



LEARNING JavaScript®

A Hands-On Guide to the Fundamentals of Modern JavaScript



Praise for *Learning JavaScript*

“Between modern web interfaces, server side technologies, and HTML5 games, JavaScript has never been a more important or versatile tool. To anyone just starting out with JavaScript or looking to deepen their knowledge of the practical core of the language, I would highly recommend *Learning JavaScript*.”

—Evan Burchard, Independent Web Developer

“Although I’ve read a couple of books about JavaScript before, as a backend developer, I was thrilled to see Tim Wright’s *Learning JavaScript*. The nuances of progressive enhancement versus graceful degradation are finally explained in a manner that someone new to front-end coding can understand. Bravo, Tim.”

—Joe Devon, Cofounder, StartupDevs.com

“Tim Wright has written a delightfully practical book for the novice front-end developer who wants to learn JavaScript. This book’s strength is in providing a good introduction to JavaScript while also illustrating the context of when and where it should be used.”

—R. S. Doiel, Senior Software Engineer, USC Web Services

“*Learning JavaScript* is a great introduction into modern JavaScript development. From covering the history to its exciting future, *Learning JavaScript* equips the novice developer to practical application in the workforce. I wish this book came along when I was a novice!”

—Hillisha Haygood, Senior Web Developer, Sporting News

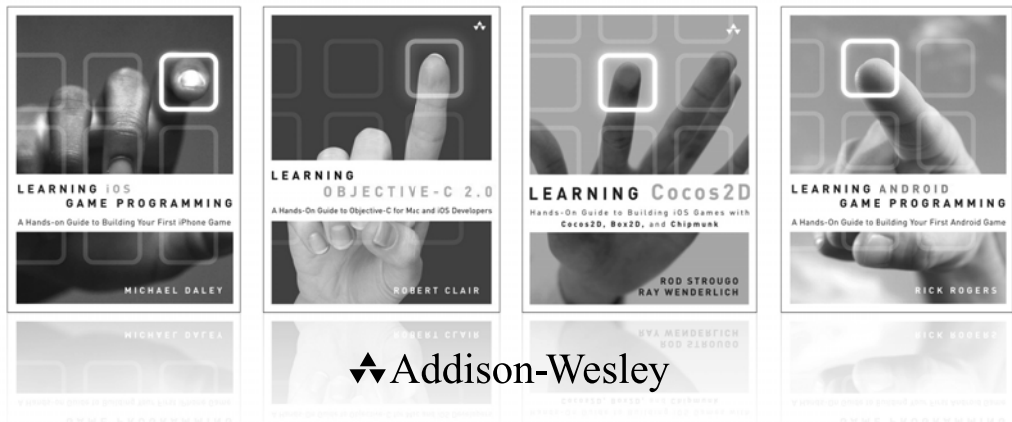
“Tim presents invaluable techniques for writing JavaScript with progressive enhancement at the forefront. If you are new to JavaScript then this book will prove to be a great asset in your learning. Covering all the basics and then right through to touch events, AJAX, and HTML5 APIs, the examples are clear and easy to follow. Using this book, you will learn when and how to use JavaScript to great effect.”

—Tom Leadbetter, Freelance Web Designer

“*Learning JavaScript* is valuable for both new and veteran developers. It is great for new developers because it is easy to read and provides a step-by-step process to becoming great at JavaScript. Veteran developers will be reminded of many of the best practices they have already forgotten.”

—Christopher Swenor, Manager of Technology, zMags

Addison-Wesley Learning Series



Visit informit.com/learningseries for a complete list of available publications.

The Addison-Wesley Learning Series is a collection of hands-on programming guides that help you quickly learn a new technology or language so you can apply what you've learned right away.

Each title comes with sample code for the application or applications built in the text. This code is fully annotated and can be reused in your own projects with no strings attached. Many chapters end with a series of exercises to encourage you to reexamine what you have just learned, and to tweak or adjust the code as a way of learning.

Titles in this series take a simple approach: they get you going right away and leave you with the ability to walk off and build your own application and apply the language or technology to whatever you are working on.

 Addison-Wesley

informIT.com

 Safari
Books Online

Learning JavaScript

This page intentionally left blank

Learning JavaScript

A Hands-On Guide
to the Fundamentals
of Modern JavaScript

Tim Wright

◆◆ Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication data

Wright, Tim, 1982-

Learning JavaScript : a hands-on guide to the fundamentals of modern JavaScript /
Tim Wright.
pages cm

Includes bibliographical references and index.

ISBN 978-0-321-83274-0 (pbk. : alk. paper) – ISBN 0-321-83274-4 (pbk. : alk. paper)

1. JavaScript (Computer program language)—Handbooks, manuals, etc. I. Title.

QA76.73.J38W755 2013

005.2'762—dc23

2012019351

Copyright © 2013 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-321-83274-0

ISBN-10: 0-321-83274-4

Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan.

First printing, August 2012

Editor-in-Chief
Mark Taub

Acquisitions Editor
Laura Lewin

Development Editor
Songlin Qiu

Managing Editor
Kristy Hart

Project Editor
Anne Goebel

Copy Editor
Barbara Hacha

Indexer
Lisa Stumpf

Proofreader
Debbie Williams

Technical Reviewers
Evan Burchard
Alex Moffat

Publishing Coordinator
Olivia Basegio

Cover Designer
Chuti Prasertsith

Compositor
Nonie Ratcliff



For Ma.



Contents

Introduction	1
Chapter 1: Progressive Enhancement	3
Chapter 2: JavaScript in the Browser	21
Chapter 3: JavaScript Terminology	39
Chapter 4: Accessing the DOM	57
Chapter 5: Storing Data in JavaScript	81
Chapter 6: Variables, Functions, and Loops	103
Chapter 7: Interacting with the User Through Events	123
Chapter 8: Communicating with the Server Through Ajax	149
Chapter 9: Code Organization	179
Chapter 10: Making JavaScript Easier with Libraries	211
Chapter 11: HTML5 JavaScript APIs	243
Chapter 12: Moving Forward with JavaScript	273
Answers	305
Index	309

Table of Contents

Introduction 1

Chapter 1: Progressive Enhancement 3

Defining Progressive Enhancement 3

History 4

Purpose 5

Accessibility 5

Reusability 5

Progressive Enhancement Versus Graceful Degradation 6

Structure Layer 6

Adding More Meaning with HTML5 8

Presentation Layer 9

Inline CSS 10

Linking Up Your Stylesheet 10

Behavior Layer 12

Inline JavaScript 12

Embedded JavaScript 13

External and Unobtrusive JavaScript 15

Benefits of Progressive Enhancement 16

Performance 17

Building for the Future 17

The Touch Interface 18

Final Words on Progressive Enhancement 19

Summary 20

Exercises 20

Chapter 2: JavaScript in the Browser 21

A People's History of JavaScript 21

Origins 22

Progressive Enhancement 23

The Behavior Layer 24

Moving Past Today 24

Browser Interactions with JavaScript 25

HTTP Requests 26

JavaScript and Rendering Engines 29

- What JavaScript Can Do 30
 - Modifying HTML 31
 - Communicating with the Server 31
 - Storing Data 31
- How You Should Use JavaScript 32
 - Improving User Experience 32
 - Using JavaScript Responsibly 32
 - Creating Fallbacks 34
- Tools to Help You Use JavaScript 36
 - Tools Built into the Language 36
 - Tools Built into the Browser 37
- Summary 38
- Exercises 38

Chapter 3: JavaScript Terminology 39

- Basics 39
 - Document Object Model (DOM) 39
 - Parents 40
 - Children 40
 - Siblings 41
 - Variables 41
 - Strings 43
 - Comments 43
 - Operators 44
 - Use Strict 45
- Storage 45
 - Cache 45
 - Arrays 45
 - Cookies 46
 - JavaScript Object Notation (JSON) 46
 - Objects 47
- Creating Interaction 47
 - Loops 48
 - Conditionals 48
 - switch Statement 49
 - Functions 50
 - Anonymous Functions 51

Callback Functions	52
Methods	53
Events	54
Ajax	54
Summary	55
Exercises	55

Chapter 4: Accessing the DOM 57

What Is the DOM?	57
The DOM Tree	58
Element Nodes	59
Text Nodes	60
Attribute Nodes	62
Working with the Element Node	62
Targeting by ID	63
Targeting by Tag Name	64
Targeting by Class	67
Using CSS Selectors in JavaScript to Target Nodes	68
Working with the Attribute Node	70
Getting an Attribute	71
Setting an Attribute	72
Removing an Attribute	73
Working with the Text Node and Changing Content	73
Moving Around the DOM	74
Accessing First and Last Child	76
Dynamically Adding and Removing Nodes from the DOM	77
Adding Elements to the DOM	77
Removing Elements from the DOM	78
Summary	79
Exercises	79

Chapter 5: Storing Data in JavaScript 81

Variables	81
Strings	82
Numbers	83
Boolean	84
Performance in Variables	84

- Arrays 85
 - Basic Array 85
 - Associative Array 87
 - Multidimensional Array 87
 - Pushing Data into an Array 89
 - Working with Array Methods 89
- join 90
- slice 90
- shift and unshift 91
- pop 92
- concat 92
- sort 93
- Objects 93
 - Performance in Objects 94
- JSON 95
 - Benefits of Using JSON 96
 - Using an API 96
- Web Storage in HTML5 97
 - localStorage and sessionStorage 97
- setItem 97
- getItem 98
- removeItem 98
 - Storing Chunks of Data with JSON 99
 - Using Web Storage Responsibly 100
- Summary 101
- Exercises 101

Chapter 6: Variables, Functions, and Loops 103

- Defining Variables 103
 - Grouping Variables 104
 - Reserved Terms 104
- Functions 105
 - Basic Functions 106
 - Anonymous Functions 107
- Scope 108
 - Calling a Function with a Function 109
 - Returning Data 110
 - A Function as a Method 112

Loops	113
for Loop	114
Conditionals	116
if Statement	116
if/else Statement	117
switch Statement	118
if versus switch	119
Putting It All Together	120
Summary	121
Exercises	122

Chapter 7: Interacting with the User Through Events 123

Attaching an Event	124
Event Handlers	124
Event Listeners	125
Binding Events	128
Unbinding Events	129
Mouse and Keyboard Events	130
click	132
focus and blur	134
Accessibility	135
change	135
mouseover and mouseout (hovering)	136
submit	137
Preventing Default Behavior	139
keydown, keypress, and keyup	139
Putting It All Together	140
Touch and Orientation Events	143
touchstart and touchend	144
touchmove	145
orientationchange	145
Support for Touch Events	146
Putting It All Together	147
Summary	148
Exercises	148

