes just a few calls to the bark function, passing it each dog's name and weight.

Wow, now that's simple!

Wow, that's a lot less code—and it's so much more readable to your co-worker who needs to go into your code and make a quick change. We've also got all the logic in one convenient location.

Okay, but how exactly does it all come together and actually work? Let's go through it step by step.

First we have the function.

So we've got the `bark` function right at the top of the code. The browser reads this code, sees it's a function and then takes a look at the statements in the body. The browser knows it isn't executing the function statements now; it'll wait until the function is called from somewhere else in the code.

Notice too that the function is *parameterized*, meaning it takes a dog's name and weight when it is called. That allows you to call this function for as many different dogs as you like. Each time you do, the logic applies to the name and weight you pass to the function call.

Again, these are parameters; they are assigned values when the function is called.

```
function bark(name, weight) {
   if (weight > 20) {
      console.log(name + " says WOOF WOOF");
   } else {
      console.log(name + " says woof woof");
   }
}
```

And everything inside the function is the body of the function.

Now let's call the function.

"Invoking a function" is just a fancy way of saying "calling a function." Feel free to mix and match, especially when your new boss is around.

To call, or *invoke*, a function, just use its name, followed by an open parenthesis, then any values you need to pass it, separated by commas, and finally a closing parenthesis. The values in the parentheses are *arguments*. For the `bark` function we need two arguments: the dog's name and the dog's weight.

Here's how the call works:

Our function name.

Here we're passing two arguments, the name and the weight.

```
bark("rover", 23);
```

"rover" 23

When we call the bark function, the arguments are assigned to the parameter names.

And any time the parameters appear in the function, the values we passed in are used.

```
function bark(name, weight) {
   if (weight > 20) {
      console.log(name + " says WOOF WOOF");
   } else {
      console.log(name + " says woof woof");
   }
}
```

After you call the function, the body of the function does all the work.

After we know the value for each parameter—like `name` is "rover" and `weight` is 23—then we're ready to execute the function body.

Statements in the function body are executed from top to bottom, just like all the other code you've been writing. The only difference is that the parameter names `name` and `weight` have been assigned the values of the arguments you passed into the function.

After we've assigned the argument values to the parameter names, we then execute the statements in the body.

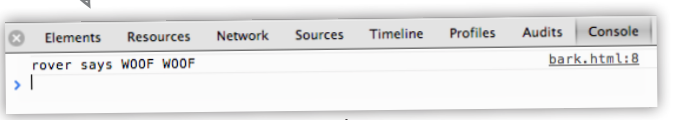
```
function bark(name, weight) {
   if (weight > 20) {
      console.log(name + " says WOOF WOOF");
   } else {
      console.log(name + " says woof woof");
   }
}
```

23 "rover" "rover"

The parameters act like variables in the body, which have been assigned the values of the arguments you passed in.

And when it's done... The logic of the body has been carried out (and, in this example, you'll see that Rover, being 23 pounds, sounds like "WOOF WOOF"), and the function is done. After the function completes, then control is returned to the statement following our call to `bark`.

Use your browser's Developer Tools to access the console so you can see the output of bark.



"rover says WOOF WOOF"

```
function bark(name, weight) {
   if (weight > 20) {
      console.log(name + " says WOOF WOOF");
   } else {
      console.log(name + " says woof woof");
   }
}
bark("rover", 23);
bark("spot", 13);
bark("spike", 53);
bark("lady", 17);
```

We just did this...

...so do this next.

When the function completes, the browser starts executing the next line of code after where we called the function.

Here, we're calling the function again, with different arguments, so the process starts all over again!

Sharpen your pencil

We've got some more calls to bark below. Next to each call, write what you think the output should be, or if you think the code will cause an error. Check your answer at the end of the chapter before you go on.

Write what you think the console log will display here.

Hmm, any ideas what these do?

```
bark("juno", 20); _____
bark("scottie", -1); _____
bark("dino", 0, 0); _____
bark("fido", "20"); _____
bark("lady", 10); _____
bark("bruno", 21); _____
```

Code Magnets

This working JavaScript code is all scrambled up on the fridge. Can you reconstruct the code snippets to make a working program that produces the output listed below? Notice, there may be some extra code on the fridge, so you may not use all the magnets. Check your answer at the end of the chapter.

```
,
,
,
}
}
}
(
function
)
{
whatShallIWear(80);
else {
    console.log("Wear t-shirt");
}
whatShallIWear
else if (temp < 70) {
    console.log("Wear a sweater");
}
temperature
if (temp < 60) {
    console.log("Wear a jacket");
}
temp
whatShallIWear(60);
whatShallIWear(50);
```

JavaScript console

```
Wear a jacket
Wear a t-shirt
Wear a sweater
```

We're using this to represent a generic console.

The Function Exposed

This week's interview: the intimate side of functions...

Head First: Welcome Function! We're looking forward to digging in and finding out what you're all about.

Function: Glad to be here.

Head First: Now we've noticed many JavaScript newbies tend to ignore you. They just get in and write their code, line by line, top to bottom, no functions at all. Are you really needed?

Function: Those newbies are missing out. That's unfortunate because I'm powerful. Think about me like this: I give you a way to take code, write it once, and then reuse it over and over.

Head First: Well, excuse me for saying this, but if you're just giving them the ability to do the same thing, over and over... that's a little boring isn't it?

Function: No no, functions are parameterized—in other words, each time you use the function, you pass it arguments so it can compute something that's relevant to what you need.

Head First: Err, example?

Function: Let's say you need to show a user how much the items in his shopping cart are going to cost, so you write a function `computeShoppingCartTotal`. Then you can pass that function the shopping carts of many users and each time I compute the amount of each specific shopping cart.

Head First: If you're so great, why aren't more new coders using you?

Function: That's not even a true statement; they use me all the time: `alert`, `prompt`, `Math.random`, `document.write`. It's hard to write anything meaningful without using functions. It's not so much that new users don't use functions, they just aren't defining *their own* functions.

Head First: Well, right, `alert` and `prompt`, those make sense, but take `Math.random`—that doesn't look quite like a function.

Function: `Math.random` is a function, but it happens to be attached to another powerful thing new coders don't make a lot of use of: *objects*.

Head First: Oh yes, objects. I believe our readers are learning about those in a later chapter.

Function: Fair enough, I'll save my breath on that one for later.

Head First: Now this argument/parameter stuff all seems a little confusing.

Function: Think about it like this: each parameter acts like a variable throughout the body of the function. When you call the function, each value you pass in is assigned to a corresponding parameter.

Head First: And arguments are what?

Function: Oh, that's just another name for the values you pass into a function… they're the arguments of the function call.

Head First: Well you don't seem all that great; I mean, okay you allow me to reuse code, and you have this way of passing values as parameters. Is that it? I don't get the mystery around you.

Function: Oh, that's just the basics, there's so much more: I can return values, I can masquerade around your code anonymously, I can do a neat trick called closures, and I have an *intimate* relationship with objects.

Head First: Ohhhhh REALLY?! Can we get an exclusive on that relationship for our next interview?

Function: We'll talk…

What can you pass to a function?

When you call a function you pass it arguments and those arguments then get matched up with the parameters in the function definition. You can pass pretty much any JavaScript value as an argument, like a string, a boolean, or a number:

Pass any JavaScript value as an argument.

Each argument is passed to its corresponding parameter in the function.

```
saveMyProfile("krissy", 1991, 3.81, false);

function saveMyProfile(name, birthday, GPA, newuser) {
    if (birthday >= 2004) {
        // code for handling a child
    }
   // rest of code for this function here
}
```

And each parameter acts as a variable within the function.

You can also pass variables as arguments, and that's often the more common case. Here's the same function call using variables:

```
var student = "krissy";
var year = 1991;
var GPA = 381/100;
var status = "existinguser";
var isNewUser = (status == "newuser");
saveMyProfile(student, year, GPA, isNewUser);
```

Now, each of the values we're passing is stored in a variable. When we call the function, the variable's values are passed as the arguments.

