

ELOQUENT JAVASCRIPT

THIRD EDITION

A Modern Introduction
to Programming

Marijn Haverbeke



ELOQUENT JAVASCRIPT

3RD EDITION

Marijn Haverbeke

Copyright © 2018 by Marijn Haverbeke

This work is licensed under a Creative Commons attribution-noncommercial license (<http://creativecommons.org/licenses/by-nc/3.0/>). All code in the book may also be considered licensed under an MIT license (<http://opensource.org/licenses/MIT>).

The illustrations are contributed by various artists: Cover and chapter illustrations by Madalina Tantareanu. Pixel art in Chapters 7 and 16 by Antonio Perdomo Pastor. Regular expression diagrams in Chapter 9 generated with regexper.com by Jeff Avallone. Village photograph in Chapter 11 by Fabrice Creuzot. Game concept for Chapter 15 by Thomas Palef.

The third edition of Eloquent JavaScript was made possible by 325 financial backers.

You can buy a print version of this book, with an extra bonus chapter included, printed by No Starch Press at <http://a-fwd.com/com=marijhaver-20&asin-com=1593279507>.

CONTENTS

Introduction	1
On programming	2
Why language matters	3
What is JavaScript?	6
Code, and what to do with it	7
Overview of this book	8
Typographic conventions	8
 1 Values, Types, and Operators	10
Values	10
Numbers	11
Strings	13
Unary operators	15
Boolean values	16
Empty values	18
Automatic type conversion	18
Summary	20
 2 Program Structure	22
Expressions and statements	22
Bindings	23
Binding names	25
The environment	25
Functions	26
The console.log function	26
Return values	27
Control flow	27
Conditional execution	28
while and do loops	30
Indenting Code	31
for loops	32
Breaking Out of a Loop	33

Updating bindings succinctly	34
Dispatching on a value with switch	34
Capitalization	35
Comments	36
Summary	37
Exercises	37
3 Functions	39
Defining a function	39
Bindings and scopes	40
Functions as values	42
Declaration notation	43
Arrow functions	44
The call stack	45
Optional Arguments	46
Closure	47
Recursion	49
Growing functions	51
Functions and side effects	54
Summary	55
Exercises	55
4 Data Structures: Objects and Arrays	57
The weresquirrel	57
Data sets	58
Properties	59
Methods	60
Objects	61
Mutability	63
The lycanthrope's log	64
Computing correlation	66
Array loops	68
The final analysis	68
Further arrayology	70
Strings and their properties	72
Rest parameters	74
The Math object	75
Destructuring	76
JSON	77
Summary	78

Exercises	79
5 Higher-Order Functions	82
Abstraction	83
Abstracting repetition	83
Higher-order functions	85
Script data set	86
Filtering arrays	87
Transforming with map	88
Summarizing with reduce	88
Composability	90
Strings and character codes	91
Recognizing text	93
Summary	95
Exercises	95
6 The Secret Life of Objects	97
Encapsulation	97
Methods	98
Prototypes	99
Classes	101
Class notation	102
Overriding derived properties	103
Maps	104
Polymorphism	106
Symbols	107
The iterator interface	108
Getters, setters, and statics	110
Inheritance	112
The instanceof operator	113
Summary	114
Exercises	115
7 Project: A Robot	117
Meadowfield	117
The task	119
Persistent data	121
Simulation	122
The mail truck's route	124
Pathfinding	124

Exercises	126
8 Bugs and Errors	128
Language	128
Strict mode	129
Types	130
Testing	131
Debugging	132
Error propagation	134
Exceptions	135
Cleaning up after exceptions	136
Selective catching	138
Assertions	140
Summary	141
Exercises	142
9 Regular Expressions	143
Creating a regular expression	143
Testing for matches	144
Sets of characters	144
Repeating parts of a pattern	146
Grouping subexpressions	147
Matches and groups	147
The Date class	148
Word and string boundaries	150
Choice patterns	150
The mechanics of matching	151
Backtracking	152
The replace method	154
Greed	155
Dynamically creating RegExp objects	157
The search method	157
The lastIndex property	158
Parsing an INI file	160
International characters	162
Summary	163
Exercises	165
10 Modules	167
Modules	167