Chapter 8: Communicating with the Server Through Ajax 149

- Ajax History 150
 - Server Communication 151
 - The XMLHttpRequest 152
- Creating an Ajax Call 154
 - Sending a Request to the Server 155
 - Receiving Data Back from the Server 158
 - Making Repeat Ajax Calls 163
- Ajax Data Formats 164
 - XML 165
 - HTML 166
 - JSON 167
- Ajax Accessibility 168
 - Live Regions and ARIA 169
- Common Ajax Mistakes 170
 - Providing Feedback 170
- Putting It All Together 172
- Where Is Ajax Going? 177
- Summary 177
- Exercises 178

Chapter 9: Code Organization 179

- General Coding Style Rules 180
 - Scope 181
 - Failing Quickly 183
 - User Experience 185
- Code Design 185
 - Files and Directories 186
 - In-document Script 187
 - Variable Declarations 188
 - Variable and Function Naming 189
 - Comments 190
 - Indentation 192
 - Whitespace 193
 - Statement Spacing 194

Line Breaks	195
Math and Operators	196
Using eval()	197
Taking Style Guides Too Far	199
Code Structure	200
Functions	200
Anonymous Functions	201
Functions as Variables	202
Functions as Methods	202
JavaScript Development Patterns	204
Summary	208
Exercises	209

Chapter 10: Making JavaScript Easier with Libraries 211

JavaScript Library Basics	212
The Library Learning Process	213
Syntax	214
Focusing on the Goal	214
Creating Shortcuts	215
Fixing Browser Issues	216
Popular Libraries	216
jQuery Basics	221
document.ready	222
Selectors	223
Traveling Through the DOM	225
Adding Style Information	226
Binding Events	227
Animation	227
jQuery Nonbasics	228
Using Ajax in jQuery	228
Looping Through Data in jQuery	230
Chaining Functions	232
Extending Libraries Through Plug-ins	233
Building a Plug-in	234
The Good of Libraries	236
Popularity and Community	236
Efficient Code	237

- The Bad of Libraries 238
 - Overhead 238
 - Performance 239
 - Overreliance and Shelf Life 239
- Using Microlibraries 240
 - The Good 240
 - The Bad 241
- Summary 242
- Exercises 242

Chapter 11: HTML5 JavaScript APIs 243

- What Is HTML5? 244
- The Markup (aka HTML) 244
 - Creating Better Semantics 245
 - Building More Accessible Content 245
- The JavaScript APIs 248
 - The navigator Object 248
 - Geolocation 249
 - Audio and Video 251
 - History API 254
 - Web Workers 259
- Device API 265
 - The Battery Status API 266
 - The Vibration API 267
 - The Network Information API 268
- Using This Today with Feature Detection 270
- Summary 271
- Exercises 272

Chapter 12: Moving Forward with JavaScript 273

- A Brief Review of Key Topics 274
 - Progressive Enhancement 274
 - DOM Manipulation 275
 - Data Storage 277
 - Server Communication 279

JavaScript for Designers	279
Advanced Interface Design	280
CSS Transforms in JavaScript	284
Interacting from the Desktop	289
JavaScript for Developers	293
JavaScript Templates	294
JavaScript on the Server with NodeJS	299
Summary	302
Exercises	303

Answers 305

Index 309

This page intentionally left blank

Acknowledgments

There are a lot of people who contributed in some way to the completion of this book. First of all, I want to thank the folks at Pearson for giving me the opportunity to not only write this book, but structure it in a way that truly reflects how I believe the topic should be taught. The book would not have stayed on track without them. My technical editors were also instrumental to the process in pointing out any missteps, giving praise when needed, and making sure every detail of the book was written with accuracy and precision; I could not have done it without you (Evan Burchard and Alex Moffat). I would also like to give special thanks to my parents, friends, and family for the continued support, encouragement, and patience throughout this long process and for pulling me out of my “writing cave” for fresh air every once in a while. Without you all, nothing would have been possible.

About the Author

Tim Wright has been a Web designer and front-end developer since 2004, primarily focusing on CSS, HTML5, accessibility, user experience, and building applications with the capability to scale seamlessly from desktop to mobile device. He has worked at various universities nationwide and fostered the advancement of Web standards at each stop along the way. Tim has written many articles for popular Web design online publications, such as *Smashing Magazine*, *SitePoint*, and *Web Designer Depot*, on all facets of front-end development from HTML5 and CSS3 to user experience and advanced JavaScript techniques. He also writes many articles via his personal blog at csskarma.com. Tim holds a Bachelor's Degree in Marketing Management from Virginia Tech, with a specialization in Graphic Design.

Introduction

When I decided to write a book about JavaScript, I wanted to create it in a way that felt natural to how I learned the language. I didn't learn it from school or a book; my JavaScript knowledge comes from real-world application, trial and error, and self-motivation. I wanted to present the information in a unique way so that you could get up to speed quickly, but still develop a solid base for the language and move forward without feeling overwhelmed with too much information. I combined my teaching experience with how I felt while I was learning to create an environment that moves quickly but has built-in break points and reviews to always keep the mind focused and clear. The JavaScript language can be confusing if taken all at once. There are hundreds of ways to accomplish the same task, most of which you don't need to know. I did my best throughout this book to not show too many ways to do the same thing, but rather focus on doing one thing really well.

The organization of this book is a little different from that of a normal JavaScript book. Often terms are introduced, explained in real-time, and readers can feel like they are taking in too much information at once. This can cause a loss of focus on the main task at hand. I addressed this issue by putting all the common JavaScript terms right up front in the book instead of piling them in a glossary that no one will read. As you go through them, they provide brief explanations of many core concepts in the language. This way we don't have to spend valuable time giving broad definitions of miscellaneous terms and can focus on getting you the most knowledge out of this short time we have together.

The process of learning a robust language like JavaScript may seem intimidating at first, but don't worry, it's not that bad. After you grasp some of the basic ideas, the rest is like learning a spoken language; the hard part is properly organizing it, performance tuning, and most of all, knowing when to use CSS instead. Hopefully, by the time you're finished reading this book, you will have gained the knowledge you need to effectively create a better user experience by responsibly using JavaScript.

JavaScript is a language with an amazingly rich history and an even brighter future. Throughout this book you learn the basics of the language, but at the same time you learn more advanced topics, such as HTML5 JavaScript APIs and how you create a touch-enabled interface. You can be assured that even though JavaScript is code, it's far from boring; you can create some pretty wild interfaces and have a lot of fun in the process.

I hope this book can serve you well for years to come and will act as a launching pad for your continued interest in JavaScript. If this is the first step in your journey to learning JavaScript, welcome aboard; if you already know the language, welcome back.

Target Audience for This Book

The audience for this book is anyone starting out in Web design and development who wants to learn about JavaScript. Before reading this book, you should be knowledgeable in HTML and CSS, and be familiar with the concepts behind progressive enhancement.

This book can equally serve absolute beginners and seasoned Web veterans who are expanding their knowledge into JavaScript. All the while, I hope it instills enthusiasm to learn more about this rapidly moving industry.

Code Samples for This Book

The code samples for this book are available on the book's website at <http://learningjsbook.com>.

Variables, Functions, and Loops

This is one of the more important chapters in the book because you learn some of the core features in JavaScript. We expand on the variables that were mentioned in the previous chapter, then move on to creating functions, and last, we go over how to loop through data to autoexecute the same code block over and over. Using variables, functions, and loops are often the only thing a person knows how to do in JavaScript, and they usually get along just fine. You're already past that part and on your way to becoming an elite JavaScript developer, so no worries there. You'll be coding while all the others are looking up how to do something.

Now that you have a solid base in how to work with a lot of the common things in JavaScript, you can start building an application and producing something tangible. Up to this point in the book, the examples have been pretty specific, but also a little abstract. You've been manipulating content and data, then alerting or observing the result. In this chapter we expand on what you've learned already and begin building a simple JavaScript application that will get more robust as you step through the subsequent chapters.

As you progress through this chapter, you notice that an address book application should be starting to form. Some of the methods that we go over repeat in their core functionality but have very different use-cases. Although they may not necessarily all live in the same application, this is the chapter where you start building that tangible knowledge that can be directly transferred into a project.

Defining Variables

For the most part, you learned about variables within the context of data storage, but they also have an integral part in your application when it comes to functionality.

When considering variable and function naming, it's best to make them meaningful and speak to their contents or purpose. For example, using a variable name of "myBestFriend" would be

much more helpful than something like, “`firstVariableName`.” Something else to consider when naming variables is that they can’t start with a number. They can *contain* numbers, such as “`dogs3`” or “`catsStink4Eva`,” but they can’t begin with a number, such as “`3dogs`.”

Grouping Variables

When you’re writing an application, it’s best to try to group all variables at the top of your JavaScript file or function (when possible) so they can all be immediately cached for later reference. Some people find this method a little unnatural because functions are defined throughout the document, and it’s a little easier to maintain when variables are right there with the function they belong to; but grouping variables at the top is one of those small performance boosts you can give to your application. It helps to think of it as one large file containing JavaScript for an application versus thinking of the file as a collection of one-off actions that get executed. When thinking of it as a single unit, it feels a little better (to me) when I’m grouping all variables together at the top.

You can group variables in your document in two ways. Up to this point we have been using a new `var` declaration for each variable; a lot of people prefer this method, and it’s perfectly fine to use. An alternative method is to use a single `var` declaration, using commas to separate the individual variables and a semicolon at the very end. Listing 6.1 shows an example of grouping variables with a single `var` declaration. Note the commas at the end of each line.

Listing 6.1 Grouping Variables with a Single `var` Declaration

```
var highSchool = "Hill",  
    college = "Paul",  
    gradSchool = "Vishaal";
```

There’s no difference in the way you access these variables compared to how you access variables declared with individual `var` declarations. At the variable level, it’s purely a way to group. It isn’t good or bad at this point—it’s only personal preference. You’ll see both methods in looking through JavaScript others have written, so it’s good to know what’s going on.

You see this style of variable declaration a lot more when getting into objects, methods, and grouping functions together. I prefer it because it feels cleaner and a little more consistent, but as you progress you will settle on a preference of your own. Both are certainly valid methods.

Reserved Terms

JavaScript contains a lot of core functionality. We’ve been over quite a bit of it so far. Beyond that core functionality you will be defining a lot of your own custom code. If the names of your custom JavaScript match up with anything built into the language, it can cause collisions and throw errors. It’s the same as if you’re writing a large JavaScript file—you want to make sure all the function and variable names are as unique as possible to prevent problems and

confusion while parsing the information. If you have two functions with the same name, it's difficult to tell the browser which one to use, so it's just not allowed.

To prevent these issues with native JavaScript, there are some reserved words (keywords) that you can't use when defining variables, functions, methods, or identifiers within your code.

Following is a list of the reserved words:

- `break`
- `case`
- `catch`
- `continue`
- `debugger`
- `default`
- `delete`
- `do`
- `else`
- `finally`
- `for`
- `function`
- `if`
- `implements`
- `in`
- `instanceof`
- `interface`
- `new`
- `package`
- `private`
- `protected`
- `public`
- `return`
- `static`
- `switch`
- `this`
- `throw`
- `try`
- `typeof`
- `var`
- `void`
- `while`
- `with`

Most of these are no-brainers, like `function` and `var`, and under normal circumstances you probably would never come across a situation where something like “implements” would be a reasonable name for a variable or function. If you end up using any of these terms in your code, the console will throw an error and let you know that you're using a reserved word. With that in mind, I think the value in this list is not so much memorizing it, but rather recognizing that these words map to native actions in the language. It will help you write better code and also aid in learning more advanced JavaScript down the road if you choose to research some of those terms that are beyond the scope of this book, such as `public`, `private`, and `protected`.

Functions

Functions in any programming language are ways to write code that can be used later. At its most basic form, this is also true for JavaScript. You can write a chunk of custom code and not

only execute it at will, but you can also execute it over and over, which can help streamline your application by increasing its maintainability (declaring a chunk of code one time and referencing it, rather than rewriting what it does). It's like keeping all your CSS in the same file or why you keep all JavaScript in the same file—you know exactly where it is when you need to change or add something.

You've been using functions already in earlier chapters when you pass data into an `alert()`. "Alert" is technically called a `method` but for all intents and purposes, it's the same as a `function`.

Basic Functions

The chance of creating a JavaScript application without having to write your own functions is pretty low. It's something that you'll be doing on every project, and it's very easy to do using the `function` keyword (remember the reserved words list? This is what `function` is for).

Using the `function` keyword is like saying, "Hey, I'm building something over here that should be treated as a function." Listing 6.2 shows a basic function declaration.

Listing 6.2 Writing a Basic Function

```
function sayHello() {  
  
    alert("hey there! ");  
  
}
```

Calling a Function

Calling a function is very simple. You type out the name, and then parentheses and the function will be executed. The parentheses tell the browser that you want to execute the function and to use any data (arguments) contained within the parentheses within the function. Listing 6.2.1 shows how to call the function we declared in Listing 6.2. It should alert the text, "hey there!"

Listing 6.2.1 Calling a Basic Function

```
sayHello(); // hey there
```

Arguments

Arguments are a way to pass information or data into a function. As previously mentioned, up to this point you've been using the `alert()` method. We've also been passing it arguments. The `alert` method is designed in native JavaScript to take arguments and display them in the form of a pop-up box in the browser.

Functions can take any number of arguments. They can be any type of information; strings, variables, large data sets, and anything else you can think of can be passed into a function through an argument. As you're defining your functions, you will be assigning names to the arguments, sort of like the way you assign names to a variable. After that argument is named in the function, it becomes a variable you'll be using inside that function.

In Listing 6.2.2 you can see that the `sayHello()` function now has a single argument called "message." Inside, the function "message" is used as a variable that gets passed into the JavaScript `alert()` method.

Listing 6.2.2 Passing a Function Variable Through Arguments

```
/* declare the function */
function sayHello(message){

    alert(message); // "message" is also an argument in the "alert" method

}

/* call it a couple times with different messages */
sayHello("Hey there, you stink!");

sayHello("I feel bad I just said that.");
```

When this function is called, we're setting the string argument to "Hey there, you stink!" and then quickly apologizing with another alert, because frankly it was kind of rude. This is a very real-life way arguments are used in functions. The string can either be declared upon calling the function (like we're doing in Listing 6.2.2) or it can be declared immediately in the function declaration. (Instead of using the message variable, you could insert the string.) Calling it the way we did is much more common in the real world, though.

Anonymous Functions

Anonymous functions are functions that have no name (obviously—they're anonymous). They execute immediately and can contain any number of other functions. The syntax for declaring an anonymous function is a little different. They are dynamic in nature because they are executed at runtime rather than waiting to be called.

Anonymous functions perform very well in the browser because there is no reference to them in another part of the document. This comes with pluses and minuses. So as you write your JavaScript, it is always good to note that if you have to rewrite an anonymous function over and over, it's probably best to pull it out into a normal function to cut down on maintenance and repetitive code.

There is often a little confusion as to the purpose of anonymous functions. If you want something to execute at runtime, why wouldn't you just dump the code right into your JavaScript

file? Why even bother wrapping it in an anonymous function? Well, this is a good place to bring up a term you may hear a lot: **scope**.

Scope

Scope is a programming concept that exists to reduce the amount of variable and function collisions in your code. It controls how far information can travel throughout your JavaScript document. Earlier on, we briefly mentioned `global variables`. “Global” is a type of scope; the global scope for a variable means that the variable can be accessed and used anywhere in the document. Global variables are generally a bad thing, especially in larger files where naming collisions are more likely. Try to keep things out of the global scope if possible. Listing 6.3 shows how to declare a basic anonymous function and keep variables out of the global scope.

Listing 6.3 Defining an Anonymous Function

```
/* set up your anonymous function */
(function () {

    /* define a variable inside the function */
    var greeting = "Hello Tim";

    /* access the variable inside the function */
    alert("in scope: " + greeting);

})(); // end anonymous function
```

For the most part, you will be dealing in function-level scope. This means that any variable defined inside a function cannot be used outside that function. This is a great benefit of using anonymous functions. If you wrap a code block in an anonymous function, the contents of that function, which would normally default to the global scope, will now be contained within the scope of that anonymous function.

Listing 6.3.1 defines a variable inside an anonymous function, alerts the variable, and then tries to alert the variable again, outside the function (it won't end well).

Listing 6.3.1 Showing Scope Inside an Anonymous Function

```
/* set up your anonymous function */
(function () {

    /* define a variable inside the function */
    var greeting = "Hello Tim";
```

```

    /* access the variable inside the function */
    alert("in scope: " + greeting);

})(); // end anonymous function

/* try and access that variable outside the function scope */
alert("out of scope: " + typeof(greeting)); // alerts "undefined"

```

As you can see, the variable `alert` is `undefined`, even though you can see it's clearly defined within the anonymous function. This is because the function scope will not allow the variable to leave the function.

Note

In the second alert of Listing 6.3.1 we're using the JavaScript method `typeof()`, which alerts the variable type "undefined." If we didn't do this, the file would throw an error, and you wouldn't see the second alert at all. The JavaScript console would display the error, "greeting is undefined."

Calling a Function with a Function

When you have a function that calls another function, the second function is referred to as a **callback**. The callback function is defined as a normal function with all the others but is executed inside another function. They're a little different because instead of *you* having to do something to execute the function, another function does something. It's like having robots that are built by other robots—total madness, I know.

Callback functions are a great way to separate out the levels of functionality in your code and make parts more reusable. Often you will see callback functions passed as arguments to other functions. We'll get more into that in the next chapter when we talk about JavaScript events, and they're especially important when dealing with server communications like Ajax. Listing 6.3.2 shows our `sayHello()` function being defined and then called inside the anonymous function. In this case, `sayHello()` is a callback function (calling it twice).

Listing 6.3.2 Using a Callback Function

```

function sayHello(message) {
    alert(message);
}

(function (){

    var greeting = "Welcome",
        exitStatement = "ok, please leave.";

```

```
    sayHello(greeting);  
    sayHello(exitStatement);  
  
})();
```

Returning Data

Every function you create will not result in a direct output. Up to this point you've been creating functions that do something tangible, usually alerting a piece of data into the browser. You won't always want to do that, though; from time to time you will want to create a function that returns information for another function to use. This will make your functions a little smaller, and if the function that gathers information is general enough, you can reuse it to pass the same (or different) information into multiple functions.

Being able to return data and pass it into another function is a powerful feature of JavaScript.

Returning a Single Value

Going back to the `sayHello()` function that was defined in Listing 6.2, we're going to remove the `alert()` action that was previously being executed when the function was called, and we'll replace it with a `return` statement. This is depicted in Listing 6.3.3.

Listing 6.3.3 Returning Data with a Function

```
function sayHello(message){  
    return message + "!"; // add some emotion too  
}
```

You'll probably notice that the `sayHello()` function doesn't do anything in the browser anymore. That's a good thing (unless you're getting an error—that's a bad thing). It means the function is now returning the data but it's just sitting there waiting to be used by another function.

Returning Multiple Values

Sometimes returning a single value isn't enough for what you're trying to accomplish. In that case you can return multiple values and pass them in an array format to other functions. Remember how I mentioned that arrays are really important? They creep up a lot when dealing in data storage and flow in JavaScript. In Listing 6.3.4 you can see the `sayHello()` function taking two arguments. Those arguments get changed slightly and are resaved to variables; then they are returned in an array format to be accessed later.

Listing 6.3.4 Returning Multiple Data Values with a Function

```
function sayHello(greeting, exitStatement){

    /* add some passion to these dry arguments */
    var newGreeting = greeting + "!",
        newExitStatement = exitStatement + "!!";

    /* return the arguments in an array */
    return [newGreeting, newExitStatement];

}
```

Passing Returned Values to Another Function

Now that you're returning variables, the next step is to pass those variables into another function so they can actually be used. Listing 6.3.5 shows the `sayHello()` function from Listing 6.3.1 returning an array of information and a new function called `startle()`, taking two arguments, passing them through the original `sayHello()` function, and alerting the results.

Listing 6.3.5 Using Returned Function Values Passed into Another Function

```
function sayHello(greeting, exitStatement){

    /* add some passion to these dry arguments */
    var newGreeting = greeting + "!",
        newExitStatement = exitStatement + "!!";

    /* return the arguments in an array */
    return [newGreeting, newExitStatement];

}

function startle(polite, rude){

    /* call the sayHello function, with arguments and same each response to a
    ↪variable */
    var greeting = sayHello(polite, rude)[0],
        exit = sayHello(polite, rude)[1];

    /* alert the variables that have been passed through each function */
    alert(greeting + " -- " + exit);

}

/* call the function with our arguments defined */
startle("thank you", "you stink");
```

A Function as a Method

Just as you can group variables and data into objects, you can also do it with functions. When you group functions into objects, they're not called functions anymore; they're called "methods."

When I first started out with JavaScript, I came in from a design background rather than as a developer. This meant that I wasn't familiar with common programming terms such as object, function, method, loop, and so on. I quickly learned what a function was and how to work with them through a lot of Googling. But I would hear people talk about the `alert()` method and other methods native to JavaScript, and I wouldn't really get it because they look the same as functions. Why isn't it the "alert function"? I had no idea. This comes up a lot when you're dealing with JavaScript libraries as well (we get into that later in the book); everything is a method and nothing is a function, even though they all look and act the same.

Here's what's going on. In Chapter 5, "Storing Data in JavaScript," you learned about storing information in objects. I mentioned that you could also store functions in objects. When you do that, they're called methods instead of functions, but they work the same way. It's weird, I know, and it's not even an important distinction while you're coding. It's more about organizing your functions in groups to make them easier to maintain. The `alert()` method lives inside a global object (you never see it), which is why it's called a method.

Now that we're past that ordeal, organizing your functions into meaningful objects can clean up a lot of your code, especially on larger projects where you need the code organization help to keep your sanity. Listing 6.4 should look a little familiar; it shows how to organize our two functions (`sayHello` and `startle`) inside an object called "addressBookMethods." If we were building a large-scale application with many features, this would be a great way to section off the functionality meant only for the address book feature.

Listing 6.4 Grouping Similar Functions

```
var addressBookMethods = {  
  
    sayHello: function(message){  
  
        return message;  
  
    },  
    startle: function(){  
  
        alert(addressBookMethods.sayHello("hey there, called from a method"));  
  
    }  
  
}  
  
/* call the function */  
addressBookMethods.startle();
```

Calling a method is a little different from calling a function. You'll notice in Listing 6.4 that instead of calling `startle()` by itself, you have to call `addressBookMethods.startle()`. This is because before you can access the method, you have to access the object and drill down to the method.

Performance Considerations

Nesting functions in objects has the same performance implications that we spoke of when nesting variables in objects. The deeper a function is nested inside an object (`addressBookMethods`), the more resources it takes to extract. This is another place in your code where you will have to balance performance with maintainability. We're not talking a ton of time here—maybe a few milliseconds difference—but it can add up. Most of the time it won't matter, but if you find yourself needing a performance boost, function objects would be a place to look for a bottleneck. I probably wouldn't go more than a few levels deep when creating these objects. Listing 6.4 goes only one level deep, which is a nice balance between performance and maintainability.

Loops

A loop will execute a block of code over and over until you tell it to stop. It can iterate through data or HTML. For our purposes we'll mostly be looping through data. Much the way a function is a chunk of JavaScript code, a loop can make that function execute over and over—like a little buddy you have to do your repetitive tasks for you. And they're built right into the language!

For this one, we need some data to loop through. We'll be using contact information for the data and saving it to a JSON object called "contacts." Listing 6.5 shows a small sample of the contact information we'll be looping through. I find it easier to work with data that represents people, because when something goes wrong with one of the items it's more difficult to get angry at someone you know than it is at anything else. Feel free to substitute your own friends or family in the data so you don't get frustrated if something goes wrong.

Listing 6.5 Creating Data in a JSON Object

```
var contacts = {
  "addressBook" : [
    {
      "name": "hillisha",
      "email": "hill@example.com",
    },
    {
      "name": "paul",
      "email": "cleveland@example.com",
    },
    {
```

```

        "name": "vishaal",
        "email": "vish@example.com",
    },
    {
        "name": "mike",
        "email": "grady@example.com",
    },
    {
        "name": "jamie",
        "email": "dusted@example.com",
    }
]
};

```

for Loop

There are few different types of loops in JavaScript, a `while` loop, a `do-while` loop, and a `for` loop. Most of them are perfectly fine; I would avoid the `foreach` loop because it's known to be a performance hog, but the others are fine to use. A `while` loop and a `for` loop are basically the same thing, but the `for` loop is a little more direct, to the point, and it's the most common kind of loop you're going to find in the wild. In all the years I've been writing JavaScript, it's been 99% `for` loops. With that in mind, we're going to go over the `for` loop in this book.

Listing 6.5.1 will show you a basic `for` loop, and then we'll go over what's happening.

Listing 6.5.1 A `for` Loop Integrating Address Book Data

```

/* cache some initial variables */
var object = contacts.addressBook,
    contactsCount = object.length,
    i;

/* loop through each JSON object item until you hit #5, then stop */
for (i = 0; i < contactsCount; i = i + 1) {

    // code you want to execute over and over again

} // end for loop

```

Right away, you can see that we're saving some information to variables. The first variable "`object`" is saving the JSON object we create to a variable so it's a little easier to work with. The second variable, "`contactsCount`", looks through the JSON object and counts the number of items in there. This will let us know how many times to loop through the data. The third variable, "`i`", is just a way to declare the counting variable for our loop. Later on we'll be setting the value.

Inside the `for` you can see three statements. The first statement is setting the counter variable (`i`) to its initial value of 0 (we start at 0). The second statement is the condition in which you run the loop. As long as the “`i`” value is less than the overall count of items in the data, it should execute the code contained inside the loop brackets `{ }`. The last statement takes the “`i`” value and adds 1 to it each time the loop executes until it’s no longer less than the overall count. In our case, this loop will execute 5 times because there are five people in the address book.

Listing 6.5.2 will show the actual loop to iterate through the address book data saved to the JSON object, and then, using the `innerHTML` DOM method, output the result into the document’s `<body>` element. Besides the output, a main difference to note in Listing 6.5.2 is that we’re now running a check on the `contactsCount` variable to make sure it’s greater than 0 before continuing onto the loop. This is a general best practice to prevent unnecessary code from executing should there be an error with the data.

Listing 6.5.2 A `for` Loop Integrating Address Book Data

```

/* cache some initial variables */
var object = contacts.addressBook, // save the data object
    contactsCount = object.length, // how many items in the JSON object? "5"
    target = document.getElementsByTagName("body")[0], // where you're outputting the
    ↪data
    i; // declare the "i" variable for later use in the loop

/* before doing anything make sure there are contacts to loop through */
if(contactsCount > 0) {

    /* loop through each JSON object item until you hit #5, then stop */
    for (i = 0; i < contactsCount; i = i + 1) {

        /* inside the loop "i" is the array index */
        var item = object[i],
            name = item.name,
            email = item.email;

        /* insert each person's name & mailto link in the HTML */
        target.innerHTML += '<p><a href="mailto:'+ email +'>' + name + '</a></p>';

    }
}

```

It’s nice to be rid of that annoying alert box, isn’t it? Rather than alerting each value, we are now choosing a target within the HTML document (`<body>` element) and outputting the data there. This is more along the lines of what you’ll be doing in the real world, so we’ll be doing that now instead of using the `alert()` method.

Performance Considerations

As mentioned in an earlier chapter, JavaScript, by nature, is blocking. That means it will stop the download of other objects on the page until it is finished with its business. This can be very evident when dealing with loops. The data we're dealing with here is only five items in length, so there isn't a problem executing this block of code 5 times. However, as the number of elements you're looping through increases, so will the time it takes to iterate over them. This is important to note when you're looping through a lot of items because it can really bog down the loading time of a page.

Any variable that doesn't change and can be defined outside the loop *should* be defined outside the loop. You'll notice in our loop that there is a variable called `contactsCount`; it is defined outside the loop and then referenced within. We can do this because the length of the data never changes while the information is being looped through. If it were inside the loop, the length would have to be recalculated each time the loop ran, which can get very resource intensive. Little things like that can help you conserve resources when you're working with loops.

Conditionals

Conditionals are how you let your program make decisions for you. Decisions can be based on the data presented (decisions you make) or based on user input, like one of those choose-your-own adventure books. It's a way to inject some logic into your JavaScript.

Conditionals can be used for everything from outputting different information into the DOM to loading a completely different JavaScript file. They're very powerful things to have in your JavaScript toolkit.

if Statement

By far, the most common type of conditional is the `if` statement. An `if` statement checks a certain condition, and if true, executes a block of code. The `if` statement is contained within two curly brackets `{ }`, just like the loops we were talking about earlier and the functions before that.

This is best described through a coding sample so let's move right to it. In Listing 6.5.3 you can see a basic `if` statement that is being applied inside the loop of our JSON object in Listing 6.5.2. Inside the loop, if the person's name is "hillisha" the name and mailto link with an exclamation point at the end will be outputting into the document. This output should only be Hillisha's mailto link without any other names.

Listing 6.5.3 Basic `if` Statement

```

/* if "hillisha comes up, add an exclamation point to the end" */

if(name === "hillisha"){

    target.innerHTML += '<p><a href="mailto:' + email + '"'> + name + '</a>!</p>';

}

```

Note

Note that we're using "===" in the conditional to check if the names match what we're looking for. This triple equal sign operator signifies an *exact* match. There is also a double equal sign (==) you can use that means "match." It's best practice to use === rather than == because it's more specific, and when dealing in Boolean values it can get confusing because true = 1 and false = 0. Therefore if you're looking for a "false" Boolean value, using a double equal sign would not only return what you're looking for, but a "0" would do the same. In a nutshell, **use the === operator and not the == operator** and you won't hit that weird gray area of false versus 0 and true versus 1 when dealing with Booleans.

if/else Statement

In Listing 6.5.3 the output was only a single person's name because the condition was set to handle only that one instance of name === "hillisha". Normally you will want do something for the rest of the people in your address book as they are outputted. The `if/else` statement is for just that purpose.

The `if/else` statement gives you the capability to create multiple conditions and then a fall-back condition for any items that don't meet the conditions' criteria. In Listing 6.5.4 you can see that we are still looping through the address book JSON object, but this time we're setting three conditions:

- if name is hillisha
- if name is paul
- everyone else

Listing 6.5.4 `if/else` Statement

```

if(name === "hillisha"){

    /* if "hillisha comes up, add an exclamation point to the end" */
    target.innerHTML += '<p><a href="mailto:' + email + '"'> + name + '</a>!</p>';

}

```

```

} else if (name === "paul") { // line 5

    /* if "paul" comes up, add a question mark */
    target.innerHTML += '<p><a href="mailto:' + email + '>' + name + '</a>?</p>';

/* otherwise, output the person as normal*/
} else {

    target.innerHTML += '<p><a href="mailto:' + email + '>' + name + '</a></p>';

}

```

On line 5 in Listing 6.5.4, you can see that you can combine the two types of statements into `else if` to create a flow of conditional statements. Using this method, there is no limit to the amount of conditionals you can write. When you get to a large number of conditionals like this, you may consider changing from an `if/else` statement to a slightly more efficient `switch` statement.

switch Statement

A `switch` statement, on the surface, functions almost exactly like an `if/else` statement. In a `switch` statement, you first have to set a switch value (the thing you're going to check for); in this example, we have been checking for `name`, so that's the switch value. You then set up cases to test against. We checked for "hillisha" once and also "paul"; those would be the cases used. Last, there is a default state if none of the cases return as true.

The `switch` statement in Listing 6.5.5 creates the same output as the `if/else` statement in Listing 6.5.4, but under the hood and in syntax they are pretty different. Let's take a look at this `switch` statement.

Listing 6.5.5 Basic switch Statement

```

switch(name){
    case "hillisha":

        /* if "hillisha comes up, add an exclamation point to the end" */
        target.innerHTML += '<p><a href="mailto:' + email + '>' + name + '</a>!</p>';

        /* break out of the statement */
        break;

    case "paul":

        /* if "paul" comes up, add a question mark */
        target.innerHTML += '<p><a href="mailto:' + email + '>' + name + '</a>?</p>';

```

```

        /* break out of the statement */
        break;

    default:

        /* otherwise, output the people as normal*/
        target.innerHTML += '<p><a href="mailto:' + email + '>' + name + '</a></p>';
} // end switch statement

```

if versus switch

Besides syntax there is one major difference in how an `if/else` statement functions when compared to a `switch` statement. First, the `else` in an `if/else` isn't required; you can just run an `if` statement like we did in Listing 6.5.3. In a `switch` statement, the default option is required.

The iteration mechanism is also different. In the `if/else` statement in Listing 6.5.4, it still runs the same process over each item in the JSON object. For example, the first person listed is "Hillisha," so when the conditional statement is executed on that item, it asks three questions:

- Does this name equal "hillisha?" – `true`
- Does this name equal "paul?" – `false`, it's "hillisha"
- Does it equal something else – `false`

Even if the first condition is true, the statement continues checking against the other conditions. If you have a lot of conditions, this can be very resource intensive. This is where the `switch` statement really shines.

In the `switch` statement, after a condition is found to be true, it breaks out of the cases so there are no more checks made. In the `switch` statement in Listing 6.5.5, the second condition of looking for the name "paul" would look something like this:

- Does this name equal "paul?" – `false`, it's "hillisha"
- Does this name equal "paul?" – `true`, found it!
- Stop asking questions you know the answer to.

Many people like using `if/else` because it feels more natural, but after you get to a certain conditional count, you should consider moving over to the `switch` statement for a little better performance in your JavaScript.

Putting It All Together

Up to this point in the chapter, you have been building a simple address book and outputting the data.

Listing 6.6 is a cumulative dump of the code you've been putting together. It contains the JSON object with contact information, an anonymous function, and a loop with a conditional statement to check the JSON object length.

Listing 6.6 Application Code

```

/* create some data in the form of a JSON object you can consume and loop through */

var contacts = {
  "addressBook" : [
    {
      "name": "hillisha",
      "email": "hill@example.com",
    },
    {
      "name": "paul",
      "email": "cleveland@example.com",
    },
    {
      "name": "vishaal",
      "email": "vish@example.com",
    },
    {
      "name": "mike",
      "email": "grady@example.com",
    },
    {
      "name": "jamie",
      "email": "dusted@example.com",
    }
  ]
};

/* wrap everything in an anonymous function to contain the variables */
(function () {

/* cache some initial variables */

var object = contacts.addressBook, // save the JSON object
    contactsCount = object.length, // how many items in the JSON object? "5"
    target = document.getElementsByTagName("body")[0], // where you're outputting the
    ↪data

```

```

    i; // declare the "i" variable for later use in the loop

/* before doing anything make sure there are contacts to loop through */

if(contactsCount > 0) {

    /* loop through each JSON object item until you hit #5, then stop */

    for (i = 0; i < contactsCount; i = i + 1) {

        /* inside the loop "i" is the array index */

        var item = object[i],
            name = item.name,
            email = item.email;

        target.innerHTML += '<p><a href="mailto:' + email + '">' + name + '</a></p>';

    } // end for loop

} // end count check

})(); // end anonymous function

```

There's the address book application as it stands right now. You've created the contact information for our five friends and inserted them into a JSON object. After storing the JSON object, you're looping through each item (person) and outputting them individually into the `<body>` element, one after another. You're also creating HTML fragments that are paragraphs and mailto links for each person.

The processes of looping through data, storing the items as variables, and outputting them into the DOM is, by far, the most common looping method you will see as you build more applications with JavaScript. This code will not only serve as a base for our application, but as a good reference point for your future JavaScript development.

Summary

In this chapter, we started off by diving a little deeper into variables. You learned the different grouping options when declaring variables, along with some best practice considerations like why you should declare variables at the top of your JavaScript document. We also went over the list of reserved terms you should consider when naming functions and variables to help prevent collisions in your scripting file.

After that, we elaborated on the different types of functions, how they differ from each other, and discussed different case scenarios for when you might want to use each type of function.

We talked about basic functions, anonymous functions, callback functions, and functions in objects, along with how to get your functions working together by returning and passing data to one another, returning both single and multiple values.

Before this chapter, we were accessing items directly when working with data. This chapter showed how to execute the same code over and over for each data item in the form of a loop. We learned about the `for` loop specifically and talked about performance considerations and why the loop is assembled in the way it is.

After loops, we got into conditionals in the form of `if/else` and `switch` statements. They appear similar on the surface, but we also talked about why they're different and the scenarios where you may want to use one style over the other.

This chapter was the first step in building a real JavaScript application (an address book). In the next chapter, we start to bring users into the mix when we talk about events, how we might apply user interactions to this application, and learn some general information about events in JavaScript.

Exercises

1. Why is it best to position all variables at the top of your JavaScript file?
2. Why are some words reserved in JavaScript?
3. How are anonymous functions different from basic functions?

Index

Symbols

== (double equal sign), 117

A

accessibility

Ajax, 168

focus, 135

progressive enhancement, 5

Accessible Rich Internet Applications.

See ARIA (Accessible Rich Internet Applications)

accessing first and last child, 76

ActiveX control, 23

addClass() method, 225

addEventListener(), 128, 216

adding

elements to DOM, 77-78

nodes from DOM, 77

address book application, code, 120-121

address books, data, 132

advanced interface design, 280-282

createEvent() method, 282-283

dispatchEvent() method, 283-284

phantom events, 282

Ajax, 23, 54-55, 149

- accessibility, 168
- calls, 154-155
 - functions, 161-162
 - receiving data back from servers, 158-163
 - repeating, 163-164
 - returning data, 162-163
 - sending requests to the server, 155-158
- data formats, 164
 - HTML, 166-167
 - JSON (JavaScript Object Notation), 167-168
 - XML, 165-166
- file, 156
- future of, 177
- history of, 150
- jQuery, 228-229
 - getJSON, 229-230
- mistakes, 170
 - Back button, 171-172
 - providing feedback, 170-171
 - security, 172
- overview, 172-176
- readyState, 158-159
- send() method, 157-158
- server communication, 151
 - asynchronous, 151-152
 - synchronous, 151
- server response, 160
- server status, 159
- strings, 160
- URL, 156
- XML, 160

- XMLHttpRequest object, 152
 - cross-browser issues, 152-153

- ajaxCall() function, 164**

- alert() method, 106, 112**

- animation, jQuery, 227-228**

- anonymous functions, 51-52, 107-108**

- code structure, 201-202

- API, JSON (JavaScript Object Notation), 96-97**

- appendChild(), 77**

- arguments**

- basic functions, 106-107

- calling listeners, 129

- ARIA (Accessible Rich Internet Applications), 168**

- live regions, 168-169

- aria-atomic, 169

- aria-busy, 169

- aria-live, 169

- aria-relevant, 169-170

- role attribute, 246

- aria-atomic, 169**

- aria-busy, 169**

- aria-live, 169**

- aria-relevant, 169-170**

- arrays, 45-46, 85**

- associative arrays, 87

- basic arrays, 85-87

- methods, 89

- concat, 92

- join, 90

- pop, 92

- shift, 91

- slice, 90-91

- sort, 93

- unshift, 91

- multidimensional arrays, 87-88
 - pushing data into, 89
- associative arrays, 87**
- asynchronous Ajax, 151-152**
- attachEvent(), 127**
- attaching events, 124**
 - binding events, 128-129
 - event handlers, 124-125
 - event listeners, 125-127
 - unbinding events, 129-130
- attribute nodes, 70-71**
 - DOM (document object model), 62-63
 - getAttribute(), 71
 - removeAttribute(), 73
 - setAttribute(), 72
- audio, JavaScript APIs, 251-254**

B

- Back button, Ajax mistakes, 171-172**
- basic arrays, 85-87**
- basic functions, 106**
 - arguments, 106-107
 - calling, 106
- Battery Status API, 265-267**
- behavior layer, progressive enhancement, 12**
 - embedded JavaScript, 13-14
 - external and unobtrusive JavaScript, 15-16
 - inline JavaScript, 12-13
 - JavaScript, 24
- benefits**
 - of JSON (JavaScript Object Notation), 96

- of libraries, 236
 - code, 237-238
 - popularity and community, 236-237
 - of microlibraries, 240-241
 - of progressive enhancement, 16-17
 - building for the future, 17-18
 - performance, 17
 - touch interfaces, 18-19
 - of templates, 299
- Berners-Lee, Tim, 5**
- binding events, 128-129**
 - jQuery, 227
- blockquote, 6**
- blur, 134-135**
- Boolean values, 84**
- browser interactions, JavaScript, 25-26**
 - HTTP requests, 26-28
 - rendering engines, 29-30
- browser issues, libraries, 216**
- browser support, event listeners, 127-128**
- browsers**
 - DOM (document object model), 58
 - JavaScript, tools, 37

C

- cache, 45**
- callback functions, 52, 109**
- calling**
 - event listeners, 129
 - functions, 106
 - with functions, 109-110

calls, Ajax, 154-155

- functions, 161-162
- receiving data back from servers, 158-163
- repeating, 163-164
- returning data, 162-163
- sending requests to the server, 155-158

CDN (content delivery network), 236-237**chaining functions, jQuery, 232-233****change, mouse events, 135-136****chargingchange, 266****chargingtimechange, 266****children, 40-41**

- accessing, 76

classes, targeting element nodes, 67-68**click, mouse events, 132-134****code, efficient code, libraries, 237-238****code design, code organization, 185**

- comments, 190-192
- eval(), 197-199
- files and directories, 186-187
- functions as methods, 202-204
- indentation, 192
- in-document script, 187-188
- line breaks, 195-196
- math and operators, 196-197
- statement spacing, 194-195
- style guides, 199-200
- variable and function naming, 189-190
- variable declarations, 188-189
- whitespace, 193-194

code organization, 179-180

- code design, 185
- comments, 190-192

eval(), 197-199

files and directories, 186-187

functions as methods, 202-204

indentation, 192

in-document script, 187-188

line breaks, 195-196

math and operators, 196-197

statement spacing, 194-195

style guides, 199-200

variable and function naming, 189-190

variable declarations, 188-189

whitespace, 193-194

code structure, 200

anonymous functions, 201-202

functions, 200-201

functions as variables, 202

JavaScript development patterns, 204-208

coding style rules, 180-181

failing quickly, 183-185

scope, 181-183

user experiences, 185

code structure, code organization, 200

anonymous functions, 201-202

functions, 200-201

functions as variables, 202

JavaScript development patterns, 204-208

coding style rules, code organization, 180-181

failing quickly, 183-185

scope, 181-183

user experiences, 185

comments, 43

code design, 190-192

communicating with servers,
 JavaScript, 31

community, libraries, 236-237

concat, 92

conditionals, 48, 116

- if statements, 48-49, 116-117
- if/else statements, 117-118
- switch statements, 49-50, 118-119

content, 3

- changing with text node, 73-74

content delivery network (CDN), 236-237

cookies, 46

createElement(), 77

createEvent() method, 282-283

createTextNode(), 77

credentials, sending open() method, 157

cross-browser issues, XMLHttpRequest
 object, Ajax, 152-153

CSS (Cascading Style Sheets), 10

- inline CSS, 10
- JavaScript, HTTP requests, 28
- linking up stylesheets, 10-12

CSS selectors, targeting nodes, 68-70

CSS transforms, 284-289

D

data

- pushing into arrays, 89
- returning, 110
 - multiple values, 110-111
 - parsing returned values to
 another function, 111
 - single values, 110
- storing, 81, 277-278
 - arrays. *See* arrays

- getItem, 98
- HTML5, 97
- JavaScript, 31-32
- JSON (JavaScript Object
 Notation). *See* JSON (JavaScript
 Object Notation)
- objects. *See* objects, 47, 93-95
- removeItem. *See* removeItem,
 98-100
- setItem, 97-98
- variables. *See* variables
- storing chunks of data in JSON, 99

data formats, Ajax, 164

- HTML, 166-167
- JSON (JavaScript Object Notation),
 167-168
- XML, 165-166

data storage, 277-278

default behavior, preventing, 139

designers of JavaScript, 279

- advanced interface design, 280-284
- CSS transforms, 284-289
- interacting from the desktop,
 289-293

desktop-to-browser drag-and-drop
 interface, 289-292

developers of JavaScript, 293-294

- NodeJS, 299-300
 - installing, 300-301
 - writing the server, 301-302
- templates, 294-299

Device API, 265

- Battery Status API, 265-267
- Network Information API, 268-270
- Vibration API, 267-268

directories, code design, 186-187
dischargingtimechange, 266
dispatchEvent() method, 283-284
document object model (DOM). See DOM (document object model)
document.ready, jQuery, 222-223
DOM (document object model), 23, 39-40, 57

- adding and removing nodes, 77
- adding elements, 77-78
- browsers, 58
- creating templates within, 298-299
- jQuery, 225-226
- manipulating, 275-277
- moving around, 74-76
 - accessing first and last child, 76
- removing elements, 78-79
- structure, 58-59
- structures
 - attribute nodes, 62-63
 - element nodes, 59
 - text nodes, 60-61

DOM elements

- binding events to, 128
- events, 123

double equal sign (==), 117
dragover event listener, 290

E

each() method, 232
element nodes, 62

- DOM (document object model), 59
- targeting
 - by class, 67-68
 - by ID, 63-64
 - by name, 64-67

elements

- adding to DOM, 77-78
- removing from DOM, 78-79

embedded JavaScript, 13-14
EMCAScript, 23
escaping quotes, 83
eval(), code design, 197-199
event handlers, 124-125
event listeners, 125-127

- browser support, 127-128
- calling, 129

event-driven JavaScript, 124
event-driven models, 124
event-driven pattern, 207-208
events, 54, 123

- attaching, 124
 - binding events, 128-129
 - event handlers, 124-125
 - event listeners, 125-127
 - unbinding events, 129-130
- binding, jQuery, 227
- focus, accessibility, 135
- keyboard events, 130-132
 - keydown, 139-140
 - keypress, 139-140
 - keyup, 139-140
- mouse events, 130-132
 - blur, 134-135
 - change, 135-136
 - click, 132-134
 - focus, 134-135
 - mouseout, 136-137
 - mouseover, 136-137
 - submit, 137-138

- orientation events, 143
 - orientationchange, 145-146
- phantom events, 282
- preventing default behavior, 139
- touch events, 143
 - support for, 146-147
 - touchend, 144
 - touchmove, 145
 - touchstart, 144
- external JavaScript, 15-16
- external libraries, 187

F

- failing quickly, coding style rules, 183-185
- fallbacks, JavaScript, 34-36
- feature detection, 270-271
- feature support, 21
- feedback, Ajax mistakes, 170-171
- file, Ajax, 156
- file preview, creating, 292-293
- files, code design, 186-187
- find(), 225
- focus, 134-135
 - accessibility, 135
- footer, 245
- for loops, 114-115
 - performance, 116
- foreach loop, 114-115
- function naming, code design, 189-190
- functions, 50-51, 105-106
 - Ajax calls, 161-162
 - anonymous functions, 51-52, 107-108

- basic functions, 106
 - arguments, 106-107
 - calling, 106
- callback functions, 52, 109
- calling functions, 109-110
- chaining functions, jQuery, 232-233
- code structure, 200-201
 - as methods, 112-113
 - code structure, 202-204
 - as variables, code structure, 202
- functions and closures pattern, 205-207
- future of JavaScript, 24-25

G

- Geolocation API, 243
 - JavaScript APIs, 249-251
 - security, 250-251
- gesturechange, 287
- gestureend, 287
- GET versus POST, 155
- getAttribute(), 71
- getElementById() method, 64
- getElementsByClassName method, 68
- getItem, 98
- getJSON, jQuery, 229-230
- getTransformextension, 288
- global variables, 42, 182
- goals, libraries, 214-215
- Google, libraries, 237
- graceful degradation versus progressive enhancement, 6
- grouping variables, 104

H

- hasAttribute()**, 71-73
- header**, 245
- helpers**, 217
- history**
 - of Ajax, 150
 - of JavaScript, 21-22
 - origins, 22-23
 - of progressive enhancement, 4-5
- History API**, 254-255
 - popstate, 258-259
 - pushState(), 255-258
- HTML**, 6-8
 - Ajax data formats, 166-167
 - JavaScript, HTTP requests, 28
 - modifying JavaScript, 31
- HTML search form**, 130-131
- HTML5**, 8-9, 243, 244-245
 - building more accessible content, 245-247
 - creating better semantics, 245-246
 - web storage, 97
- HTTP requests, JavaScript**, 26-29
 - CSS, 28
 - HTML, 28

I

- ID, targeting element nodes**, 63-64
- if statements**, 48-49, 116-117
 - versus switch statements, 119
- if/else statements**, 117-118
- improving user experiences, JavaScript**, 32
- indentation, code design**, 192
- in-document script, code design**, 187-188
- inline CSS**, 10

- inline JavaScript**, 12-13
- innerHTML DOM method**, 115
- installing NodeJS**, 300-301
- integrating templating systems into JavaScript**, 296-297
- interacting from the desktop, JavaScript for designers**, 289-293
- interfaces, desktop-to-browser drag-and-drop interface**, 289-292

J

- Java Applets**, 22
- JavaScript**, 12, 21, 30
 - browser interactions, 25-26
 - HTTP requests, 26-28
 - rendering engines, 29-30
 - communicating with the server, 31
 - for designers, 279
 - advanced interface design, 280-284
 - CSS transforms, 284-289
 - interacting from the desktop, 289-293
 - for developers, 293-294
 - NodeJS, 299-302
 - templates, 294-299
 - embedded JavaScript, 13-14
 - external and unobtrusive JavaScript, 15-16
 - future of, 24-25
 - history of, 21-22
 - origins, 22-23
 - how to use it, 32
 - creating fallbacks, 34-36
 - improving user experience, 32
 - using responsibly, 32-34

- HTTP requests, 28-29
 - CSS, 28
 - HTML, 28
- inline JavaScript, 12-13
- modifying HTML, 31
- progressive enhancement, 23-24
 - behavior layer, 24
- rendering engines, 29-30
- storing data, 31-32
- tools, 36
 - tools built into the browser, 37
 - tools built into the language, 36-37
- JavaScript APIs, 243, 248**
 - audio and video, 251-254
 - Geolocation, 249-251
 - History API, 254-255
 - popstate, 258-259
 - pushState(), 255-258
 - navigator object, 248
 - web workers, 259
 - message event, 260-264
 - postMessage(), 260
- JavaScript development patterns, code, 204-208**
- JavaScript Object Notation (JSON), 46-47**
- join, 90
- jQuery, 218-222, 228**
 - adding style information, 226
 - Ajax, 228-229
 - getJSON, 229-230
 - animation, 227-228
 - binding events, 227
 - chaining functions, 232-233
 - document.ready, 222-223