So, in this case we're passing the value in the variable student, "krissy", as the argument to the name parameter.

And we're also using variables for these other arguments.

And, you can even use expressions as arguments:

```
var student = "krissy";
var status = "existinguser";
var year = 1991;

saveMyProfile(student, year, 381/100, status == "newuser");
```

In each case we first evaluate the expression to a value, and then that value is passed to the function.

Yes, even these expressions will work as arguments!

We can evaluate a numeric expression...

... or a boolean expression, like this one that results in passing false to the function.

I'm still not sure I get the difference between a parameter and an argument—are they just two names for the same thing?

No, they're different.

When you define a function you can *define* it with one or more *parameters*.

Here we're defining three parameters: degrees, mode and duration.

```
function cook(degrees, mode, duration) {
   // your code here
}
```

When you call a function, you *call* it with *arguments*:

```
cook(425.0, "bake", 45);
```

These are arguments. There are three arguments: a floating point number, a string and an integer.

```
cook(350.0, "broil", 10);
```

So you'll only define your parameters once, but you'll probably call your function with many different arguments.

What does this code output? Are you sure?

```
function doIt(param) {
    param = 2;
}
var test = 1;
doIt(test);
console.log(test);
```

Exercise

Below you'll find some JavaScript code, including variables, function definitions and function calls. Your job is to identify all the variables, functions, arguments and parameters. Write the names of each in the appropriate boxes on the right. Check your answer at the end of the chapter before you go on.

```
function dogYears(dogName, age) {
    var years = age * 7;
    console.log(dogName + " is " + years + " years old");
}
var myDog = "Fido";
dogYears(myDog, 4);

function makeTea(cups, tea) {
    console.log("Brewing " + cups + " cups of " + tea);
}
var guests = 3;
makeTea(guests, "Earl Grey");

function secret() {
    console.log("The secret of life is 42");
}
secret();

function speak(kind) {
    var defaultSound = "";
    if (kind == "dog") {
        alert("Woof");
    } else if (kind == "cat") {
        alert("Meow");
    } else {
        alert(defaultSound);
    }
}
var pet = prompt("Enter a type of pet: ");
speak(pet);
```

Variables

Functions

Built-in functions

Arguments

Parameters

JavaScript is pass-by-value.

That means pass-by-copy.

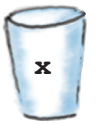
It's important to understand how JavaScript passes arguments. JavaScript passes arguments to a function using *pass-by-value*. What that means is that each argument is *copied* into the parameter variable. Let's look at a simple example to see how this works.

1 Let's declare a variable age, and initialize it to the value 7.

```
var age = 7;
```

2 Now let's declare a function addOne, with a parameter named x, that adds 1 to the value of x.

```
function addOne(x) {
    x = x + 1;
}
```



3 Now let's call the function addOne, pass it the variable age as the argument. The value in age is copied into the parameter x.

```
addOne(age);
```

4 Now the value of x is incremented by one. But remember x is a copy, so only x is incremented, not age.

We're incrementing x.

```
function addOne(x) {
    x = x + 1;
}
```

So how do I think about this pass-by-value stuff? On the one hand it feels pretty straightforward, and on the other hand I feel like I might be missing something.

We're glad you're thinking about it. Understanding how JavaScript passes values to functions is important. On the one hand it is pretty straightforward: when an argument is passed to a function its value is first *copied* and then assigned to the corresponding parameter. But, if you don't understand this, you can make some wrong assumptions about how functions, arguments and parameters all work together.

The real impact of pass-by-value is that any changes to a parameter's value within the function will affect *only the parameter*, not the original variable passed to the function. That's pretty much it.

But of course, there's an exception to every rule, and we're going to have to talk about this topic again when we learn objects, which we'll talk about in a couple of chapters. But no worries, with a solid understanding of pass-by-value, you're in good shape to have that discussion.

And, for now, just remember that because of pass-by-value, *whatever happens to a parameter in the function, stays in the function.* Kinda like Vegas.

BRAIN POWER REVISITED

Remember this Brain Power? Do you think about it differently now, knowing about pass by value? Or did you guess correctly the first time?

```
function doIt(param) {
    param = 2;
}
var test = 1;
doIt(test);
console.log(test);
```

WEIRD FUNCTIONS

So far you've seen the normal way to use functions, but what happens when you experiment a little by, say, passing too many arguments to a function? Or not enough? Sounds dangerous. Let's see what happens:

EXPERIMENT #1: what happens when we don't pass enough arguments?

Sounds dicey, but all that really happens is each parameter that doesn't have a matching argument is set to undefined. Here's an example:

```
function makeTea(cups, tea) {
  console.log("Brewing " + cups + " cups of " + tea);
}
makeTea(3);
```

Notice that the value of the parameter tea is undefined because we didn't pass in a value.

JavaScript console

```
Brewing 3 cups of undefined
```

EXPERIMENT #2: what happens when we pass too many argments?

Ah, in this case JavaScript just ignores the extra. Here's an example:

```
function makeTea(cups, tea) {
  console.log("Brewing " + cups + " cups of " + tea);
}
makeTea(3, "Earl Grey", "hey ma!", 42);
```

Works fine, the function ignores the extras.

JavaScript console

```
Brewing 3 cups of Earl Grey
```

There's actually a way to get at the extra arguments, but we won't worry about that just now...

EXPERIMENT #3: what happens when we have NO parameters?

No worries, many functions have no parameters!

```
function barkAtTheMoon() {
  console.log("Wooooooooooooo!");
}
barkAtTheMoon();
```

JavaScript console

```
Wooooooooooooo!
```

Functions can return things too

You know how to communicate with your functions in one direction; that is, you know how to pass arguments *to functions*. But what about the other way? Can a function communicate back? Let's check out the `return` statement:

Here we've got a new bake function that takes the temperature in degrees for the oven.

```
function bake(degrees) {
    var message;

    if (degrees > 500) {
        message = "I'm not a nuclear reactor!";
    } else if (degrees < 100) {
        message = "I'm not a refrigerator!";
    } else {
        message = "That's a very comfortable temperature for me.";
        setMode("bake");
        setTemp(degrees);
    }

    return message;
}

var status = bake(350);
```

It then sets a variable to a string that depends on the temperature requested in the degrees parameter.

And presumably some real work is getting done here, but we won't worry about those details for now...

What we care about is that a return statement is returning the message as the result of this function.

Now, when the function is called and returns, the string that is returned as a result will be assigned to the status variable.

And in this case, if the status variable was printed, it would hold the string "That's a very comfortable temperature for me." Work through the code and make sure you see why!

350 degrees is the perfect temperature for good cookies. Feel free to make some and return to the next page.

Tracing through a function with a return statement

Now that you know how arguments and parameters work, and how you can return a value from a function, let's trace through a function call from start to finish to see what happens at every step along the way. Be sure to follow the steps in order.

1. First, we declare a variable radius and initialize it to 5.2.
2. Next, we call the calculateArea function, and pass the radius variable as the argument.
3. The argument is sent to the parameter r, and the calculateArea function begins executing with r containing the value 5.2.
4. The body of the function executes starting with declaring a variable, area. We then test to see if the parameter r has a value <= 0.
5. If r <= 0, then we return 0 from the function and the function stops executing. But we passed in 5.2 so this line does NOT execute.
6. We execute the else clause instead.
7. We compute the area of the circle using the value 5.2 in the parameter r.
8. We return the value of area from the function. This stops the execution of the function and returns the value.
9. The value returned from the function is stored in the variable theArea.
10. Execution continues on the next line.

```
3 function calculateArea(r) {
      var area;
4     if (r <= 0) {
5       return 0;
6     } else {
7       area = Math.PI * r * r;
8       return area;
      }
  }
1 var radius = 5.2;
              2
9 var theArea = calculateArea(radius);

10 console.log("The area is: " + theArea);
```

Output here!