Packages	168
Improvised modules	169
Evaluating data as code	170
CommonJS	171
ECMAScript modules	173
Building and bundling	175
Module design	176
Summary	178
Exercises	178
11 Asynchronous Programming	180
Asynchronicity	180
Crow tech	182
Callbacks	183
Promises	185
Failure	186
Networks are hard	188
Collections of promises	190
Network flooding	191
Message routing	192
Async functions	194
Generators	196
The event loop	197
Asynchronous bugs	199
Summary	200
Exercises	201
12 Project: A Programming Language	202
Parsing	202
The evaluator	207
Special forms	208
The environment	210
Functions	211
Compilation	212
Cheating	213
Exercises	214
13 JavaScript and the Browser	216
Networks and the Internet	216
The Web	218

HTML	218
HTML and JavaScript	221
In the sandbox	222
Compatibility and the browser wars	222
14 The Document Object Model	224
Document structure	224
Trees	225
The standard	226
Moving through the tree	227
Finding elements	228
Changing the document	229
Creating nodes	230
Attributes	232
Layout	233
Styling	235
Cascading styles	236
Query selectors	237
Positioning and animating	238
Summary	241
Exercises	241
15 Handling Events	243
Event handlers	243
Events and DOM nodes	244
Event objects	245
Propagation	245
Default actions	247
Key events	247
Pointer events	249
Scroll events	253
Focus events	254
Load event	255
Events and the event loop	255
Timers	257
Debouncing	257
Summary	259
Exercises	259

16 Project: A Platform Game	261
The game	261
The technology	262
Levels	262
Reading a level	263
Actors	265
Encapsulation as a burden	268
Drawing	269
Motion and collision	274
Actor updates	277
Tracking keys	279
Running the game	280
Exercises	282
 17 Drawing on Canvas	 284
SVG	284
The canvas element	285
Lines and surfaces	286
Paths	287
Curves	289
Drawing a pie chart	291
Text	292
Images	293
Transformation	295
Storing and clearing transformations	297
Back to the game	299
Choosing a graphics interface	304
Summary	305
Exercises	306
 18 HTTP and Forms	 308
The protocol	308
Browsers and HTTP	310
Fetch	312
HTTP sandboxing	313
Appreciating HTTP	314
Security and HTTPS	314
Form fields	315
Focus	317
Disabled fields	318

The form as a whole	318
Text fields	320
Checkboxes and radio buttons	321
Select fields	322
File fields	323
Storing data client-side	325
Summary	327
Exercises	328
19 Project: A Pixel Art Editor	330
Components	330
The state	332
DOM building	333
The canvas	334
The application	337
Drawing tools	339
Saving and loading	342
Undo history	345
Let's draw	346
Why is this so hard?	347
Exercises	348
20 Node.js	350
Background	350
The node command	351
Modules	352
Installing with NPM	353
The file system module	355
The HTTP module	357
Streams	359
A file server	361
Summary	366
Exercises	367
21 Project: Skill-Sharing Website	369
Design	369
Long polling	370
HTTP interface	371
The server	373
The client	380

Exercises	387
Exercise Hints	388
Program Structure	388
Functions	389
Data Structures: Objects and Arrays	390
Higher-Order Functions	392
The Secret Life of Objects	393
Project: A Robot	394
Bugs and Errors	395
Regular Expressions	395
Modules	396
Asynchronous Programming	398
Project: A Programming Language	399
The Document Object Model	400
Handling Events	400
Project: A Platform Game	402
Drawing on Canvas	402
HTTP and Forms	404
Project: A Pixel Art Editor	406
Node.js	408
Project: Skill-Sharing Website	409

“We think we are creating the system for our own purposes. We believe we are making it in our own image... But the computer is not really like us. It is a projection of a very slim part of ourselves: that portion devoted to logic, order, rule, and clarity.”

—Ellen Ullman, *Close to the Machine: Technophilia and its Discontents*

INTRODUCTION

This is a book about instructing computers. Computers are about as common as screwdrivers today, but they are quite a bit more complex, and making them do what you want them to do isn’t always easy.

If the task you have for your computer is a common, well-understood one, such as showing you your email or acting like a calculator, you can open the appropriate application and get to work. But for unique or open-ended tasks, there probably is no application.

That is where programming may come in. *Programming* is the act of constructing a *program*—a set of precise instructions telling a computer what to do. Because computers are dumb, pedantic beasts, programming is fundamentally tedious and frustrating.

