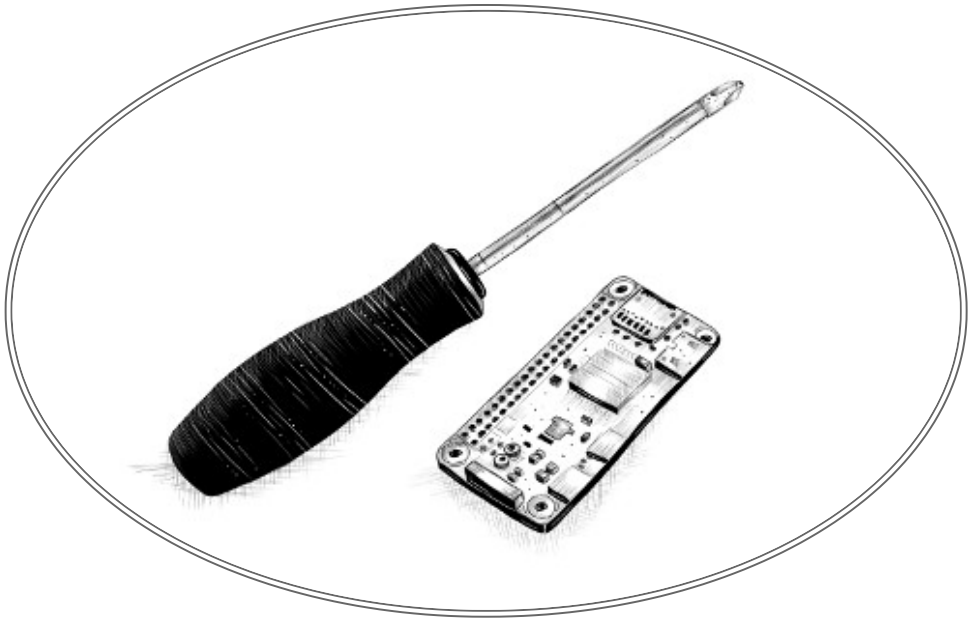
Introduction

> "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.

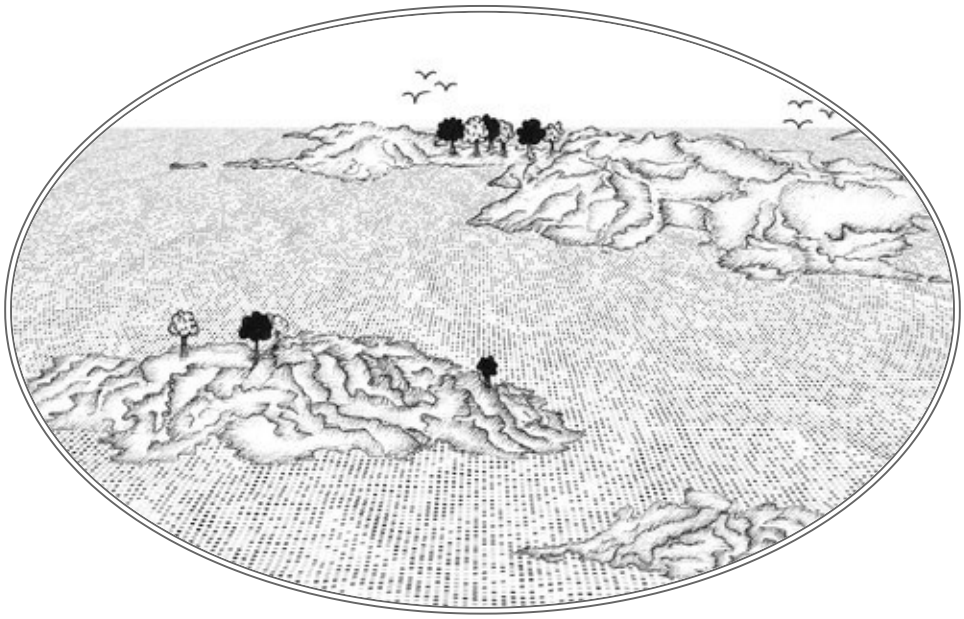
```
console.log(factorial(8));
// → 40320
```

Good luck!

CHAPTER 1

VALUES, TYPES, AND OPERATORS

> "Below the surface of the machine, the program moves. Without effort, it expands and contracts. In great harmony, electrons scatter and regroup. The forms on the monitor are but ripples on the water. The essence stays invisibly below." ¶
>
> —Master Yuan-Ma, *The Book of Programming*



Inside the computer's world, there is only data. You can read data, modify data, create new data—but that which isn't data cannot be mentioned. All this data is stored as long sequences of bits and is thus fundamentally alike.

Bits are any kind of two-valued things, usually described as zeros and ones. Inside the computer, they take forms such as a high or low electrical charge, a strong or weak signal, or a shiny or dull spot on the surface of a CD. Any piece of discrete information can be reduced to a sequence of zeros and ones and thus represented in bits.

For example, we can express the number 13 in bits. It works the same way as a decimal number, but instead of 10 different digits, you have only 2, and the

weight of each increases by a factor of 2 from right to left. Here are the bits that make up the number 13, with the weights of the digits shown below them:

```
  0   0   0   0   1   1   0   1
128  64  32  16   8   4   2   1
```

So that's the binary number 00001101. Its non-zero digits stand for 8, 4, and 1, and add up to 13.

Values

Imagine a sea of bits—an ocean of them. A typical modern computer has more than 30 billion bits in its volatile data storage (working memory). Nonvolatile storage (the hard disk or equivalent) tends to have yet a few orders of magnitude more.

To be able to work with such quantities of bits without getting lost, we must separate them into chunks that represent pieces of information. In a JavaScript environment, those chunks are called *values*. Though all values are made of bits, they play different roles. Every value has a type that determines its role. Some values are numbers, some values are pieces of text, some values are functions, and so on.

To create a value, you must merely invoke its name. This is convenient. You don't have to gather building material for your values or pay for them. You just call for one, and *whoosh*, you have it. They are not really created from thin air, of course. Every value has to be stored somewhere, and if you want to use a gigantic amount of them at the same time, you might run out of memory. Fortunately, this is a problem only if you need them all simultaneously. As soon as you no longer use a value, it will dissipate, leaving behind its bits to be recycled as building material for the next generation of values.

This chapter introduces the atomic elements of JavaScript programs, that is, the simple value types and the operators that can act on such values.

Numbers

Values of the *number* type are, unsurprisingly, numeric values. In a JavaScript program, they are written as follows:

```
13
```

Use that in a program, and it will cause the bit pattern for the number 13 to come into existence inside the computer's memory.

JavaScript uses a fixed number of bits, 64 of them, to store a single number value. There are only so many patterns you can make with 64 bits, which means that the number of different numbers that can be represented is limited. With N decimal digits, you can represent 10^N numbers. Similarly, given 64 binary digits, you can represent 2^{64} different numbers, which is about 18 quintillion (an 18 with 18 zeros after it). That's a lot.

Computer memory used to be much smaller, and people tended to use groups of 8 or 16 bits to represent their numbers. It was easy to accidentally *overflow* such small numbers—to end up with a number that did not fit into the given number of bits. Today, even computers that fit in your pocket have plenty of memory, so you are free to use 64-bit chunks, and you need to worry about overflow only when dealing with truly astronomical numbers.

Not all whole numbers less than 18 quintillion fit in a JavaScript number, though. Those bits also store negative numbers, so one bit indicates the sign of the number. A bigger issue is that nonwhole numbers must also be represented. To do this, some of the bits are used to store the position of the decimal point. The actual maximum whole number that can be stored is more in the range of 9 quadrillion (15 zeros)—which is still pleasantly huge.

Fractional numbers are written by using a dot.

```
9.81
```

For very big or very small numbers, you may also use scientific notation by adding an *e* (for *exponent*), followed by the exponent of the number.

```
2.998e8
```

That is $2.998 \times 10^8 = 299{,}800{,}000$.

Calculations with whole numbers (also called *integers*) smaller than the aforementioned 9 quadrillion are guaranteed to always be precise. Unfortunately, calculations with fractional numbers are generally not. Just as π (pi) cannot be precisely expressed by a finite number of decimal digits, many numbers lose some precision when only 64 bits are available to store them. This is a shame, but it causes practical problems only in specific situations. The important thing is to be aware of it and treat fractional digital numbers as approximations, not as precise values.

Arithmetic

The main thing to do with numbers is arithmetic. Arithmetic operations such as addition or multiplication take two number values and produce a new number from them. Here is what they look like in JavaScript:

```
100 + 4 * 11
```

The `+` and `*` symbols are called *operators*. The first stands for addition, and the second stands for multiplication. Putting an operator between two values will apply it to those values and produce a new value.

But does the example mean “add 4 and 100, and multiply the result by 11,” or is the multiplication done before the adding? As you might have guessed, the multiplication happens first. But as in mathematics, you can change this by wrapping the addition in parentheses.

```
(100 + 4) * 11
```

For subtraction, there is the `-` operator, and division can be done with the `/` operator.

When operators appear together without parentheses, the order in which they are applied is determined by the *precedence* of the operators. The example

shows that multiplication comes before addition. The `/` operator has the same precedence as `*`. Likewise for `+` and `-`. When multiple operators with the same precedence appear next to each other, as in `1 - 2 + 1`, they are applied left to right: `(1 - 2) + 1`.

These rules of precedence are not something you should worry about. When in doubt, just add parentheses.

There is one more arithmetic operator, which you might not immediately recognize. The `%` symbol is used to represent the *remainder* operation. `X % Y` is the remainder of dividing `X` by `Y`. For example, `314 % 100` produces `14`, and `144 % 12` gives `0`. The remainder operator's precedence is the same as that of multiplication and division. You'll also often see this operator referred to as *modulo*.

Special numbers

There are three special values in JavaScript that are considered numbers but don't behave like normal numbers.

The first two are `Infinity` and `-Infinity`, which represent the positive and negative infinities. `Infinity - 1` is still `Infinity`, and so on. Don't put too much trust in infinity-based computation, though. It isn't mathematically sound, and it will quickly lead to the next special number: `NaN`.

`NaN` stands for "not a number", even though it *is* a value of the number type. You'll get this result when you, for example, try to calculate `0 / 0` (zero divided by zero), `Infinity - Infinity`, or any number of other numeric operations that don't yield a meaningful result.

Strings

The next basic data type is the *string*. Strings are used to represent text. They are written by enclosing their content in quotes.

```
`Down on the sea`
"Lie on the ocean"
```

```
'Float on the ocean'
```

You can use single quotes, double quotes, or backticks to mark strings, as long as the quotes at the start and the end of the string match.

Almost anything can be put between quotes, and JavaScript will make a string value out of it. But a few characters are more difficult. You can imagine how putting quotes between quotes might be hard. *Newlines* (the characters you get when you press ENTER) can be included without escaping only when the string is quoted with backticks (`).

To make it possible to include such characters in a string, the following notation is used: whenever a backslash (`\`) is found inside quoted text, it indicates that the character after it has a special meaning. This is called *escaping* the character. A quote that is preceded by a backslash will not end the string but be part of it. When an `n` character occurs after a backslash, it is interpreted as a newline. Similarly, a `t` after a backslash means a tab character. Take the following string:

```
"This is the first line\nAnd this is the second"
```

The actual text contained is this:

```
This is the first line
And this is the second
```

There are, of course, situations where you want a backslash in a string to be just a backslash, not a special code. If two backslashes follow each other, they will collapse together, and only one will be left in the resulting string value. This is how the string "*A newline character is written like `"\n"`.*" can be expressed:

```
"A newline character is written like \"\\n\"."
```

Strings, too, have to be modeled as a series of bits to be able to exist inside the computer. The way JavaScript does this is based on the *Unicode* standard. This

standard assigns a number to virtually every character you would ever need, including characters from Greek, Arabic, Japanese, Armenian, and so on. If we have a number for every character, a string can be described by a sequence of numbers.

And that's what JavaScript does. But there's a complication: JavaScript's representation uses 16 bits per string element, which can describe up to 2^{16} different characters. But Unicode defines more characters than that—about twice as many, at this point. So some characters, such as many emoji, take up two "character positions" in JavaScript strings. We'll come back to this in Chapter 5.