- DOM (document object model), 225-226
- looping through data, 230-232
- selectors, 223-224
- JSON (JavaScript Object Notation), 46-47, 95-96**
 - Ajax data formats, 167-168
 - API, 96-97
 - benefits of, 96
 - storing chunks of data, 99

K

- keyboard events, 130-132**
 - keydown, 139-140
 - keypress, 139-140
 - keyup, 139-140
- keydown, 139-140**
- keypress, 139-140**
- keyup, 139-140**

L

- learning processes, libraries, 213**
- levelchange, 266**
- libraries, 211-212, 216-217**
 - benefits of, 236
 - code, 237-238
 - popularity and community, 236-237
 - browser issues, fixing, 216
 - CDN (content delivery network), 236-237
 - external libraries, 187
 - goals, 214-215
 - Google, 237

- jQuery, 218-222
 - adding style information, 226
 - Ajax, 228-229
 - animation, 227-228
 - binding events, 227
 - chaining functions, 232-233
 - document.ready, 222-223
 - DOM (document object model), 225-226
 - looping through data, 230-232
 - selectors, 223-224
- learning processes, 213
- microlibraries, 240
 - benefits of, 240-241
 - problems with, 241
- plug-ins, 233-235
- problems with, 238
 - overhead, 238-239
 - overreliance and shelf life, 239
 - performance, 239
- shortcuts, creating, 215
- syntax, 214
- templates, 295-296
- YUI, 217-218
- line breaks, code design, 195-196**
- linking up stylesheets, CSS, 10-12**
- listings**
 - Accepting a Callback Function, 162-163
 - Activating the Back Button with the History API, 259
 - Adding a Basic Click Event through a Listener, 126-127
 - Adding a Basic onClick Event through an Event Handler, 125
 - Adding a Class to Previous and Next Sibling Nodes, 75
 - Adding a Class to the First and Last Items in Our nav, 76
 - Adding a Hover Behavior to the Form, 136
 - Adding a Listener to a Form submit Event, 137-138
 - Adding Descriptive Roles to the New HTML5 Elements, 247
 - Adding focus and blur Methods to the Search Box, 134-135
 - Adding Items to an Array, 89
 - Adding Strings and Numbers Together with Operators, 44
 - Adding Style Information with jQuery, 226
 - Address Book Data, 132
 - addr.getAllContacts Method Modified to Use Mustache Templates, 297
 - Ajax Call from Chapter 8 Using an eval() Alternative, JSON.parse(), 199
 - Ajax Call from Chapter 8 Using eval(), 198-199
 - Ajax Call with jQuery Targeting Toward JSON Data, 230
 - All the Touch Methods Together, 147-148
 - Anatomy of an Event Listener, 126
 - Application Code, 120-121
 - Application Data Storage, 277-278
 - ARIA Accessibility for the Contacts Ajax Output Area, 170
 - Array Value Indexes, 86
 - An Autocomplete keyup Event Listener, 140
 - Basic Ajax Call with jQuery, 229
 - A Basic for Loop Parsing Through Each Item in the Family Array, 48

- A Basic HTML Document for a Hide Behavior Demo, 33
- Basic HTML Example of Navigation, 75
- Basic HTML Search Form, 130-131
- Basic HTML Structure to Illustrate the DOM, 58
- Basic HTML Structure with Elements, IDs and Classes, 69
- Basic if Statement, 117
- A Basic if/else Statement Checking to See if a Person Is "tim," 49
- Basic switch Statement, 118-119
- Basics of the Vibration API, 267
- Binding a click Event with jQuery, 227
- Binding an Event to a DOM Element, 128
- Bottom Section of Our HTML5 Document with External JavaScript, 15
- Browser Data Storage, 278
- Calling a Basic Function, 106
- Calling a Listener with Arguments, 129
- Chaining Functions with Ajax Methods, 232-233
- Checking for localStorage Support Before Moving Forward, 100
- Checking the Ajax Request State, 158
- Checking the Network Connection to Load Larger Images, 270
- Checking the Server Status, 159
- Children Element in HTML, 40-41
- Cleaning the Global Scope, 182-183
- Cleaning Up After Yourself, 98
- Code Block Title, 82
- Code with Better Whitespace, 194
- Code with Poor Whitespace, 194
- Complete and Cleaned JavaScript to This Point in the Chapter, 140-143
- Complete HTML Structure for Creating Custom Audio Controls, 252
- Contact Information Data the Worker Will Be Parsing, 261-262
- Contents of audio-controls.css, 205
- Contents of contacts.json Data File, 154
- Contents of script.js, 15-16
- Contents of server.js, Including the http Module, 301
- Converting getFamilyMemberNames to an Anonymous Function, 51-52
- createEvent() and initMouseEvent(), 283
- Creating an Instance of the XMLHttpRequest, 152
- Creating Code that Fails Quickly, 184-185
- Creating Custom Controls for Audio or Video Elements, 252-253
- Creating Data in a JSON Object, 113-114
- CSS Attached to the HTML Document in Listing 2.2, 34
- CSS Contained in the styles.css File Referenced in Listing 1.1.3, 11
- CSS to Apply to Our Image, 72
- Data Saved to a JSON Object, 95-96
- Declaring an Array, 85
- Declaring an Array and Assigning Values in the Same Step, 85
- Defining an Anonymous Function, 108
- dispatchEvent(), 284

- An Empty Global Variables, 42
- The Entire Contact Search Form
JavaScript to This Point, 172-176
- Escaping Quotes in a String, 83
- Example Data in HTML Format, 166
- Example Data in JSON Format, 167
- Example Data in XML Format, 165
- Example HTML for JavaScript
Patterns, 205
- Example of a Function as a Variable,
202
- An Example of a Function Calling a
Function (callback), 52
- An Example of a JavaScript
Method, 53
- Example of a Multidimensional
Array, 88
- Example of a switch Statement
Looping Through the Family
Array, 49-50
- An Example of an Associative
Array, 87
- An Example of Array Popping, 92
- Example of Bad Commenting, 191
- Example of Better Commenting,
191-192
- Example of Event-Driven JavaScript,
207-208
- An Example of Feature Detection in
Geolocation, 271
- Example of Functions and Closure
in JavaScript, 206-207
- Example of Implementing the
History API, 255-256
- An Example of Inline CSS, 10
- Example of Using CSS with
JavaScript, 276
- Executing a Method on
orientationchange, 146
- First, Last, Next, and Previous DOM
Nodes, 226
- A Function That Will Take the
Family Array, Loop, and Alert the
Results, 50-51
- Function to Return the Correct Ajax
Object, 153
- Generate HTML After Adding
Classes, 75, 76
- Geolocation Example to Return All
the Data Available, 250
- Getting Data from localStorage, 98
- Getting the File Data, 291-292
- Getting the Vendor Extension,
285-286
- Grouping Similar Functions, 112
- Grouping Variables with a Single
var Declaration, 104
- head Section of the HTML5
Document, 11
- History API with Ajax, 257-258
- HTML Document, 7
- HTML Document to Display Web
Worker Functionality, 260
- HTML Document to Show Off the
Network Information API, 269
- HTML Document with Drop Zone
for Image Upload, 289-290
- HTML Documents to Show the
History API, 254-255
- HTML Element Labeled with an ID
for Quick Access, 63
- HTML Example for Attribute Nodes,
71
- HTML Example to Create a New
Element, 77, 78
- HTML Example to Illustrate
Changing Content, 74

- HTML for Vendor Extension Example, 286-287
- HTML in JavaScript (snippet from autocomplete search form), 294
- HTML Snippet Illustrating Element Node with Classes, 67
- HTML Structure in Preparation for Mustache Templates, 295-296
- HTML Structure with Mixed and Repeating Element Nodes, 65
- HTML When Dealing with External Libraries, 187
- HTML with Concatenated Script, 188
- HTML5 Document, 8-9
- HTML5 Document with Embedded JavaScript, 13-14
- IE Fallback Code for Listeners, 127-128
- If pizzaParty Were One Large Anonymous Function, 201
- if Statement to Check for Support of the XMLHttpRequest, 153
- if/else Statement, 117-118
- Initializing a Worker, 261
- JavaScript Attached to the HTML Document in Listing 2.2, 34
- JavaScript Comments, 43
- JavaScript Example of getElementByTagName, 65
- JavaScript for Removing an Element from the DOM, 79
- JavaScript Used to Access an Element by Its ID, 64
- JavaScript Used to Get the Class Value of Our Images, 71
- JavaScript Used to Remove the Class Value of Our Images, 73
- JavaScript Used to Set the Class Attribute of Our Images, 72
- Joining All the Arrays into a Single Array, 92
- jQuery's document.ready Method, 223
- JSON Data Ajax Call with a Loop Through the Response, 231
- Linking to the Google Hosted jQuery, 237
- A List of Empty Variables, 42
- Listening for Touch Events, 284
- Load and Click Events to Execute the getFamilyMemberNames Function, 54
- A for Loop Integrating Address Book Data, 114, 115
- Main JavaScript Contents in html5.js, 262-263
- Mapping Mustache Template to the Data, 298-299
- Mapping Touch to Click, 281-282
- Method to Get All Contacts from JSON Object, 133-134
- Multiline Variable Declarations, 157-189
- Mustache Templates to Be Added to the HTML, 298
- nav Section of the HTML5 Document with Inline JavaScript, 13
- New Element Inserted into the DOM, 78
- Normal Audio Element with Standard Attributes, 251
- The Normal pizzaParty Functions, 200
- Outputting a Loading Message, 171
- Outputting the Returned Data, 160