Fortunately, if you can get over that fact, and maybe even enjoy the rigor of thinking in terms that dumb machines can deal with, programming can be rewarding. It allows you to do things in seconds that would take *forever* by hand. It is a way to make your computer tool do things that it couldn’t do before. And it provides a wonderful exercise in abstract thinking.

Most programming is done with programming languages. A *programming language* is an artificially constructed language used to instruct computers. It is interesting that the most effective way we’ve found to communicate with a computer borrows so heavily from the way we communicate with each other. Like human languages, computer languages allow words and phrases to be combined in new ways, making it possible to express ever new concepts.

At one point language-based interfaces, such as the BASIC and DOS prompts of the 1980s and 1990s, were the main method of interacting with computers. They have largely been replaced with visual interfaces, which are easier to learn but offer less freedom. Computer languages are still there, if you know where to look. One such language, JavaScript, is built into every modern web browser and is thus available on almost every device.

This book will try to make you familiar enough with this language to do useful and amusing things with it.

ON PROGRAMMING

Besides explaining JavaScript, I will introduce the basic principles of programming. Programming, it turns out, is hard. The fundamental rules are simple and clear, but programs built on top of these rules tend to become complex enough to introduce their own rules and complexity. You're building your own maze, in a way, and you might just get lost in it.

There will be times when reading this book feels terribly frustrating. If you are new to programming, there will be a lot of new material to digest. Much of this material will then be *combined* in ways that require you to make additional connections.

It is up to you to make the necessary effort. When you are struggling to follow the book, do not jump to any conclusions about your own capabilities. You are fine—you just need to keep at it. Take a break, reread some material, and make sure you read and understand the example programs and exercises. Learning is hard work, but everything you learn is yours and will make subsequent learning easier.

When action grows unprofitable, gather information; when information grows unprofitable, sleep.

—Ursula K. Le Guin, *The Left Hand of Darkness*

A program is many things. It is a piece of text typed by a programmer, it is the directing force that makes the computer do what it does, it is data in the computer's memory, yet it controls the actions performed on this same memory. Analogies that try to compare programs to objects we are familiar with tend to fall short. A superficially fitting one is that of a machine—lots of separate parts tend to be involved, and to make the whole thing tick, we have to consider the ways in which these parts interconnect and contribute to the operation of the whole.

A computer is a physical machine that acts as a host for these immaterial machines. Computers themselves can do only stupidly straightforward things. The reason they are so useful is that they do these things at an incredibly high speed. A program can ingeniously combine an enormous number of these simple actions to do very complicated things.

A program is a building of thought. It is costless to build, it is weightless, and it grows easily under our typing hands.

But without care, a program's size and complexity will grow out of control, confusing even the person who created it. Keeping programs under control is the main problem of programming. When a program works, it is beautiful. The

art of programming is the skill of controlling complexity. The great program is subdued—made simple in its complexity.

Some programmers believe that this complexity is best managed by using only a small set of well-understood techniques in their programs. They have composed strict rules (“best practices”) prescribing the form programs should have and carefully stay within their safe little zone.

This is not only boring, it is ineffective. New problems often require new solutions. The field of programming is young and still developing rapidly, and it is varied enough to have room for wildly different approaches. There are many terrible mistakes to make in program design, and you should go ahead and make them so that you understand them. A sense of what a good program looks like is developed in practice, not learned from a list of rules.

WHY LANGUAGE MATTERS

In the beginning, at the birth of computing, there were no programming languages. Programs looked something like this:

```
00110001 00000000 00000000
00110001 00000001 00000001
00110011 00000001 00000010
01010001 00001011 00000010
00100010 00000010 00001000
01000011 00000001 00000000
01000001 00000001 00000001
00010000 00000010 00000000
01100010 00000000 00000000
```

That is a program to add the numbers from 1 to 10 together and print out the result: $1 + 2 + \dots + 10 = 55$. It could run on a simple, hypothetical machine. To program early computers, it was necessary to set large arrays of switches in the right position or punch holes in strips of cardboard and feed them to the computer. You can probably imagine how tedious and error-prone this procedure was. Even writing simple programs required much cleverness and discipline. Complex ones were nearly inconceivable.

Of course, manually entering these arcane patterns of bits (the ones and zeros) did give the programmer a profound sense of being a mighty wizard. And that has to be worth something in terms of job satisfaction.

Each line of the previous program contains a single instruction. It could be written in English like this: