O'REILLY®

3rd Edition

Learning JavaScript

ADD SPARKLE AND LIFE TO YOUR WEB PAGES

Ethan Brown

O'REILLY®

Learning JavaScript

This is an exciting time to learn JavaScript. Now that the latest JavaScript specification—ECMAScript 6.0 (ES6)—has been finalized, learning how to develop high-quality applications with this language is easier and more satisfying than ever. This practical book takes programmers (amateurs and pros alike) on a no-nonsense tour of ES6, along with some related tools and techniques.

Author Ethan Brown (*Web Development with Node and Express*) not only guides you through simple and straightforward topics (variables, control flow, arrays), but also covers complex concepts such as functional and asynchronous programming. You'll learn how to create powerful and responsive web applications on the client, or with Node.js on the server.

- Use ES6 today and transcompile code to portable ES5
- Translate data into a format that JavaScript can use
- Understand the basic usage and mechanics of JavaScript functions
- Explore objects and object-oriented programming
- Tackle new concepts such as iterators, generators, and proxies
- Grasp the complexities of asynchronous programming
- Work with the Document Object Model for browser-based apps
- Learn Node.js fundamentals for developing server-side applications

Ethan Brown is Director of Engineering at Pop Art, an interactive marketing agency, where he is responsible for the architecture and implementation of websites and web services for clients ranging from small businesses to international enterprise companies. He has over 20 years of programming experience.

"It's high time for all JS devs to really learn JS. But I don't just mean shallow 'I got some code running' kind of learning. This book guides you to the deeper kind of learning we all need!"

—Kyle Simpson,
Author, *You Don't Know JS* series

"A well-written, compact introduction to all of JavaScript, up to and including ECMAScript 6."

—Axel Rauschmayer
Author, *Speaking JavaScript*

WEB PROGRAMMING / JAVASCRIPT

US $39.99 CAN $45.99
ISBN: 978-1-491-91491-5

9 781491 914915 53999

Twitter: @oreillymedia
facebook.com/oreilly

Learning JavaScript
by Ethan Brown

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

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

Editor: Meg Foley
Production Editor: Kristen Brown
Copyeditor: Rachel Monaghan
Proofreader: Jasmine Kwityn
Indexer: Judith McConville
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

October 2006:	First Edition
December 2008:	Second Edition
March 2016:	Third Edition

Revision History for the Third Edition

2016-02-12: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491914915* for release details.

978-1-491-91491-5

[LSI]

Table of Contents

Preface xv

1. Your First Application 1
- Where to Start 2
- The Tools 2
- A Comment on Comments 4
- Getting Started 5
- The JavaScript Console 7
- jQuery 8
- Drawing Graphics Primitive 9
- Automating Repetitive Tasks 11
- Handling User Input 12
- Hello, World 13

2. JavaScript Development Tools 15
- Writing ES6 Today 15
- ES6 Features 16
  - Installing Git 17
  - The Terminal 17
  - Your Project Root 18
  - Version Control: Git 18
  - Package Management: npm 21
  - Build Tools: Gulp and Grunt 23
  - Project Structure 24
- The Transcompilers 25
  - Running Babel with Gulp 25
- Linting 27
- Conclusion 30

3. Literals, Variables, Constants, and Data Types. 33
Variables and Constants 33
Variables or Constants: Which to Use? 35
Identifier Names 35
Literals 36
Primitive Types and Objects 37
Numbers 38
Strings 40
  Escaping 40
Special Characters 41
  Template Strings 42
  Multiline Strings 43
  Numbers as Strings 44
Booleans 44
Symbols 45
null and undefined 45
Objects 46
Number, String, and Boolean Objects 48
Arrays 49
Trailing Commas in Objects and Arrays 50
Dates 51
Regular Expressions 51
Maps and Sets 52
Data Type Conversion 52
  Converting to Numbers 52
  Converting to String 53
  Converting to Boolean 53
Conclusion 54

4. Control Flow. 55
A Control Flow Primer 55
  while Loops 59
  Block Statements 59
  Whitespace 60
  Helper Functions 61
  if…else Statement 62
  do…while Loop 63
  for Loop 64
  if Statement 65
  Putting It All Together 66
Control Flow Statements in JavaScript 68
  Control Flow Exceptions 68

Chaining if...else Statements 69
Metasyntax 69
Additional for Loop Patterns 71
switch Statements 72
for...in loop 75
for...of loop 75
Useful Control Flow Patterns 76
Using continue to Reduce Conditional Nesting 76
Using break or return to Avoid Unnecessary Computation 76
Using Value of Index After Loop Completion 77
Using Descending Indexes When Modifying Lists 77
Conclusion 78