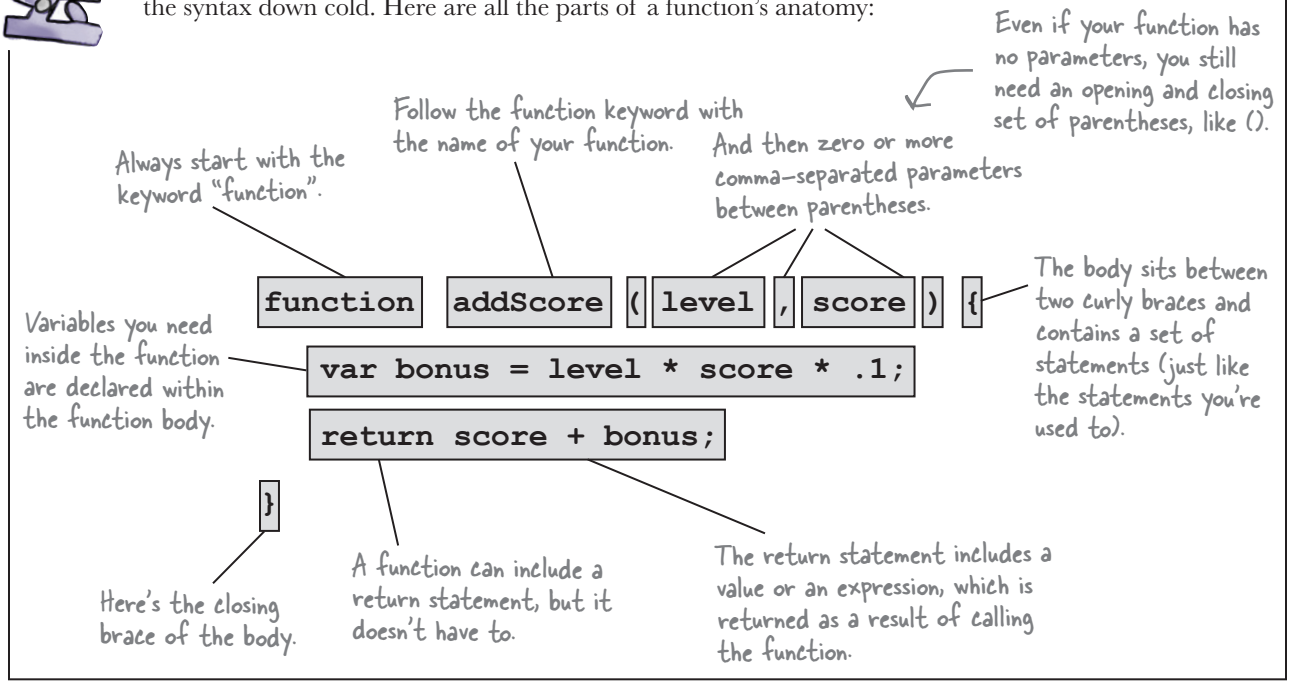
JavaScript console

```
The area is: 84.94866535306801
```

Developers often call this "tracing the flow of execution" or just "tracing." As you can see the flow can jump around when you're calling functions and returning values. Just take it slow, one step at a time.

Anatomy of a Function

Now that you know how to define and call a function, let's make sure we've got the syntax down cold. Here are all the parts of a function's anatomy:



there are no Dumb Questions

Q: What happens if I mix up the order of my arguments, so that I'm passing the wrong arguments into the parameters?

A: All bets are off; in fact, we'd guess you're pretty much guaranteed either an error at run time or incorrect code. Always take a careful look at a function's parameters, so you know what arguments the function expects to be passed in.

Q: Why don't the parameter names have var in front of them? A parameter is a new variable right?

A: Effectively yes. The function does all the work of instantiating the variable for you, so you don't need to supply the var keyword in front of your parameter names.

Q: What are the rules for function names?

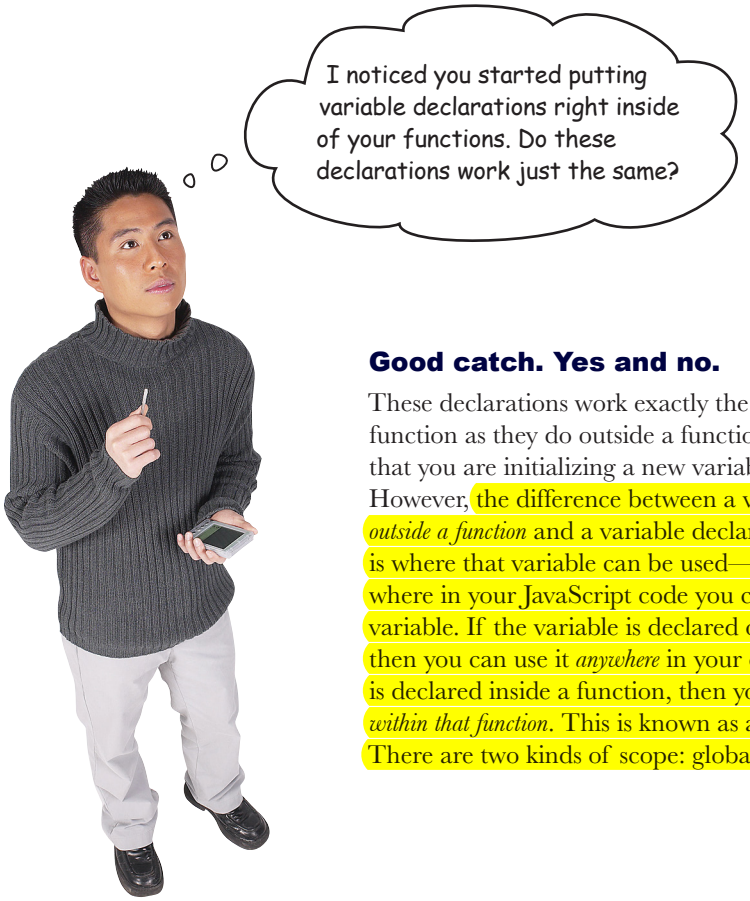
A: The rules for naming a function are the same as the rules for naming a variable. Just like with variables, you'll want to use names that make sense to you when you read them, and provide some indication of what the function does, and you can use camel case (e.g. camelCase) to combine words in function names, just like with variables.

Q: What happens if I use the same name for an argument variable as the parameter? Like if I use the name x for both?

A: Even if your argument and parameter have the same name, like x, the parameter x gets a copy of the argument x, so they are two different variables. Changing the value of the parameter x does not change the value of the argument x.

Q: What does a function return if it doesn't have a return statement?

A: A function without a return statement returns `undefined`.



Good catch. Yes and no.

These declarations work exactly the same within a function as they do outside a function, in the sense that you are initializing a new variable to a value. However, the difference between a variable declared *outside a function* and a variable declared *inside a function* is where that variable can be used—in other words, where in your JavaScript code you can reference the variable. If the variable is declared outside a function, then you can use it *anywhere* in your code. If a variable is declared inside a function, then you can use it only *within that function*. This is known as a variable's *scope*. There are two kinds of scope: global and local.

Global and local variables

Know the difference or risk humiliation

You already know that you can declare a variable by using the `var` keyword and a name anywhere in your script:

```
var avatar;
var levelThreshold = 1000;
```

These are global variables; they're accessible everywhere in your JavaScript code.

And you've seen that you can also declare variables inside a function:

```
function getScore(points) {
    var score;
    var i = 0;
    while (i < levelThreshold) {
       //code here
       i = i + 1;
    }
    return score;
}
```

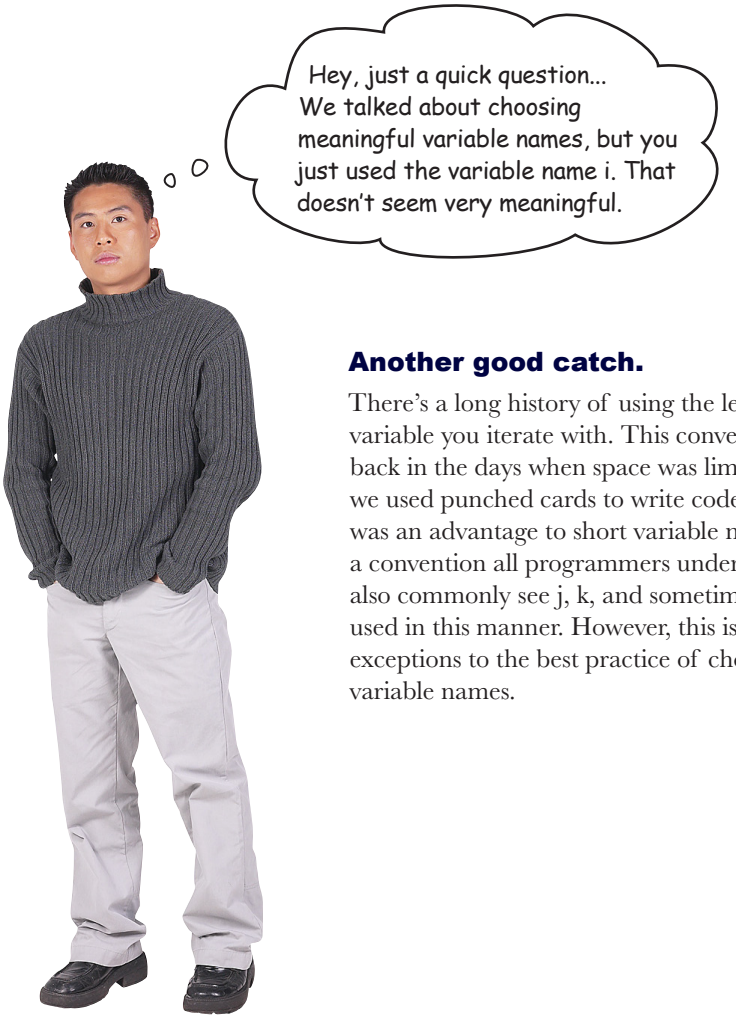
The points, score and i variables are all declared within a function.

We call them local variables because they are known locally only within the function itself.

Even if we use levelThreshold inside the function, it's global because it's declared outside the function.

If a variable is declared outside a function, it's GLOBAL. If it's declared inside a function, it's LOCAL.

But what does it matter? Variables are variables, right? Well, *where* you declare your variables determines *how visible* they are to other parts of your code, and, later, understanding how these two kinds of variables operate will help you write more maintainable code (not to mention, help you understand the code of others).



Another good catch.

There's a long history of using the letter i as the variable you iterate with. This convention developed back in the days when space was limited (like when we used punched cards to write code), and there was an advantage to short variable names. Now it's a convention all programmers understand. You'll also commonly see j, k, and sometimes even x and y used in this manner. However, this is one of the only exceptions to the best practice of choosing meaningful variable names.

Knowing the scope of your local and global variables

Where you define your variables determines their *scope*; that is, where they're visible to your code and where they aren't. Let's look at an example of both locally and globally scoped variables—remember, the variables you define outside a function are globally scoped, and the function variables are locally scoped:

```
var avatar = "generic";
var skill = 1.0;
var pointsPerLevel = 1000;
var userPoints = 2008;

function getAvatar(points) {
    var level = points / pointsPerLevel;

    if (level == 0) {
       return "Teddy bear";
    } else if (level == 1) {
       return "Cat";
    } else if (level >= 2) {
       return "Gorilla";
    }
}

function updatePoints(bonus, newPoints) {
    var i = 0;
    while (i < bonus) {
        newPoints = newPoints + skill * bonus;
        i = i + 1;
    }
    return newPoints + userPoints;
}

userPoints = updatePoints(2, 100);
avatar = getAvatar(2112);
```

These four variables are globally scoped. That means they are defined and visible in all the code below.

Note that if you link to additional scripts in your page, they will see these global variables, and you will see their global variables too!

The level variable here is local and is visible only to the code within the getAvatar function. That means only this function can access the level variable.

And let's not forget the points parameter, which also has local scope in the getAvatar function.

Note that getAvatar makes use of the pointsPerLevel global variable too.

In updatePoints we have a local variable i. i is visible to all of the code in updatePoints.

bonus and newPoints are also local to updatePoints, while userPoints is global.

And here in our code we can use only the global variables, we have no access to any variables inside the functions because they're not visible in the global scope.

The short lives of variables

When you're a variable, you work hard and life can be short. That is, unless you're a global variable, but even with globals, life has its limits. But what determines the life of a variable? Think about it like this:

Globals live as long as the page. A global variable begins life when its JavaScript is loaded into the page. But, your global variable's life ends when the page goes away. Even if you reload the same page, all your global variables are destroyed and then recreated in the newly loaded page.

Local variables typically disappear when your function ends. Local variables are created when your function is first called and live until the function returns (with a value or not). That said, you can take the values of your local variables and return them from the function before the variables meet their digital maker.

We say "typically" because there are some advanced ways to retain locals a little longer, but we're not going to worry about them now.

So, there really is NO escape from the page is there? If you're a local variable, your life comes and goes quickly, and if you're lucky enough to be a global, you're good as long as that browser doesn't reload the page.

Don't forget to declare your locals!

If you use a variable without declaring it first, that variable will be global. That means that even if you use a variable for the first time inside a function (because you meant for it to be local), the variable will actually be global, and be available outside the function too (which might cause confusion later). So, don't forget to declare your locals!

If you forget to declare a variable before using it, the variable will always be global (even if the first time you use it is in a function).

```
function playTurn(player, location) {
    points = 0;
    if (location == 1) {
        points = points + 100;
    }
    return points;
}
var total = playTurn("Jai", 1);
alert(points);
```

We forgot to declare points with "var" before we used it. So points is automatically global.

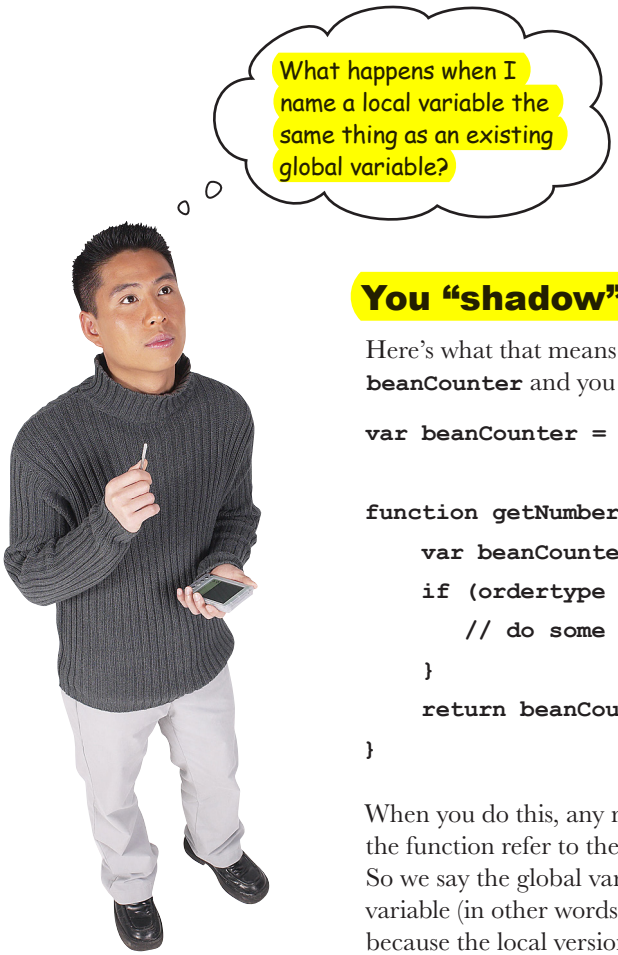
That means we can use points outside the function! The value doesn't go away (like it should) when the function is done executing.

This program behaves as if you'd written this instead:

JavaScript assumes that we meant for points to be global because we forgot to use "var", and behaves as if points were declared at the global level.

```
var points = 0;
function playTurn(player, location) {
    points = 0;
    if (location == 1) {
        points = points + 100;
    }
    return points;
}
var total = playTurn("Jai", 1);
alert(points);
```

Forgetting to declare your local variables can cause problems if you use the same name as another global variable. You might overwrite a value you didn't mean to.



You "shadow" your global.

Here's what that means: say you have a global variable **beanCounter** and you then declare a function, like this:

```
var beanCounter = 10;


function getNumberOfItems(ordertype) {
    var beanCounter = 0;
    if (ordertype == "order") {
       // do some stuff with beanCounter...
    }
    return beanCounter;
}
```

We've got a global and a local!

When you do this, any references to **beanCounter** within the function refer to the local variable and not the global. So we say the global variable is in the shadow of the local variable (in other words we can't see the global variable because the local version is in our way).

Note that the local and global variables have no effect on each other: if you change one, it has no effect on the other. They are independent variables.

Exercise

Below you'll find some JavaScript code, including variables, function definitions and function calls. Your job is to identify the variables used in all the arguments, parameters, local variables and global variables. Write the variable names in the appropriate boxes on the right. Then circle any variables that are shadowed. Check your answer at the end of the chapter.

```
var x = 32;
var y = 44;
var radius = 5;

var centerX = 0;
var centerY = 0;
var width = 600;
var height = 400;

function setup(width, height) {
    centerX = width/2;
    centerY = height/2;
}

function computeDistance(x1, y1, x2, y2) {
    var dx = x1 - x2;
    var dy = y1 - y2;
    var d2 = (dx * dx) + (dy * dy);
    var d = Math.sqrt(d2);
    return d;
}

function circleArea(r) {
    var area = Math.PI * r * r;
    return area;
}

setup(width, height);
var area = circleArea(radius);
var distance = computeDistance(x, y, centerX, centerY);
alert("Area: " + area);
alert("Distance: " + distance);
```

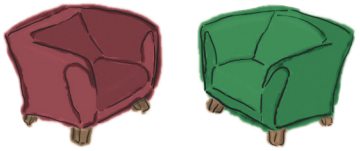
Arguments

Parameters

Locals

Globals

Fireside Chats



Tonight's talk: **Global and Local variables argue over who is most important in a program.**

Global variable:

Hey Local, I'm really not sure why you're here because I can handle any need for a variable a coder might have. After all, I'm visible everywhere!

You have to admit that I could replace all your previous local variables with global ones and your functions would work just the same.

It wouldn't have to be a mess. Programmers could just create all the variables they need up at the top of a program, so they'd all be in one place…

Well, if you'd use better names, then you might be able to keep track of your variables more easily.

True. But why bother with arguments and parameters if you've got all the values you need in globals?

Local variable:

Yes but using globals everywhere is just bad style. Lots of functions need variables that are local. You know, their own private variables for their own use. Globals can be seen everywhere.

Well, yes and no. If you're extremely careful, sure. But being that careful is difficult, and if you make a mistake, then we've got functions using variables that other functions are using for different purposes. You'd also be littering the program with global variables that you only need inside one function call… that would just make a huge mess.

Yeah, and so what happens if you need to call a function that needs a variable, like, oh I dunno, x, and you can't remember what you've used x for before. You have to go searching all over your code to see if you've used x anywhere else! What a nightmare.

And what about parameters? Function parameters are always local. So you can't get around that.

Excuse me, do you hear what you're saying? The whole point of functions is so we can reuse code to compute different things based on different inputs.

Global variable:

But your variables are just so... temporary. Locals come and go at a moment's notice.

Not at all? Globals are the mainstay of JavaScript programmers!

I think I need a drink.

Local variable:

Face it. It's just good programming practice to use local variables unless you absolutely need globals. And globals can get you into real trouble. I've seen JavaScript programs that barely use globals at all!

Of inexperienced programmers, sure. But as programmers learn to correctly structure their code for correctness, maintainability, and just good coding style, they learn how to stop using globals except when necessary.

They let globals drink? Now, we're *really* in dangerous territory.

there are no Dumb Questions

Q: Keeping track of the scope of all these locals and globals is confusing, so why not just stick to globals? That's what I've always done.

A: If you're writing code that is complex or that needs to be maintained over a long period of time, then you really have to watch how you manage your variables. When you're overzealous in creating global variables, it becomes difficult to track where your variables are being used (and where you're making changes to your variables' values), and that can lead to buggy code. All this becomes even more important when you're writing code with co-workers or you're using third-party libraries (although if those libraries are written well, they should be structured to avoid these issues).

So, use globals where it makes sense, but use them in moderation, and whenever possible, make your variables local. As you get more experience with JavaScript, you can investigate additional techniques to structure code so that it's more maintainable.

Q: I have global variables in my page, but I'm loading in other JavaScript files as well. Do those files have separate sets of global variables?

A: There is only one global scope so every file you load sees the same set of variables (and creates globals in the same space). That's why it is so important you be careful with your use of variables to avoid clashes (and reduce or eliminate global variables when you can).

Q: If I use the same name for a parameter as I do for a global variable, does the parameter shadow the global?

A: Yes. Just like if you declare a new, local variable in a function with the same name as a global, if you use the same name for a parameter as a global, you're also going to shadow the global with that name. It's perfectly fine to shadow a global name as long as you don't want to use the global variable inside your function. But it's a good idea to document what you're doing with comments so you don't get confused later when you're reading your code.

Q: If I reload a page in the browser, do the global variables all get re-initialized?

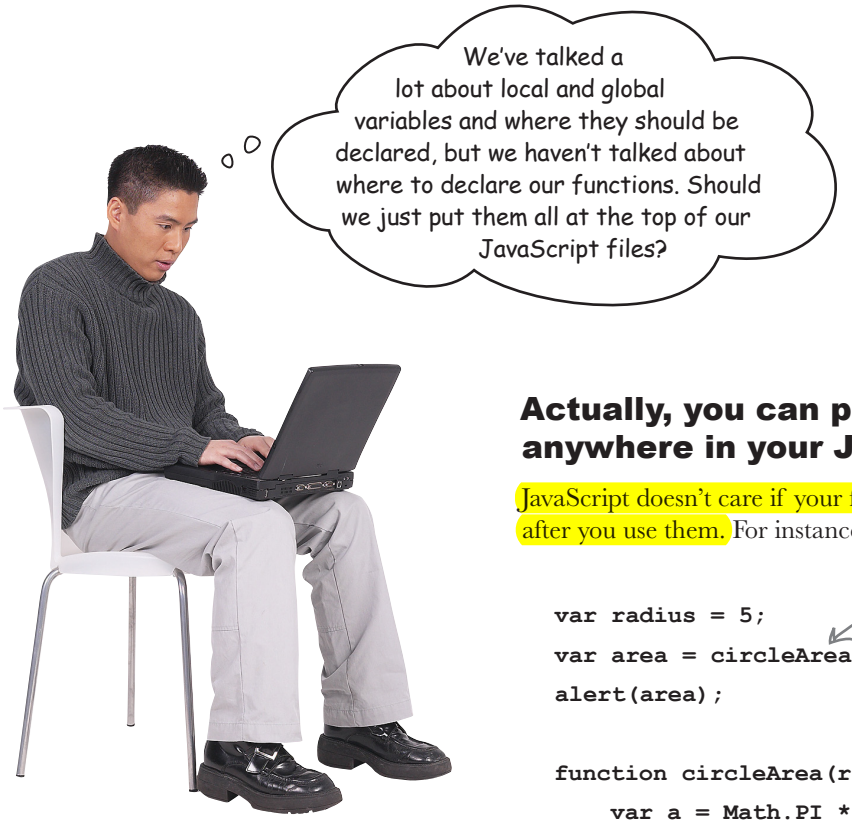
A: Yes. Reloading a page is like starting over from scratch as far as the variables are concerned. And if any code was in the middle of executing when you reload the page, any local variables will disappear, too.

Q: Should we always declare our local variables at the top of a function?

A: Just like with global variables, you can declare local variables when you first need to use them in a function. However, it's a good programming practice to go ahead and declare them at the top of your function so someone reading your code can easily find those declarations and get a sense at a glance of all the local variables used within the function. In addition, if you delay declaring a variable and then decide to use that variable earlier in the body of the function than you originally anticipated, you might get behavior that you don't expect. JavaScript creates all local variables at the beginning of a function whether you declare them or not (this is called "hoisting" and we'll come back to it later), but the variables are all undefined until they are assigned a value, which might not be what you want.

Q: Everyone seems to complain about the overuse of global variables in JavaScript. Why is this? Was the language badly designed or do people not know what they're doing, or what? And what do we do about it?

A: Globals are often overused in JavaScript. Some of this is because the language makes it easy to just jump in and start coding—and that's a good thing—because JavaScript doesn't enforce a lot of structure or overhead on you. The downside is when people write serious code this way and it has to be changed and maintained over the long term (and that pretty much describes all web pages). All that said, JavaScript is a powerful language that includes features like objects you can use to organize your code in a modular way. Many books have been written on that topic alone, and we're going to give you a taste of objects in Chapter 5 (which is only a couple of chapters away).



Actually, you can put your functions anywhere in your JavaScript file.

JavaScript doesn't care if your functions are declared before or after you use them. For instance, check out this code:

```
var radius = 5;
var area = circleArea(radius);
alert(area);

function circleArea(r) {
    var a = Math.PI * r * r;
    return a;
}
```

Notice that we're using the circleArea function before we're defining it!

The circleArea function isn't actually defined until after we've called it, in the code above. How on earth does this work?

This might seem really odd, especially if you remember when the browser loads your page, it starts executing the JavaScript from the top to the bottom of your file. But, the truth is JavaScript actually makes two passes over your page: in the first pass it reads all the function definitions, and in the second it begins executing your code. So, that allows you to place functions anywhere in your file.

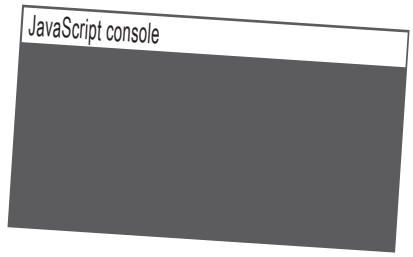
The Thing-A-Ma-Jig

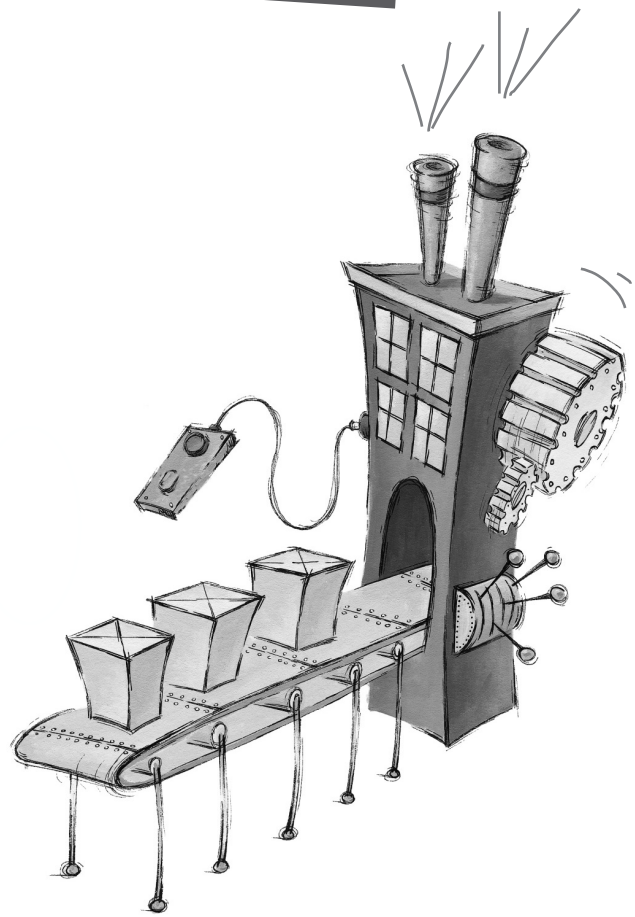
The Thing-A-Ma-Jig is quite a contraption—it clanks and clunks and even thunks, but what it really does, well, you've got us stumped. Coders claim they know how it works. Can you uncrack the code and find its quirks?

```
function clunk(times) {
    var num = times;
    while (num > 0) {
        display("clunk");
        num = num - 1;
    }
}

function thingamajig(size) {
    var facky = 1;
    clunkCounter = 0;
    if (size == 0) {
       display("clank");
    } else if (size == 1) {
       display("thunk");
    } else {
        while (size > 1) {
            facky = facky * size;
            size = size - 1;
        }
        clunk(facky);
    }
}

function display(output) {
    console.log(output);
    clunkCounter = clunkCounter + 1;
}
var clunkCounter = 0;
thingamajig(5);
console.log(clunkCounter);
```

We recommend passing the Thing-A-Ma-Jig the numbers 0, 1, 2, 3, 4, 5, etc. See if you know what it's doing.

Your output here!





Webville Guide to Code Hygiene

In Webville we like to keep things clean, organized and ready for expansion. There's no place that needs to be better maintained than your code, and JavaScript can seem pretty loosey-goosey when it comes to organizing your variables and functions. So we've put together a neat little guide for you that makes a few recommendations for those new to Webville. Take one, they're FREE.

Global variables, right at the TOP!

It's a good idea to keep your globals grouped together as much as possible, and if they're all up at the top, it's easy to find them. Now you don't have to do this, but isn't it easier for you and others to locate the variables used in your code if they're all at the top?

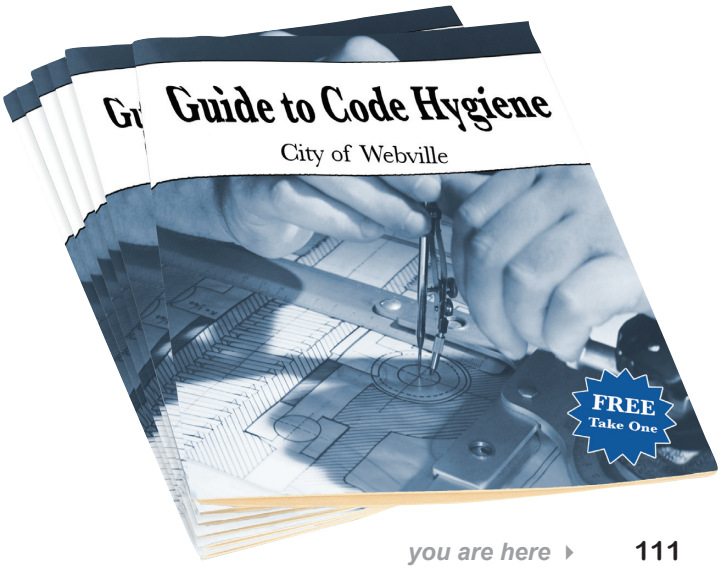
Functions like to sit together.

Well, not really; they actually don't care, they're just functions. But, if you keep your functions together, it's a lot easier to locate them. As you know, the browser actually scans your JavaScript for the functions before it does anything else. So you can place them at the top or bottom of the file, but if you keep them in one place your life will be easier. Here in Webville, we often start with our global variables and then put our functions next.

Let your local variables be declared at the TOP of the function they're in.

Put all your local variable declarations at the beginning of the function body. This makes them easy to find and ensures they are all declared properly before use.

That's it, just be safe and we hope you enjoy your time coding in Webville!



A bunch of JavaScript attendees, in full costume, are playing a party game, "Who am I?" They give you a clue, and you try to guess who they are, based on what they say. Assume they always tell the truth about themselves. Fill in the blank next to each sentence with the name of one attendee.

Tonight's attendees:

function, argument, return, scope, local variable, global variable, pass-by-value, parameter, function call, Math.random, built-in functions, code reuse.

I get passed into a function. _____

I send values back to the calling code. _____

I'm the all important keyword. _____

I'm what receives arguments. _____

It really means 'make a copy'. _____

I'm everywhere. _____

Another phrase for invoking a function. _____

Example of a function attached to an object. _____

alert and prompt are examples. _____

What functions are great for. _____

Where I can be seen. _____

I'm around when my function is. _____

The case of the attempted robbery not worth investigating

Sherlock finished his phone call with the bumbling chief of police, Lestrade, and sat down in front of the fireplace to resume reading the newspaper. Watson looked at him expectantly.

"What?" said Sherlock, not looking up from the paper.

"Well? What did Lestrade have to say?" Watson asked.

"Oh, he said they found a bit of rogue code in the bank account where the suspicious activity was taking place."

"And?" Watson said, trying to hide his frustration.

"Lestrade emailed me the code, and I told him it wasn't worth pursuing. The criminal made a fatal flaw and will never be able to actually steal the money," Sherlock said.

"How do you know?" Watson asked.

"It's obvious if you know where to look," Sherlock exclaimed. "Now stop bothering me with questions and let me finish this paper."

With Sherlock absorbed in the latest news, Watson snuck a peek at Sherlock's phone and pulled up Lestrade's email to look at the code.

```
var balance = 10500;
var cameraOn = true;

function steal(balance, amount) {
    cameraOn = false;
    if (amount < balance) {
        balance = balance - amount;
    }
    return amount;
    cameraOn = true;
}

var amount = steal(balance, 1250);
alert("Criminal: you stole " + amount + "!");
```

This is the real, actual bank balance in the account.

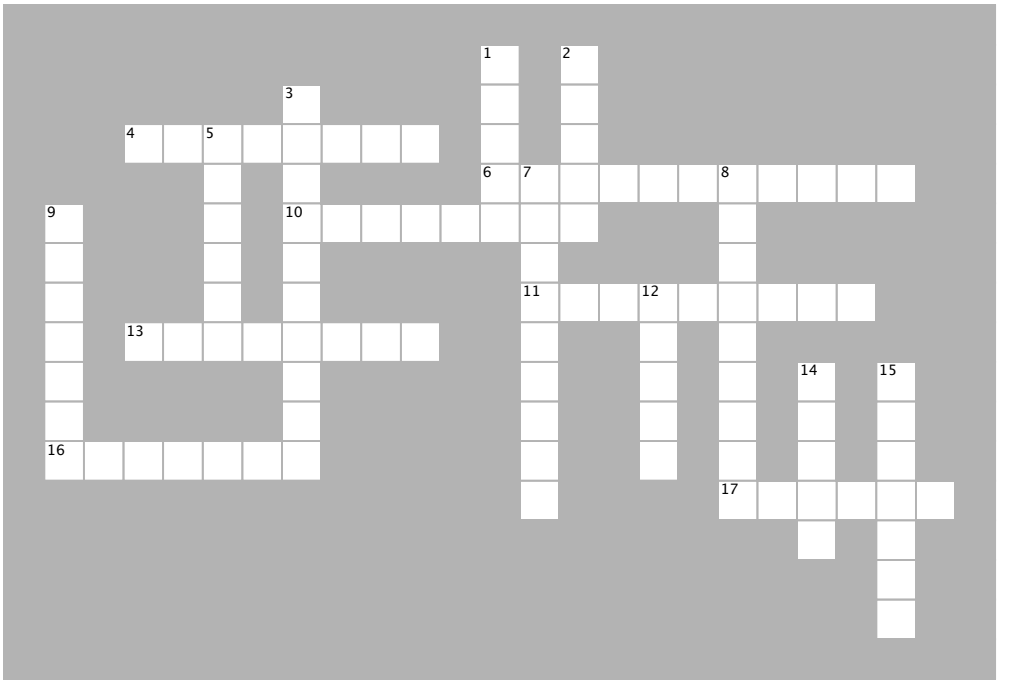
Why did Sherlock decide not to investigate the case? How could he know that the criminal would never be able to steal the money just by looking at the code? Is there one problem with the code? Or more?

BULLET POINTS

- Declare a function using the **function** keyword, followed by the name of the function.
- Use parentheses to enclose any **parameters** a function has. Use empty parentheses if there are no parameters.
- Enclose the **body** of the function in curly braces.
- The statements in the body of a function are executed when you call a function.
- **Calling** a function and **invoking** a function are the same thing.
- You call a function by using its name and passing arguments to the function's parameters (if any).
- A function can optionally return a value using the **return** statement.
- A function creates a local scope for parameters and any local variables the function uses.
- Variables are either in the **global scope** (visible everywhere in your program) or in the **local scope** (visible only in the function where they are declared).
- Declare local variables at the top of the body of your function.
- If you forget to declare a local variable using **var**, that variable will be global, which could have unintended consequences in your program.
- Functions are a good way to organize your code and create reusable chunks of code.
- You can customize the code in a function by passing in arguments to parameters (and using different arguments to get different results).
- Functions are also a good way to reduce or eliminate duplicate code.
- You can use JavaScript's many built-in functions, like alert, prompt, and Math.random, to do work in your programs.
- Using built-in functions means using existing code you don't have to write yourself.
- It's a good idea to organize your code so your functions are together, and your global variables are together, at the top of your JavaScript file.

JavaScript cross

In this chapter you got functional. Now use some brain functions to do this crossword.



ACROSS

4. A parameter acts like a _____ in the body of a function.

6. JavaScript uses _____ when passing arguments to functions.

10. You can declare your functions _____ in your JavaScript file.

11. What gets returned from a function without a return statement.

13. Local variables disappear when the _____ returns.

16. If you forget to declare your locals, they'll be treated like _____.

17. A local variable can _____ a global variable.

DOWN

1. A variable with global _____ is visible everywhere.

2. Use functions so you can _____ code over and over again.

3. The variables that arguments land in when they get passed to functions.

5. To get a value back from a function, use the _____ statement.

7. What gets passed to functions.

8. When you reload your page, all your _____ get re-initialized.

9. _____ through your code means following the execution line by line.

12. Watson looked at the bank heist code in Sherlock's _____ on his phone.

14. It's better to use _____ variables whenever you can.

15. Extra arguments to a function are _____.

Sharpen your pencil Solution

Do a little analysis of the code below. How does it look? Choose as many of the options below as you like, or write in your own analysis. Here's our solution.

```
var dogName = "rover";
var dogWeight = 23;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
dogName = "spot";
dogWeight = 13;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
dogName = "spike";
dogWeight = 53;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
dogName = "lady";
dogWeight = 17;
if (dogWeight > 20) {
   console.log(dogName + " says WOOF WOOF");
} else {
   console.log(dogName + " says woof woof");
}
```

We chose all of them!

- ☑ A. The code seems very redundant.
- ☑ B. If we wanted to change how this outputs, or if we wanted to add another weight for dogs, this is going to require a lot of reworking.
- ☑ C. Looks tedious to type in!
- ☑ D. Not the most readable code I've ever seen.
- ☑ E. Looks like the developer thought the weights might change over time.

Sharpen your pencil Solution

We've got some more calls to bark below. Next to each call, write what you think the output should be, or if you think the code will cause an error. Here's our solution.

```
bark("juno", 20);
```
juno says woof woof

```
bark("scottie", -1);
```
scottie says woof woof

Our bark function doesn't check to make sure dog weights are greater than 0. So this works because -1 is less than 20.

```
bark("dino", 0, 0);
```
dino says woof woof

The bark function just ignores the extra argument, 0. And using 0 as the weight doesn't make sense, but it still works.

```
bark("fido", "20");
```
fido says woof woof

We compare the string "20" to the number 20. "20" isn't greater than 20, so fido says "woof woof". (You'll find out later how JavaScript compares "20" and 20.)

```
bark("lady", 10);
```
lady says woof woof

```
bark("bruno", 21);
```
bruno says WOOF WOOF

Code Magnets Solution

This working JavaScript code is all scrambled up on the fridge. Can you reconstruct the code snippets to make a working program that produces the output listed below? Notice, there may be some extra code on the fridge, so you may not use all the magnets. Here's our solution.

```
function whatShallIWear ( temp ) {
    if (temp < 60) {
        console.log("Wear a jacket");
    }
    else if (temp < 70) {
        console.log("Wear a sweater");
    }
    else {
        console.log("Wear t-shirt");
    }
}
whatShallIWear(50);
whatShallIWear(80);
whatShallIWear(60);
```

Leftover magnets.

```
,
,
,
}
}
temperature
```

JavaScript console

```
Wear a jacket
Wear a t-shirt
Wear a sweater
```

Below you'll find some JavaScript code, including variables, function definitions and function calls. Your job is to identify all the variables, functions, arguments and parameters. Write the names of each in the appropriate boxes on the right. Here's our solution.

```
function dogYears(dogName, age) {
    var years = age * 7;
    console.log(dogName + " is " + years + " years old");
}
var myDog = "Fido";
dogYears(myDog, 4);

function makeTea(cups, tea) {
    console.log("Brewing " + cups + " cups of " + tea);
}
var guests = 3;
makeTea(guests, "Earl Grey");

function secret() {
    console.log("The secret of life is 42");
}
secret();

function speak(kind) {
    var defaultSound = "";
    if (kind == "dog") {
        alert("Woof");
    } else if (kind == "cat") {
        alert("Meow");
    } else {
        alert(defaultSound);
    }
}
var pet = prompt("Enter a type of pet: ");
speak(pet);
```

Variables

myDog, guests, pet, years, defaultSound

Functions

dogYears, makeTea, secret, speak,

Built-in functions

alert, console.log, prompt

Arguments

myDog, 4, guests, "Earl Grey", pet, plus all the string arguments to alert and console.log

Parameters

dogName, age, cups, tea, kind

Below you'll find some JavaScript code, including variables, function definitions and function calls. Your job is to identify the variables used in all the arguments, parameters, local variables and global variables. Write the variable names in the appropriate boxes on the right. Then circle any variables that are shadowed. Here's our solution.

```
var x = 32;
var y = 44;
var radius = 5;

var centerX = 0;
var centerY = 0;
var width = 600;
var height = 400;

function setup(width, height) {
    centerX = width/2;
    centerY = height/2;
}

function computeDistance(x1, y1, x2, y2) {
    var dx = x1 - x2;
    var dy = y1 - y2;
    var d2 = (dx * dx) + (dy * dy);
    var d = Math.sqrt(d2);
    return d;
}

function circleArea(r) {
    var area = Math.PI * r * r;
    return area;
}

setup(width, height);
var area = circleArea(radius);
var distance = computeDistance(x, y, centerX, centerY);
alert("Area: " + area);
alert("Distance: " + distance);
```

Arguments

width, height, radius,
x, y, centerX,
centerY,
"Area: " + area,
"Distance: "+ distance

Don't forget the arguments to the alert function.

Parameters

width, height,
x1, y1, x2, y2, r

Locals

dx, dy, d2, d,
area

The local variable area shadows the global variable area

Globals

x, y, radius,
centerX, centerY,
width, height, area,
distance

Don't forget area and distance. These are globals too.

The Thing-A-Ma-Jig

The Thing-A-Ma-Jig is quite a contraption—it clanks and clunks and even thunks, but what it really does, well, you've got us stumped. Coders claim they know how it works. Can you uncrack the code and find its quirks?

Here's our solution:

```
JavaScript console
(120) clunk
120
```

If you pass 5 to thingamajig, you'll see "clunk" in the console 120 times (or you might see your console abbreviate 120 clunks, like above), and then the number 120 at the end.

What about the other values?

```
JavaScript console
clank
1
JavaScript console
thunk
1
JavaScript console
(2) clunk
2
JavaScript console
(6) clunk
6
JavaScript console
(24) clunk
24
JavaScript console
(120) clunk
120
```

Thing-A-Ma-Jig of 0

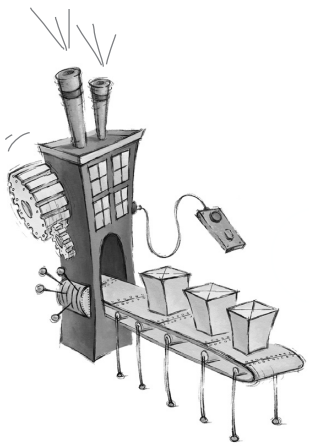
Thing-A-Ma-Jig of 1

Thing-A-Ma-Jig of 2

Thing-A-Ma-Jig of 3

Thing-A-Ma-Jig of 4

Thing-A-Ma-Jig of 5



What does it all mean? We hear the Thing-A-Ma-Jig was invented by a curious chap who was fascinated by rearranging words. You know like DOG rearranged is GOD, OGD, DGO, GDO and ODG. So if a word has three letters the Thing-A-Ma-Jig says you can make six total combinations from those letters. If you use the word "mixes" you can make 120 combinations of letters, wow! Anyway, that's what we heard. And here we just thought it computed mathematical factorials! Who knew!?

Check out 'factorial' in wikipedia for more info!

Five Minute Mystery Solution

Why did Sherlock decide not to investigate the case? How could he know that the criminal would never be able to steal the money just by looking at the code? Is there one problem with the code? Or more? Here's our solution.

```
var balance = 10500;
var cameraOn = true;

function steal(balance, amount) {
    cameraOn = false;
    if (amount < balance) {
        balance = balance - amount;
    }
    return amount;
    cameraOn = true;
}

var amount = steal(balance, 1250);
alert("Criminal: you stole " + amount + "!");
```

balance is a global variable...

... but it's shadowed by this parameter.

So when you change the balance in the function steal, you're not changing the actual bank balance!

We're returning the amount stolen...

... but we're not using it to update the real balance in the account. So the balance of the bank account is the same as it was originally.

The criminal thinks he stole the money, but he didn't!

And, in addition to not actually stealing any money, the criminal forgets to turn the camera back on, which is a dead giveaway to the police that something nefarious is going on. Remember, when you return from a function, the function stops executing, so any lines of code after the return are ignored!

Who am I? Solution

A bunch of JavaScript attendees, in full costume, are playing a party game, "Who am I?" They give you a clue, and you try to guess who they are, based on what they say. Assume they always tell the truth about themselves. Fill in the blank next to each sentence with the name of one attendee. Here's our solution.

Tonight's attendees:

function, argument, return, scope, local variable, global variable, pass-by-value, parameter, function call, Math.random, built-in functions, code reuse.

I get passed into a function.	argument
I send values back to the calling code.	return
I'm the all important keyword.	function
I'm what receives arguments.	parameter
It really means 'make a copy'.	pass-by-value
I'm everywhere.	global variable
Another phrase for invoking a function.	function call
Example of a function attached to an object.	Math.random
alert and prompt are examples.	built-in functions
What functions are great for.	code reuse
Where I can be seen.	scope
I'm around when my function is.	local variable

JavaScript cross Solution