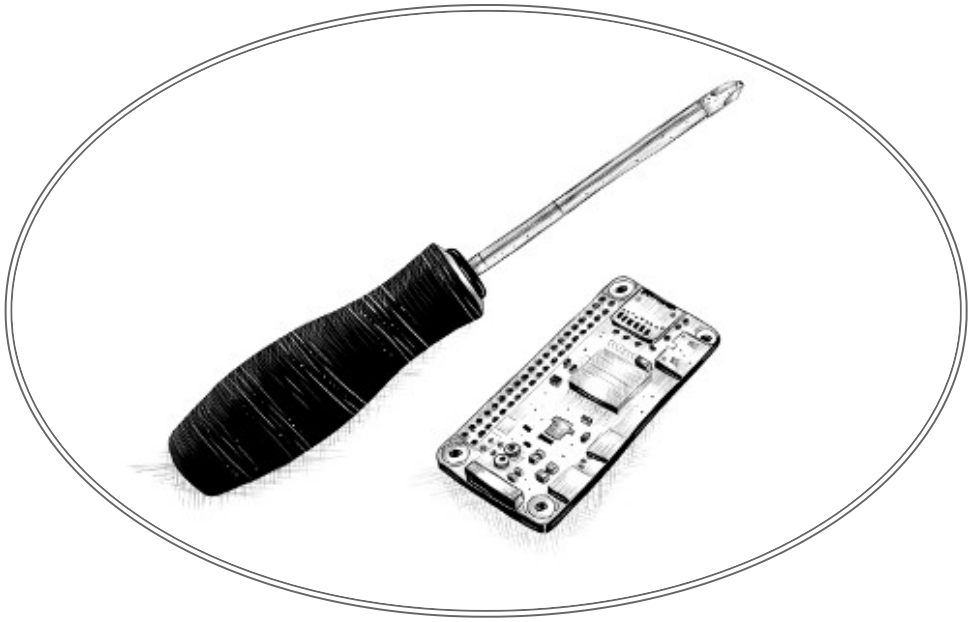
Introduction

> "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*



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 + ... + 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:

1. Store the number 0 in memory location 0.
2. Store the number 1 in memory location 1.
3. Store the value of memory location 1 in memory location 2.
4. Subtract the number 11 from the value in memory location 2.
5. If the value in memory location 2 is the number 0, continue with instruction 9.
6. Add the value of memory location 1 to memory location 0.
7. Add the number 1 to the value of memory location 1.
8. Continue with instruction 3.
9. Output the value of memory location 0.

Although that is already more readable than the soup of bits, it is still rather obscure. Using names instead of numbers for the instructions and memory locations helps.

```
  Set “total” to 0.
  Set “count” to 1.
[loop]
  Set “compare” to “count”.
  Subtract 11 from “compare”.
  If “compare” is zero, continue at [end].
  Add “count” to “total”.
  Add 1 to “count”.
  Continue at [loop].
[end]
  Output “total”.
```

Can you see how the program works at this point? The first two lines give two memory locations their starting values: `total` will be used to build up the result of the computation, and `count` will keep track of the number that we are

currently looking at. The lines using `compare` are probably the weirdest ones. The program wants to see whether `count` is equal to 11 to decide whether it can stop running. Because our hypothetical machine is rather primitive, it can only test whether a number is zero and make a decision based on that. So it uses the memory location labeled `compare` to compute the value of `count - 11` and makes a decision based on that value. The next two lines add the value of `count` to the result and increment `count` by 1 every time the program has decided that `count` is not 11 yet.

Here is the same program in JavaScript:

```
let total = 0, count = 1;
while (count <= 10) {
  total += count;
  count += 1;
}
console.log(total);
// → 55
```

This version gives us a few more improvements. Most important, there is no need to specify the way we want the program to jump back and forth anymore. The `while` construct takes care of that. It continues executing the block (wrapped in braces) below it as long as the condition it was given holds. That condition is `count <= 10`, which means "*count* is less than or equal to 10". We no longer have to create a temporary value and compare that to zero, which was just an uninteresting detail. Part of the power of programming languages is that they can take care of uninteresting details for us.

At the end of the program, after the `while` construct has finished, the `console.log` operation is used to write out the result.

Finally, here is what the program could look like if we happened to have the convenient operations `range` and `sum` available, which respectively create a collection of numbers within a range and compute the sum of a collection of numbers:

```
console.log(sum(range(1, 10)));
// → 55
```

The moral of this story is that the same program can be expressed in both long and short, unreadable and readable ways. The first version of the program was extremely obscure, whereas this last one is almost English: `log` the `sum` of the `range` of numbers from 1 to 10. (We will see in later chapters how to define operations like `sum` and `range`.)

A good programming language helps the programmer by allowing them to talk about the actions that the computer has to perform on a higher level. It helps omit details, provides convenient building blocks (such as `while` and `console.log`), allows you to define your own building blocks (such as `sum` and `range`), and makes those blocks easy to compose.

What is JavaScript?

JavaScript was introduced in 1995 as a way to add programs to web pages in the Netscape Navigator browser. The language has since been adopted by all other major graphical web browsers. It has made modern web applications possible—applications with which you can interact directly without doing a page reload for every action. JavaScript is also used in more traditional websites to provide various forms of interactivity and cleverness.

It is important to note that JavaScript has almost nothing to do with the programming language named Java. The similar name was inspired by marketing considerations rather than good judgment. When JavaScript was being introduced, the Java language was being heavily marketed and was gaining popularity. Someone thought it was a good idea to try to ride along on this success. Now we are stuck with the name.

After its adoption outside of Netscape, a standard document was written to describe the way the JavaScript language should work so that the various pieces of software that claimed to support JavaScript were actually talking about the same language. This is called the ECMAScript standard, after the

Ecma International organization that did the standardization. In practice, the terms ECMAScript and JavaScript can be used interchangeably—they are two names for the same language.

There are those who will say *terrible* things about JavaScript. Many of these things are true. When I was required to write something in JavaScript for the first time, I quickly came to despise it. It would accept almost anything I typed but interpret it in a way that was completely different from what I meant. This had a lot to do with the fact that I did not have a clue what I was doing, of course, but there is a real issue here: JavaScript is ridiculously liberal in what it allows. The idea behind this design was that it would make programming in JavaScript easier for beginners. In actuality, it mostly makes finding problems in your programs harder because the system will not point them out to you.

This flexibility also has its advantages, though. It leaves space for a lot of techniques that are impossible in more rigid languages, and as you will see (for example in Chapter 10), it can be used to overcome some of JavaScript's shortcomings. After learning the language properly and working with it for a while, I have learned to actually *like* JavaScript.

There have been several versions of JavaScript. ECMAScript version 3 was the widely supported version in the time of JavaScript's ascent to dominance, roughly between 2000 and 2010. During this time, work was underway on an ambitious version 4, which planned a number of radical improvements and extensions to the language. Changing a living, widely used language in such a radical way turned out to be politically difficult, and work on the version 4 was abandoned in 2008, leading to a much less ambitious version 5, which made only some uncontroversial improvements, coming out in 2009. Then in 2015 version 6 came out, a major update that included some of the ideas planned for version 4. Since then we've had new, small updates every year.

The fact that the language is evolving means that browsers have to constantly keep up, and if you're using an older browser, it may not support every feature. The language designers are careful to not make any changes that could break

existing programs, so new browsers can still run old programs. In this book, I'm using the 2017 version of JavaScript.

Web browsers are not the only platforms on which JavaScript is used. Some databases, such as MongoDB and CouchDB, use JavaScript as their scripting and query language. Several platforms for desktop and server programming, most notably the Node.js project (the subject of Chapter 20), provide an environment for programming JavaScript outside of the browser.

Code, and what to do with it

Code is the text that makes up programs. Most chapters in this book contain quite a lot of code. I believe reading code and writing code are indispensable parts of learning to program. Try to not just glance over the examples—read them attentively and understand them. This may be slow and confusing at first, but I promise that you'll quickly get the hang of it. The same goes for the exercises. Don't assume you understand them until you've actually written a working solution.

I recommend you try your solutions to exercises in an actual JavaScript interpreter. That way, you'll get immediate feedback on whether what you are doing is working, and, I hope, you'll be tempted to experiment and go beyond the exercises.

When reading this book in your browser, you can edit (and run) all example programs by clicking them.

If you want to run the programs defined in this book outside of the book's website, some care will be required. Many examples stand on their own and should work in any JavaScript environment. But code in later chapters is often written for a specific environment (the browser or Node.js) and can run only there. In addition, many chapters define bigger programs, and the pieces of code that appear in them depend on each other or on external files. The sandbox on the website provides links to Zip files containing all the scripts and data files necessary to run the code for a given chapter.

Overview of this book

This book contains roughly three parts. The first 12 chapters discuss the JavaScript language. The next seven chapters are about web browsers and the way JavaScript is used to program them. Finally, two chapters are devoted to Node.js, another environment to program JavaScript in.

Throughout the book, there are five *project chapters*, which describe larger example programs to give you a taste of actual programming. In order of appearance, we will work through building a delivery robot, a programming language, a platform game, a pixel paint program, and a dynamic website.

The language part of the book starts with four chapters that introduce the basic structure of the JavaScript language. They introduce control structures (such as the `while` word you saw in this introduction), functions (writing your own building blocks), and data structures. After these, you will be able to write basic programs. Next, Chapters 5 and 6 introduce techniques to use functions and objects to write more *abstract* code and keep complexity under control.

After a first project chapter, the language part of the book continues with chapters on error handling and bug fixing, regular expressions (an important tool for working with text), modularity (another defense against complexity), and asynchronous programming (dealing with events that take time). The second project chapter concludes the first part of the book.

The second part, Chapters 13 to 19, describes the tools that browser JavaScript has access to. You'll learn to display things on the screen (Chapters 14 and 17), respond to user input (Chapter 15), and communicate over the network (Chapter 18). There are again two project chapters in this part.

After that, Chapter 20 describes Node.js, and Chapter 21 builds a small website using that tool.

Typographic conventions

In this book, text written in a `monospaced` font will represent elements of programs—sometimes they are self-sufficient fragments, and sometimes they just refer to part of a nearby program. Programs (of which you have already seen a few) are written as follows:

```
function factorial(n) {
  if (n == 0) {
    return 1;
  } else {
    return factorial(n - 1) * n;
  }
}
```

Sometimes, to show the output that a program produces, the expected output is written after it, with two slashes and an arrow in front.

```
console.log(factorial(8));
// → 40320
```

Good luck!