Strings cannot be divided, multiplied, or subtracted, but the `+` operator *can* be used on them. It does not add, but it *concatenates*—it glues two strings together. The following line will produce the string `"concatenate"`:

```
"con" + "cat" + "e" + "nate"
```

String values have a number of associated functions (*methods*) that can be used to perform other operations on them. I'll say more about these in Chapter 4.

Strings written with single or double quotes behave very much the same—the only difference is in which type of quote you need to escape inside of them. Backtick-quoted strings, usually called *template literals*, can do a few more tricks. Apart from being able to span lines, they can also embed other values.

```
`half of 100 is ${100 / 2}`
```

When you write something inside `${}` in a template literal, its result will be computed, converted to a string, and included at that position. The example produces "*half of 100 is 50*".

Unary operators

Not all operators are symbols. Some are written as words. One example is the `typeof` operator, which produces a string value naming the type of the value you give it.

```
console.log(typeof 4.5)
// → number
console.log(typeof "x")
// → string
```

edit & run code by clicking it

We will use `console.log` in example code to indicate that we want to see the result of evaluating something. More about that in the next chapter.

The other operators shown all operated on two values, but `typeof` takes only one. Operators that use two values are called *binary* operators, while those that take one are called *unary* operators. The minus operator can be used both as a binary operator and as a unary operator.

```
console.log(- (10 - 2))
// → -8
```

Boolean values

It is often useful to have a value that distinguishes between only two possibilities, like “yes” and “no” or “on” and “off”. For this purpose, JavaScript has a *Boolean* type, which has just two values, true and false, which are written as those words.

Comparison

Here is one way to produce Boolean values:

```
console.log(3 > 2)
// → true
console.log(3 < 2)
// → false
```

The > and < signs are the traditional symbols for “is greater than” and “is less than”, respectively. They are binary operators. Applying them results in a

Boolean value that indicates whether they hold true in this case.

Strings can be compared in the same way.

```
console.log("Aardvark" < "Zoroaster")
// → true
```

The way strings are ordered is roughly alphabetic but not really what you'd expect to see in a dictionary: uppercase letters are always "less" than lowercase ones, so `"Z" < "a"`, and nonalphabetic characters (!, -, and so on) are also included in the ordering. When comparing strings, JavaScript goes over the characters from left to right, comparing the Unicode codes one by one.

Other similar operators are `>=` (greater than or equal to), `<=` (less than or equal to), `==` (equal to), and `!=` (not equal to).

```
console.log("Itchy" != "Scratchy")
// → true
console.log("Apple" == "Orange")
// → false
```

There is only one value in JavaScript that is not equal to itself, and that is `NaN` ("not a number").

```
console.log(NaN == NaN)
// → false
```

`NaN` is supposed to denote the result of a nonsensical computation, and as such, it isn't equal to the result of any *other* nonsensical computations.

Logical operators

There are also some operations that can be applied to Boolean values themselves. JavaScript supports three logical operators: *and*, *or*, and *not*. These can be used to "reason" about Booleans.

The `&&` operator represents logical *and*. It is a binary operator, and its result is true only if both the values given to it are true.

```
console.log(true && false)
// → false
console.log(true && true)
// → true
```

The `||` operator denotes logical *or*. It produces true if either of the values given to it is true.

```
console.log(false || true)
// → true
console.log(false || false)
// → false
```

Not is written as an exclamation mark (`!`). It is a unary operator that flips the value given to it—`!true` produces `false`, and `!false` gives `true`.

When mixing these Boolean operators with arithmetic and other operators, it is not always obvious when parentheses are needed. In practice, you can usually get by with knowing that of the operators we have seen so far, `||` has the lowest precedence, then comes `&&`, then the comparison operators (`>`, `==`, and so on), and then the rest. This order has been chosen such that, in typical expressions like the following one, as few parentheses as possible are necessary:

```
1 + 1 == 2 && 10 * 10 > 50
```

The last logical operator I will discuss is not unary, not binary, but *ternary*, operating on three values. It is written with a question mark and a colon, like this:

```
console.log(true ? 1 : 2);
// → 1
console.log(false ? 1 : 2);
// → 2
```

This one is called the *conditional* operator (or sometimes just the *ternary* operator since it is the only such operator in the language). The value on the

left of the question mark "picks" which of the other two values will come out. When it is true, it chooses the middle value, and when it is false, it chooses the value on the right.