5. Expressions and Operators. 79
Operators 81
Arithmetic Operators 81
Operator Precedence 84
Comparison Operators 85
Comparing Numbers 87
String Concatenation 88
Logical Operators 88
Truthy and Falsy Values 89
AND, OR, and NOT 89
Short-Circuit Evaluation 91
Logical Operators with Nonboolean Operands 91
Conditional Operator 92
Comma Operator 93
Grouping Operator 93
Bitwise Operators 93
typeof Operator 95
void Operator 96
Assignment Operators 96
Destructuring Assignment 98
Object and Array Operators 99
Expressions in Template Strings 100
Expressions and Control Flow Patterns 100
Converting if...else Statements to Conditional Expressions 100
Converting if Statements to Short-Circuited Logical OR Expressions 101
Conclusion 101

6. Functions. 103
Return Values 104

Calling Versus Referencing 104
Function Arguments 105
Do Arguments Make the Function? 107
Destructuring Arguments 108
Default Arguments 109
Functions as Properties of Objects 109
The this Keyword 110
Function Expressions and Anonymous Functions 112
Arrow Notation 113
call, apply, and bind 114
Conclusion 116

7. Scope. 117
Scope Versus Existence 118
Lexical Versus Dynamic Scoping 118
Global Scope 119
Block Scope 121
Variable Masking 121
Functions, Closures, and Lexical Scope 123
Immediately Invoked Function Expressions 124
Function Scope and Hoisting 125
Function Hoisting 127
The Temporal Dead Zone 127
Strict Mode 128
Conclusion 129

8. Arrays and Array Processing. 131
A Review of Arrays 131
Array Content Manipulation 132
Adding or Removing Single Elements at the Beginning or End 133
Adding Multiple Elements at the End 133
Getting a Subarray 134
Adding or Removing Elements at Any Position 134
Cutting and Replacing Within an Array 134
Filling an Array with a Specific Value 135
Reversing and Sorting Arrays 135
Array Searching 136
The Fundamental Array Operations: map and filter 138
Array Magic: reduce 140
Array Methods and Deleted or Never-Defined Elements 143
String Joining 143
Conclusion 144

9. Objects and Object-Oriented Programming. 147
Property Enumeration 147
for...in 148
Object.keys 148
Object-Oriented Programming 149
Class and Instance Creation 150
Dynamic Properties 151
Classes Are Functions 152
The Prototype 153
Static Methods 155
Inheritance 156
Polymorphism 157
Enumerating Object Properties, Revisited 158
String Representation 159
Multiple Inheritance, Mixins, and Interfaces 159
Conclusion 161

10. Maps and Sets. 163
Maps 163
Weak Maps 165
Sets 166
Weak Sets 167
Breaking the Object Habit 167

11. Exceptions and Error Handling. 169
The Error Object 169
Exception Handling with try and catch 170
Throwing Errors 171
Exception Handling and the Call Stack 171
try...catch...finally 173
Let Exceptions Be Exceptional 174

12. Iterators and Generators. 175
The Iteration Protocol 177
Generators 179
yield Expressions and Two-Way Communication 180
Generators and return 182
Conclusion 182

13. Functions and the Power of Abstract Thinking. 183
Functions as Subroutines 183
Functions as Subroutines That Return a Value 184

Functions as…Functions 185
So What? 187
  Functions Are Objects 188
IIFEs and Asynchronous Code 189
Function Variables 191
  Functions in an Array 193
  Pass a Function into a Function 194
  Return a Function from a Function 195
Recursion 196
Conclusion 197

14. Asynchronous Programming. 199
The Analogy 200
Callbacks 200
  setInterval and clearInterval 201
  Scope and Asynchronous Execution 202
  Error-First Callbacks 203
  Callback Hell 204
Promises 205
  Creating Promises 206
  Using Promises 206
  Events 208
  Promise Chaining 210
  Preventing Unsettled Promises 211
Generators 212
  One Step Forward and Two Steps Back? 215
  Don’t Write Your Own Generator Runner 216
  Exception Handling in Generator Runners 216
Conclusion 217

15. Date and Time. 219
Dates, Time Zones, Timestamps, and the Unix Epoch 219
Constructing Date Objects 220
Moment.js 221
A Practical Approach to Dates in JavaScript 222
Constructing Dates 222
  Constructing Dates on the Server 222
  Constructing Dates in the Browser 223
Transmitting Dates 223
Displaying Dates 224
Date Components 225
Comparing Dates 226

Date Arithmetic 226
User-Friendly Relative Dates 227
Conclusion 227

16. Math. 229
Formatting Numbers 229
Fixed Decimals 230
Exponential Notation 230
Fixed Precision 230
Different Bases 231
Advanced Number Formatting 231
Constants 231
Algebraic Functions 232
Exponentiation 232
Logarithmic Functions 233
Miscellaneous 233
Pseudorandom Number Generation 234
Trigonometric Functions 235
Hyperbolic Functions 236

17. Regular Expressions. 237
Substring Matching and Replacing 237
Constructing Regular Expressions 238
Searching with Regular Expressions 239
Replacing with Regular Expressions 239
Input Consumption 240
Alternation 242
Matching HTML 242
Character Sets 243
Named Character Sets 244
Repetition 245
The Period Metacharacter and Escaping 246
A True Wildcard 246
Grouping 247
Lazy Matches, Greedy Matches 248
Backreferences 249
Replacing Groups 250
Function Replacements 251
Anchoring 253
Word Boundary Matching 253
Lookaheads 254
Constructing Regexes Dynamically 255

Conclusion 256

18. JavaScript in the Browser. 257
ES5 or ES6? 257
The Document Object Model 258
Some Tree Terminology 261
DOM "Get" Methods 261
Querying DOM Elements 262
Manipulating DOM Elements 263
Creating New DOM Elements 263
Styling Elements 264
Data Attributes 265
Events 266
  Event Capturing and Bubbling 267
  Event Categories 270
Ajax 271
Conclusion 274

19. jQuery. 275
The Almighty Dollar (Sign) 275
Including jQuery 276
Waiting for the DOM to Load 276
jQuery-Wrapped DOM Elements 277
Manipulating Elements 277
Unwrapping jQuery Objects 279
Ajax 280
Conclusion 280

20. Node. 281
Node Fundamentals 281
Modules 282
Core Modules, File Modules, and npm Modules 284
Customizing Modules with Function Modules 287
Filesystem Access 289
Process 291
Operating System 294
Child Processes 294
Streams 295
Web Servers 297
Conclusion 299

21. Object Property Configuration and Proxies. 301
Accessor Properties: Getters and Setters 301
Object Property Attributes 303
Protecting Objects: Freezing, Sealing, and Preventing Extension 305
Proxies 308
Conclusion 310

22. Additional Resources. 311
Online Documentation 311
Periodicals 312
Blogs and Tutorials 312
Stack Overflow 313
Contributing to Open Source Projects 315
Conclusion 315

A. Reserved Words. 317

B. Operator Precedence. 321

Index. 325

CHAPTER 1

Your First Application

Often, the best way to learn is to *do*: so we're going to start off by creating a simple application. The point of this chapter is not to explain everything that's going on: there's a lot that's going to be unfamiliar and confusing, and my advice to you is to relax and not get caught up in trying to understand everything right now. The point of this chapter is to get you excited. Just enjoy the ride; by the time you finish this book, everything in this chapter will make perfect sense to you.

If you don't have much programming experience, one of the things that is going to cause you a lot of frustration at first is how *literal* computers are. Our human minds can deal with confusing input very easily, but computers are terrible at this. If I make a grammatical error, it may change your opinion about my writing ability, but you will probably still understand me. JavaScript—like all programming languages—has no such facility to deal with confusing input. Capitalization, spelling, and the order of words and punctuation are crucial. If you're experiencing problems, make sure you've copied everything correctly: you haven't substituted semicolons for colons or commas for periods, you haven't mixed single quotation and double quotation marks, and you've capitalized all of your code correctly. Once you've had some experience, you'll learn where you can "do things your way," and where you have to be perfectly literal, but for now, you will experience less frustration by entering the examples exactly as they're written.

Historically, programming books have started out with an example called "Hello, World" that simply prints the phrase "hello world" to your terminal. It may interest you to know that this tradition was started in 1972 by Brian Kernighan, a computer scientist working at Bell Labs. It was first seen in print in 1978 in *The C Programming*

Language, by Brian Kernighan and Dennis Ritchie. To this day, *The C Programming Language* is widely considered to be one of the best and most influential programming language books ever written, and I have taken much inspiration from that work in writing this book.

While “Hello, World” may seem dated to an increasingly sophisticated generation of programming students, the implicit meaning behind that simple phrase is as potent today as it was in 1978: they are the first words uttered by something that *you* have breathed life into. It is proof that you are Prometheus, stealing fire from the gods; a rabbi scratching the true name of God into a clay golem; Doctor Frankenstein breathing life into his creation.[1] It is this sense of creation, of genesis, that first drew me to programming. Perhaps one day, some programmer—maybe you—will give life to the first artificially sentient being. And perhaps its first words will be “hello world.”

In this chapter, we will balance the tradition that Brian Kernighan started 44 years ago with the sophistication available to programmers today. We will see “hello world” on our screen, but it will be a far cry from the blocky words etched in glowing phosphor you would have enjoyed in 1972.

Where to Start

In this book, we will cover the use of JavaScript in all its current incarnations (server-side, scripting, desktop, browser-based, and more), but for historical and practical reasons, we’re going to start with a browser-based program.

One of the reasons we’re starting with a browser-based example is that it gives us easy access to graphics libraries. Humans are inherently visual creatures, and being able to relate programming concepts to visual elements is a powerful learning tool. We will spend a lot of time in this book staring at lines of text, but let’s start out with something a little more visually interesting. I’ve also chosen this example because it organically introduces some very important concepts, such as event-driven programming, which will give you a leg up on later chapters.

The Tools

Just as a carpenter would have trouble building a desk without a saw, we can’t write software without some tools. Fortunately, the tools we need in this chapter are minimal: a browser and a text editor.

I am happy to report that, as I write this, there is not one browser on the market that is not suited to the task at hand. Even Internet Explorer—which has long been a

1 I hope you have more compassion for your creations than Dr. Frankenstein—and fare better.

thorn in the side of programmers—has cleaned up its act, and is now on par with Chrome, Firefox, Safari, and Opera. That said, my browser of choice is Firefox, and in this text, I will discuss Firefox features that will help you in your programming journey. Other browsers also have these features, but I will describe them as they are implemented in Firefox, so the path of least resistance while you go through this book will be to use Firefox.

You will need a text editor to actually write your code. The choice of text editors can be a very contentious—almost religious—debate. Broadly speaking, text editors can be categorized as text-mode editors or windowed editors. The two most popular text-mode editors are vi/vim and Emacs. One big advantage to text-mode editors is that, in addition to using them on your computer, you can use them over SSH—meaning you can remotely connect to a computer and edit your files in a familiar editor. Windowed editors can feel more modern, and add some helpful (and more familiar) user interface elements. At the end of the day, however, you are editing text only, so a windowed editor doesn't offer an inherent advantage over a text-mode editor. Popular windowed editors are Atom, Sublime Text, Coda, Visual Studio, Notepad++, TextPad, and Xcode. If you are already familiar with one of these editors, there is probably no reason to switch. If you are using Notepad on Windows, however, I highly recommend upgrading to a more sophisticated editor (Notepad++ is an easy and free choice for Windows users).

Describing all the features of your editor is beyond the scope of this book, but there are a few features that you will want to learn how to use:

Syntax highlighting

Syntax highlighting uses color to distinguish syntactic elements in your program. For example, literals might be one color and variables another (you will learn what these terms mean soon!). This feature can make it easier to spot problems in your code. Most modern text editors will have syntax highlighting enabled by default; if your code isn't multicolored, consult your editor documentation to learn how to enable it.

Bracket matching

Most programming languages make heavy use of parentheses, curly braces, and square brackets (collectively referred to as "brackets"). Sometimes, the contents of these brackets span many lines, or even more than one screen, and you'll have brackets within brackets, often of different types. It's critical that brackets match up, or "balance"; if they don't, your program won't work correctly. Bracket matching provides visual cues about where brackets begin and end, and can help you spot problems with mismatched brackets. Bracket matching is handled differently in different editors, ranging from a very subtle cue to a very obvious one. Unmatched brackets are a common source of frustration for beginners, so I strongly recommend that you learn how to use your editor's bracket-matching feature.

Code folding

Somewhat related to bracket matching is *code folding*. Code folding refers to the ability to temporarily hide code that's not relevant to what you're doing at the moment, allowing you to focus. The term comes from the idea of folding a piece of paper over on itself to hide unimportant details. Like bracket matching, code folding is handled differently by different editors.