- Parent Element in HTML, 40
- Passing a Function Variable Through Arguments, 107
- pizzaParty Function as a Method, 203-204
- Polluting the Global Scope, 181-182
- Prepping the Ajax Call, 156
- Previewing the File, 292-293
- Proper Statement Spacing, 195
- Pulling Data Out of a Multidimensional Array, 88
- Putting It All Together, 66-67
- Remove and Add Items to the Beginning of an Array, 91
- Removing an Event Listener, 129-130
- Returning Data with a Function, 110
- Returning Multiple Data Values with a Function, 111
- Returning the Total Number of Elements in Our NodeList, 65
- Reusable Ajax Function, 161
- Sample HTML to Show jQuery's Functionality, 221-222
- Saving Data in an Object, 93-94
- Saving Data to an Array, 46
- Saving Data to an Object, 47
- Saving Data to JSON, 47
- Saving Data to Variables, 43
- Saving Data with localStorage, 98
- Saving Number Values as Variables, 84
- Saving Strings to Variables, 82
- Scaling a Block, 288
- Selecting an Element by Class Name, 224
- Selecting an Element by ID, 224
- Sending Extra Data for Server-Side Filtering, 157
- Sending the Actual Data, 157
- Sending the navigator Object to the JavaScript Console for Analysis, 248
- Setting "use strict" Mode, 45
- Showing Scope Inside an Anonymous Function, 108-109
- Sibling Element in HTML, 41
- Simple jQuery Animation, 228
- Skeleton HTML Document Linking Both JavaScript Files, 220
- Slicing an Array, 91
- Slightly Modified Ajax Function from Earlier, 256-257
- Sorting the New Concatenated Array Alphabetically, 93
- Spacing and Operators, 196
- Targeting a parentNode, 75
- Traveling Through the DOM with jQuery, 225
- Using a Callback Function, 109-110
- Using a Media Query in CSS, 18
- Using a touch Event, 143-144
- Using a touchmove Event, 145
- Using Callback Function Functionality, 163
- Using getElementsByClassName, 68
- Using getElementsByTagName to Target a Single Node, 66
- Using innerHTML, 74
- Using JavaScript Responsibly, 35-36
- Using Join on an Array, 90
- Using JSON.parse to Get Data, 99
- Using JSON.stringify to Store Data, 99

- Using Mouse and Click Events, 280
- Using PEMDAS, 197
- Using querySelectors, 69
- Using Returned Values Passed into Another Function, 111
- Using setInterval, 164
- Using setInterval with Ajax, 164
- Using the Battery API, 266-267
- Using the Vendor Extension, 287
- Using the Vibration API, 268
- Using Touch-based JavaScript, 19
- Utilizing the JavaScript Console, 36
- A Variable Definition Block
 - Example, 189
- A Variable List Broken at Commas, 196
- Variable Naming Conventions, 190
- Worker Script in worker.js, 264
- Writing a Basic Function, 106
- Zebra Striping Function, 234
- Zebra Striping jQuery Plug-in, 235
- A Zoomed-In Version of preventDefault, 139
- live regions, ARIA (Accessible Rich Internet Applications), 168-169**
 - aria-atomic, 169
 - aria-busy, 169
 - aria-live, 169
 - aria-relevant, 169-170
- local variables, 41-42**
- localStorage, 97**
- looping through data, jQuery, 230-232**
- loops, 48, 113-114**
 - foreach loop, 114-115
 - for loops, 114-115
 - performance, 116

M

- MAMP, 149**
- manipulating DOM (document object model), 275-277**
- math, code design, 196-197**
- media queries, 18**
- message event, web workers, 260-264**
- methods, 53-54**
 - array methods, 89
 - concat, 92
 - join, 90
 - pop, 92
 - shift, 91
 - slice, 90-91
 - sort, 93
 - unshift, 91
 - functions as methods, 112-113
- microlibraries, 240**
 - benefits of, 240-241
 - problems with, 241
- mistakes, Ajax, 170**
 - Back button, 171-172
 - providing feedback, 170-171
 - security, 172
- modifying HTML, JavaScript, 31**
- mouseout events, 130-132**
 - blur, 134-135
 - change, 135-136
 - click, 132-134
 - focus, 134-135
 - mouseout, 136-137
 - mouseover, 136-137
 - submit, 137-138
- mouseout, 136-137**

mouseover, 136-137
mouseover events, 281
moving around, DOM (document object model), 74-76
 accessing first and last child, 76
multidimensional arrays, 87-88
Mustache, templates, 295

N

names, targeting element nodes, 64-67
nav, 12
navigator object, JavaScript APIs, 248
Netflix API, 97
Netscape, 22
Network Information API, 268-270
NodeJS, 299-300
 installing, 300-301
 writing the server, 301-302
NodeList, 65
nodes, 58
 adding/removing from DOM, 77
 attribute nodes, 70-71
 getAttribute(), 71
 removeAttribute(), 73
 setAttribute(), 72
 DOM (document object model)
 attribute nodes, 62-63
 element nodes, 59
 text nodes, 60-61
 element nodes. *See* element nodes
 targeting with CSS selectors, 68-70
 text nodes, 73-74
numbers, 83-84

O

objects, 47, 93-94
 performance, 94-95
onclick, 125
onreadystatechange event handler, 162
open() method, 156
 asynchronous or synchronous, 156
 sending credentials, 157
operators, 44
 code design, 196-197
orientation events, 143
 orientationchange, 145-146
orientationchange, 145-146
overreliance, libraries, 239

P

parent(), 225
parents, 40
parents(), 225
parsing returned values to another function, 111
patterns
 event-driven pattern, 207-208
 functions and closures pattern, 205-207
pause(), 252
performance
 functions as methods, 113
 GET versus POST, 155
 libraries, 239
 for loops, 116
 objects, 94-95
 progressive enhancement, 17
 variables, 84-85
phantom events, 282

play(), 252
plug-ins, libraries, 233-235
pop, 92
popstate, History API, 258-259
popularity, libraries, 236-237
POST versus GET, 155
postMessage(), web workers, 260
presentation layer, progressive enhancement, 9-10

- inline CSS, 10
- linking up stylesheets, 10-12

preventDefault, 139
preventing default behavior, 139
progressive enhancement, 3, 19, 274-275

- accessibility, 5
- behavior layer, 12
 - embedded JavaScript, 13-14
 - external and unobtrusive JavaScript, 15-16
 - inline JavaScript, 12-13
- benefits of, 16-17
 - building for the future, 17-18
 - performance, 17
 - touch interfaces, 18-19
- defining, 3-4
- versus graceful degradation, 6
- history of, 4-5
- JavaScript, 23-24
 - behavior layer, 24
- presentation layer, 9-10
 - inline CSS, 10
 - linking up stylesheets, 10-12
- purpose of, 5
- reusability, 5-6
- structure layer, 6-8
 - HTML5, 8-9

purpose of progressive enhancement, 5
pushState(), History API, 255-258

Q

querySelector, 69
quotes, escaping, 83

R

ready() method, 222
readyState, Ajax, 158-159
receiving data back from servers, Ajax calls, 158-163
removeAttribute(), 71-73
removeChild(), 77
removeItem, 98

- storing chunks of data with JSON, 99
- web storage, 100

removing

- elements from DOM, 78-79
- nodes from DOM, 77

renderedContent object, 297
rendering engines, JavaScript, 29-30
repeating Ajax calls, 163-164
reserved terms, variables, 104-105
Resig, John, 218
returning data, 110

- Ajax calls, 162-163
- multiple values, 110-111
- parsing returned values to another function, 111
- single values, 110

reusability, progressive enhancement, 5-6

S

scope, 108-109

- calling functions with functions, 109-110
- coding style rules, 181-183
- functions as methods, 112-113
 - performance, 113
- returning data, 110
 - multiple values, 110-111
 - parsing returned values to another function, 111
 - single values, 110

script element, 296**security**

- Ajax mistakes, 172
- Geolocation, 250-251

selectors, jQuery, 223-224**send() method, 157-158****sending**

- credentials, open() method, 157
- requests to servers, Ajax calls, 155-158

server communication, 279

- Ajax, 151
 - asynchronous, 151-152
 - synchronous, 151

server response, Ajax, 160**server status, Ajax, 159****servers, communicating with, 31****sessionStorage, 97****setAttribute(), 71-72****setItem, 97-98****shelf life, libraries, 239****shift, 91****shortcuts, libraries, 215****siblings, 41****siblings(), 225****slice, 90-91****sort, 93****source elements, 251****statement spacing, code design, 194-195****storage, 45****storing data, 81, 277-278**

- arrays. *See* arrays
- getItem, 98
- HTML5, 97
- JavaScript, 31-32
- JSON (JavaScript Object Notation). *See* JSON (JavaScript Object Notation)
- objects. *See* objects, 47, 93-95
- removeItem. *See* removeItem, 98-100
- setItem, 97-98
- variables. *See* variables

strings, 43, 82-83

- Ajax, 160

structure, DOM (document object model), 58-59**structure layer, progressive enhancement, 6-8**

- HTML5, 8-9

structures, DOM (document object model)

- attribute nodes, 62-63
- element nodes, 59
- text nodes, 60-61

style guides, code design, 199-200**style information, jQuery, 226****submit, 137-138****support for touch events, 146-147**

switch statements, 49-50, 118-119

versus if statements, 119

synchronous Ajax, 151

syntax, libraries, 214

T

targeting

element nodes

by class, 67-68

by ID, 63-64

by name, 64-67

nodes with CSS selectors, 68-70

templates, 294-299

benefits of, 299

creating within DOM, 298-299

libraries, 295-296

Mustache, 295

**templating systems, integrating into
JavaScript, 296-297**

text nodes, 73-74

DOM (document object model),
60-61

to_html method, 297

toolkits, 217

tools, JavaScript, 36

tools built into the browser, 37

tools built into the language, 36-37

touch events, 143

support for, 146-147

touchend, 144

touchmove, 145

touchstart, 144

touch interfaces, 18-19

touchend, 144

touchmove, 145

touchstart, 144

U

unbinding events, 129-130

unobtrusive JavaScript, 15-16

unshift, 91

URL, Ajax, 156

use strict, 45

user experiences

coding style rules, 185

improving JavaScript, 32

V

var declaration, 104

**variable declarations, code design,
188-189**

variable naming, code design, 189-190

variable scope, 181

variables, 41, 81-82

Boolean values, 84

defining, 103-104

global variables, 42, 182

grouping, 104

local variables, 41-42

numbers, 83-84

performance, 84-85

reserved terms, 104-105

strings, 82-83

Vibration API, 267-268

video, JavaScript APIs, 251-254

W

WAMP, 149

web storage, 100

HTML5, 97

web workers, 259

message event, 260-264

postMessage(), 260

whitespace, code design, 193-194

writing servers, NodeJS, 301-302

X

XML

Ajax, 160

Ajax data formats, 165-166

XMLHTTP, 23

XMLHttpRequest, 216

XMLHttpRequest object, 23, 150

Ajax, 152

cross-browser issues, 152-153

Y-Z

YUI, 217-218

YUI Compressor, 37