Autocompletion

Autocompletion (also called *word completion* or *IntelliSense*[2]) is a convenience feature that attempts to guess what you are typing before you finish typing it. It has two purposes. The first is to save typing time. Instead of typing, for example, **`encodeURIComponent`**, you can simply type **`enc`**, and then select `encodeURIComponent` from a list. The second purpose is called *discoverability*. For example, if you type **`enc`** because you want to use `encodeURIComponent`, you'll find (or "discover") that there's also a function called `encodeURI`. Depending on the editor, you may even see some documentation to distinguish the two choices. Autocompletion is more difficult to implement in JavaScript than it is in many other languages because it's a loosely typed language, and because of its scoping rules (which you will learn about later). If autocompletion is an important feature to you, you may have to shop around to find an editor that meets your needs: this is an area in which some editors definitely stand out from the pack. Other editors (vim, for example) offer very powerful autocompletion, but not without some extra configuration.

A Comment on Comments

JavaScript—like most programming languages—has a syntax for making *comments* in code. Comments are completely ignored by JavaScript; they are meant for you or your fellow programmers. They allow you to add natural language explanations of what's going on when it's not clear. In this book, we'll be liberally using comments in code samples to explain what's happening.

In JavaScript, there are two kinds of comments: inline comments and block comments. An inline comment starts with two forward slashes (`//`) and extends to the end of the line. A block comment starts with a forward slash and an asterisk (`/*`) and ends with an asterisk and a forward slash (`*/`), and can span multiple lines. Here's an example that illustrates both types of comments:

2 Microsoft's terminology.

```
console.log("echo");    // prints "echo" to the console
/*
   In the previous line, everything up to the double forward slashes
   is JavaScript code, and must be valid syntax.  The double
   forward slashes start a comment, and will be ignored by JavaScript.

   This text is in a block comment, and will also be ignored
   by JavaScript.  We've chosen to indent the comments of this block
   for readability, but that's not necessary.
*/
/*Look, Ma, no indentation!*/
```

Cascading Style Sheets (CSS), which we'll see shortly, also use JavaScript syntax for block comments (inline comments are not supported in CSS). HTML (like CSS) doesn't have inline comments, and its block comments are different than JavaScript. They are surrounded by the unwieldy `<!--` and `-->`:

```
<head>
   <title>HTML and CSS Example</title>
   <!-- this is an HTML comment...
            which can span multiple lines. -->
   <style>
      body: { color: red; }
      /* this is a CSS comment...
            which can span multiple lines. */
   </style>
   <script>
      console.log("echo"); // back in JavaScript...
      /* ...so both inline and block comments
            are supported. */
   </script>
</head>
```

Getting Started

We're going to start by creating three files: an HTML file, a CSS file, and a JavaScript source file. We could do everything in the HTML file (JavaScript and CSS can be embedded in HTML), but there are certain advantages to keeping them separate. If you're new to programming, I strongly recommend that you follow along with these instructions step by step: we're going to take a very exploratory, incremental approach in this chapter, which will facilitate your learning process.

It may seem like we're doing a lot of work to accomplish something fairly simple, and there's some truth in that. I certainly could have crafted an example that does the same thing with many fewer steps, but by doing so, I would be *teaching you bad habits*. The extra steps you'll see here are ones you'll see over and over again, and while it may seem overcomplicated now, you can at least reassure yourself that you're learning to do things *the right way*.

One last important note about this chapter. This is the lone chapter in the book in which the code samples will be written in ES5 syntax, not ES6 (Harmony). This is to ensure that the code samples will run, even if you aren't using a browser that has implemented ES6. In the following chapters, we will talk about how to write code in ES6 and "transcompile" it so that it will run on legacy browsers. After we cover that ground, the rest of the book will use ES6 syntax. The code samples in this chapter are simple enough that using ES5 doesn't represent a significant handicap.

For this exercise, you'll want to make sure the files you create are in the same directory or folder. I recommend that you create a new directory or folder for this example so it doesn't get lost among your other files.

Let's start with the JavaScript file. Using a text editor, create a file called *main.js*. For now, let's just put a single line in this file:

```
console.log('main.js loaded');
```

Then create the CSS file, *main.css*. We don't actually have anything to put in here yet, so we'll just include a comment so we don't have an empty file:

```
/* Styles go here. */
```

Then create a file called *index.html*:

```
<!doctype html>
<html>
   <head>
      <link rel="stylesheet" href="main.css">
   </head>
   <body>
      <h1>My first application!</h1>
      <p>Welcome to <i>Learning JavaScript, 3rd Edition</i>.</p>

      <script src="main.js"></script>
   </body>
</html>
```

While this book isn't about HTML or web application development, many of you are learning JavaScript for that purpose, so we will point out some aspects of HTML as they relate to JavaScript development. An HTML document consists of two main parts: the *head* and the *body*. The head contains information that is *not directly displayed in your browser* (though it can affect what's displayed in your browser). The body contains the contents of your page that will be rendered in your browser. It's important to understand that elements in the head will never be shown in the browser, whereas elements in the body usually are (certain types of elements, like `<script>`, won't be visible, and CSS styles can also hide body elements).

In the head, we have the line `<link rel="stylesheet" href="main.css">`; this is what links the currently empty CSS file into your document. Then, at the end of the body, we have the line `<script src="main.js"></script>`, which is what links the JavaScript file into your document. It may seem odd to you that one goes in the head and the other goes at the end of the body. While we could have put the `<script>` tag in the head, there are performance and complexity reasons for putting it at the end of the body.

In the body, we have `<h1>My first application!</h1>`, which is first-level header text (which indicates the largest, most important text on the page), followed by a `<p>` (paragraph) tag, which contains some text, some of which is italic (denoted by the `<i>` tag).

Go ahead and load *index.html* in your browser. The easiest way to do this on most systems is to simply double-click on the file from a file browser (you can also usually drag the file onto a browser window). You'll see the body contents of your HTML file.

> There are many code samples in this book. Because HTML and JavaScript files can get very large, I won't present the whole files every time: instead, I will explain in the text where the code sample fits into the file. This may cause some trouble for beginning programmers, but understanding the way code fits together is important, and can't be avoided.

The JavaScript Console

We've already written some JavaScript: `console.log('main.js loaded')`. What did that do? The *console* is a text-only tool for programmers to help them diagnose their work. You will use the console extensively as you go through this book.

Different browsers have different ways of accessing the console. Because you will be doing this quite often, I recommend learning the keyboard shortcut. In Firefox, it's Ctrl-Shift-K (Windows and Linux) or Command-Option-K (Mac).

In the page in which you loaded *index.html*, open the JavaScript console; you should see the text "main.js loaded" (if you don't see it, try reloading the page). `console.log` is a method[3] that will print whatever you want to the console, which is very helpful for debugging and learning alike.

One of the many helpful features of the console is that, in addition to seeing output from your program, you can *enter JavaScript directly in the console*, thereby testing

3 You will learn more about the difference between a *function* and a *method* in Chapter 9.

things out, learning about JavaScript features, and even modifying your program temporarily.

jQuery

We're going to add an extremely popular client-side scripting library called *jQuery* to our page. While it is not necessary, or even germane to the task at hand, it is such a ubiquitous library that it is often the first one you will include in your web code. Even though we could easily get by without it in this example, the sooner you start getting accustomed to seeing jQuery code, the better off you will be.

At the end of the body, *before* we include our own *main.js*, we'll link in jQuery:

```
<script src="https://code.jquery.com/jquery-2.1.1.min.js"></script>

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

You'll notice that we're using an Internet URL, which means your page won't work correctly without Internet access. We're linking in jQuery from a publicly hosted *content delivery network* (CDN), which has certain performance advantages. If you will be working on your project offline, you'll have to download the file and link it from your computer instead. Now we'll modify our *main.js* file to take advantage of one of jQuery's features:

```
$(document).ready(function() {
   'use strict';
   console.log('main.js loaded');
});
```

Unless you've already had some experience with jQuery, this probably looks like gibberish. There's actually a lot going on here that won't become clear until much later. What jQuery is doing for us here is making sure that the browser has loaded all of the HTML before executing our JavaScript (which is currently just a single `console.log`). Whenever we're working with browser-based JavaScript, we'll be doing this just to establish the practice: any JavaScript you write will go between the `$(document).ready(function() {` and `});` lines. Also note the line `'use strict'`; this is something we'll learn more about later, but basically this tells the JavaScript interpreter to treat your code more rigorously. While that may not sound like a good thing at first, it actually helps you write better JavaScript, and prevents common and difficult-to-diagnose problems. We'll certainly be learning to write very rigorous JavaScript in this book!

Drawing Graphics Primitive

Among many of the benefits HTML5 brought was a standardized graphics interface. The HTML5 *canvas* allows you to draw graphics primitives like squares, circles, and polygons. Using the canvas directly can be painful, so we'll use a graphics library called Paper.js to take advantage of the HTML5 canvas.

Paper.js is not the only canvas graphics library available: KineticJS, Fabric.js, and EaselJS are very popular and robust alternatives. I've used all of these libraries, and they're all very high quality.

Before we start using Paper.js to draw things, we'll need an HTML canvas element to draw on. Add the following to the body (you can put it anywhere; after the intro paragraph, for example):

```
<canvas id="mainCanvas"></canvas>
```

Note that we've given the canvas an `id` attribute: that's how we will be able to easily refer to it from within JavaScript and CSS. If we load our page right now, we won't see anything different; not only haven't we drawn anything on the canvas, but it's a white canvas on a white page and has no width and height, making it very hard to see indeed.

Every HTML element can have an ID, and for the HTML to be valid (correctly formed), each ID must be unique. So now that we've created a canvas with the `id` "mainCanvas", we can't reuse that ID. Because of this, it's recommended that you use IDs sparingly. We're using one here because it's often easier for beginners to deal with one thing at a time, and by definition, an ID can only refer to one thing on a page.

Let's modify *main.css* so our canvas stands out on the page. If you're not familiar with CSS, that's OK—this CSS is simply setting a width and height for our HTML element, and giving it a black border:[4]

```
#mainCanvas {
   width: 400px;
   height: 400px;
   border: solid 1px black;
}
```

4 If you want to learn more about CSS and HTML, I recommend the Codecademy's free HTML & CSS track.

If you reload your page, you should see the canvas now.

Now that we have something to draw on, we'll link in Paper.js to help us with the drawing. Right after we link in jQuery, but before we link in our own *main.js*, add the following line:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/paper.js/0.9.24/ ↩
paper-full.min.js"></script>
```

Note that, as with jQuery, we're using a CDN to include Paper.js in our project.

You might be starting to realize that the order in which we link things in is very important. We're going to use both jQuery and Paper.js in our own *main.js*, so we have to link in both of those first. Neither of them depends on the other, so it doesn't matter which one comes first, but I always include jQuery first as a matter of habit, as so many things in web development depend on it.

Now that we have Paper.js linked in, we have to do a little work to configure Paper.js. Whenever you encounter code like this—repetitive code that is required before you do something—it's often called *boilerplate*. Add the following to *main.js*, right after `'use strict'` (you can remove the `console.log` if you wish):

```
paper.install(window);
paper.setup(document.getElementById('mainCanvas'));

// TODO

paper.view.draw();
```

The first line installs Paper.js in the global scope (which will make more sense in Chapter 7). The second line attaches Paper.js to the canvas, and prepares Paper.js for drawing. In the middle, where we put `TODO` is where we'll actually be doing the interesting stuff. The last line tells Paper.js to actually draw something to the screen.

Now that all of the boilerplate is out of the way, let's draw something! We'll start with a green circle in the middle of the canvas. Replace the "TODO" comment with the following lines:

```
var c = Shape.Circle(200, 200, 50);
c.fillColor = 'green';
```

Refresh your browser, and behold, a green circle. You've written your first real JavaScript. There's actually a lot going on in those two lines, but for now, it's only important to know a few things. The first line creates a circle *object*, and it does so with three *arguments*: the *x* and *y* coordinates of the center of the circle, and the radius of the circle. Recall we made our canvas 400 pixels wide and 400 pixels tall, so the center of the canvas lies at (200, 200). And a radius of 50 makes a circle that's an eighth of the width and height of the canvas. The second line sets the *fill* color, which is distinct from the outline color (called the *stroke* in Paper.js parlance). Feel free to experiment with changing those arguments.

Automating Repetitive Tasks

Consider what you'd have to do if you wanted not just to add one circle, but to fill the canvas with them, laid out in a grid. If you space the circles 50 pixels apart and make them slightly smaller, you could fit 64 of them on the canvas. Certainly you could copy the code you've already written 63 times, and by hand, modify all of the coordinates so that they're spaced out in a grid. Sounds like a lot of work, doesn't it? Fortunately, this kind of repetitive task is what computers excel at. Let's see how we can draw out 64 circles, evenly spaced. We'll replace our code that draws a single circle with the following:

```
var c;
for(var x=25; x<400; x+=50) {
   for(var y=25; y<400; y+=50) {
      c = Shape.Circle(x, y, 20);
      c.fillColor = 'green';
   }
}
```

If you refresh your browser, you'll see we have 64 green circles! If you're new to programming, what you've just written may seem confusing, but you can see it's better than writing the 128 lines it would take to do this by hand.

What we've used is called a `for` loop, which is part of the control flow syntax that we'll learn about in detail in Chapter 4. A `for` loop allows you to specify an initial condition (25), an ending condition (less than 400), and an increment value (50). We use one loop inside the other to accomplish this for both the x-axis and y-axis.

There are many ways we could have written this example. The way we've written it, we've made the *x* and *y* coordinates the important pieces of information: we explicitly specify where the circles will start and how far apart they'll be spaced. We could have approached this problem from another direction: we could have said what's important is the number of circles we want (64), and let the program figure out how to space them so that they fit on the canvas. The reason we went with this solution is that it better matches what we would have done if we had cut and pasted our circle code 64 times and figured out the spacing ourselves.

Handling User Input

So far, what we've been doing hasn't had any input from the user. The user can click on the circles, but it doesn't do anything. Likewise, trying to drag a circle would have no effect. Let's make this a little more interactive, by allowing the *user* to choose where the circles get drawn.

It's important to become comfortable with the *asynchronous* nature of user input. An *asynchronous event* is an event whose timing you don't have any control over. A user's mouse click is an example of an asynchronous event: you can't be inside your users' minds, knowing when they're going to click. Certainly you can prompt their click response, but it is up to them when—and if—they actually click. Asynchronous events arising from user input make intuitive sense, but we will cover much less intuitive asynchronous events in later chapters.

Paper.js uses an object called a *tool* to handle user input. If that choice of names seems unintuitive to you, you are in good company: I agree, and don't know why the Paper.js developers used that terminology.[5] It might help you to translate "tool" to "user input tool" in your mind. Let's replace our code that drew a grid of circles with the following code:

```
var tool = new Tool();

tool.onMouseDown = function(event) {
   var c = Shape.Circle(event.point.x, event.point.y, 20);
   c.fillColor = 'green';
};
```

The first step in this code is to create our tool object. Once we've done that, we can attach an *event handler* to it. In this case, the event handler is called `onMouseDown`. Whenever the user clicks the mouse, *the function we've attached to this handler is invoked*. This is a very important point to understand. In our previous code, the code ran right away: we refreshed the browser, and the green circles appeared automatically. That is not happening here: if it were, it would draw a single green circle somewhere on the screen. Instead, the code contained between the curly braces after `function` is executed *only when the user clicks the mouse on the canvas*.

The event handler is doing two things for you: it is executing your code *when* the mouse is clicked, and it is telling you *where* the mouse was clicked. That location is stored in a property of the argument, `event.point`, which has two properties, `x` and `y`, indicating where the mouse was clicked.

5 Technical reviewer Matt Inman suggested that the Paper.js developers might have been Photoshop users familiar with "hand tool," "direct selection tool," and so on.

Note that we could save ourselves a little typing by passing the point directly to the circle (instead of passing the *x* and *y* coordinates separately):

```
var c = Shape.Circle(event.point, 20);
```

This highlights a very important aspect of JavaScript: it's able to ascertain information about the variables that are passed in. In the previous case, if it sees three numbers in a row, it knows that they represent the *x* and *y* coordinates and the radius. If it sees two arguments, it knows that the first one is a point object, and the second one is the radius. We'll learn more about this in Chapters 6 and 9.

Hello, World

Let's conclude this chapter with a manifestation of Brian Kernighan's 1972 example. We've already done all the heavy lifting: all that remains is to add the text. Before your `onMouseDown` handler, add the following:

```
var c = Shape.Circle(200, 200, 80);
c.fillColor = 'black';
var text = new PointText(200, 200);
text.justification = 'center';
text.fillColor = 'white';
text.fontSize = 20;
text.content = 'hello world';
```

This addition is fairly straightforward: we create another circle, which will be a backdrop for our text, and then we actually create the text object (`PointText`). We specify where to draw it (the center of the screen) and some additional properties (justification, color, and size). Lastly, we specify the actual text contents ("hello world").

Note that this is not the first time we emitted text with JavaScript: we did that first with `console.log` earlier in this chapter. We certainly could have changed that text to "hello world." In many ways, that would be more analogous to the experience you would have had in 1972, but the point of the example is not the text or how it's rendered: the point is that you're creating something autonomous, which has observable effects.

By refreshing your browser with this code, you are participating in a venerable tradition of "Hello, World" examples. If this is your first "Hello, World," let me welcome you to the club. If it is not, I hope that this example has given you some insight into JavaScript.

PDF
Mobi
ePub
DAISY