Empty values

There are two special values, written `null` and `undefined`, that are used to denote the absence of a *meaningful* value. They are themselves values, but they carry no information.

Many operations in the language that don't produce a meaningful value (you'll see some later) yield `undefined` simply because they have to yield *some* value.

The difference in meaning between `undefined` and `null` is an accident of JavaScript's design, and it doesn't matter most of the time. In cases where you actually have to concern yourself with these values, I recommend treating them as mostly interchangeable.

Automatic type conversion

In the Introduction, I mentioned that JavaScript goes out of its way to accept almost any program you give it, even programs that do odd things. This is nicely demonstrated by the following expressions:

```
console.log(8 * null)
// → 0
console.log("5" - 1)
// → 4
console.log("5" + 1)
// → 51
console.log("five" * 2)
// → NaN
console.log(false == 0)
// → true
```

When an operator is applied to the "wrong" type of value, JavaScript will quietly convert that value to the type it needs, using a set of rules that often aren't what you want or expect. This is called *type coercion*. The `null` in the

first expression becomes `0`, and the `"5"` in the second expression becomes `5` (from string to number). Yet in the third expression, `+` tries string concatenation before numeric addition, so the `1` is converted to `"1"` (from number to string).

When something that doesn't map to a number in an obvious way (such as `"five"` or `undefined`) is converted to a number, you get the value `NaN`. Further arithmetic operations on `NaN` keep producing `NaN`, so if you find yourself getting one of those in an unexpected place, look for accidental type conversions.

When comparing values of the same type using `==`, the outcome is easy to predict: you should get true when both values are the same, except in the case of `NaN`. But when the types differ, JavaScript uses a complicated and confusing set of rules to determine what to do. In most cases, it just tries to convert one of the values to the other value's type. However, when `null` or `undefined` occurs on either side of the operator, it produces true only if both sides are one of `null` or `undefined`.

```
console.log(null == undefined);
// → true
console.log(null == 0);
// → false
```

That behavior is often useful. When you want to test whether a value has a real value instead of `null` or `undefined`, you can compare it to `null` with the `==` (or `!=`) operator.

But what if you want to test whether something refers to the precise value `false`? Expressions like `0 == false` and `"" == false` are also true because of automatic type conversion. When you do *not* want any type conversions to happen, there are two additional operators: `===` and `!==`. The first tests whether a value is *precisely* equal to the other, and the second tests whether it is not precisely equal. So `"" === false` is false as expected.

I recommend using the three-character comparison operators defensively to prevent unexpected type conversions from tripping you up. But when you're certain the types on both sides will be the same, there is no problem with using the shorter operators.

Short-circuiting of logical operators

The logical operators `&&` and `||` handle values of different types in a peculiar way. They will convert the value on their left side to Boolean type in order to decide what to do, but depending on the operator and the result of that conversion, they will return either the *original* left-hand value or the right-hand value.

The `||` operator, for example, will return the value to its left when that can be converted to true and will return the value on its right otherwise. This has the expected effect when the values are Boolean and does something analogous for values of other types.

```
console.log(null || "user")
// → user
console.log("Agnes" || "user")
// → Agnes
```

We can use this functionality as a way to fall back on a default value. If you have a value that might be empty, you can put `||` after it with a replacement value. If the initial value can be converted to false, you'll get the replacement instead. The rules for converting strings and numbers to Boolean values state that `0`, `NaN`, and the empty string (`""`) count as `false`, while all the other values count as `true`. So `0 || -1` produces `-1`, and `"" || "!?"` yields `"!?"`.

The `&&` operator works similarly but the other way around. When the value to its left is something that converts to false, it returns that value, and otherwise it returns the value on its right.

Another important property of these two operators is that the part to their right is evaluated only when necessary. In the case of `true || X`, no matter what `X`

is—even if it's a piece of program that does something *terrible*—the result will be true, and `X` is never evaluated. The same goes for `false && X`, which is false and will ignore `X`. This is called *short-circuit evaluation*.

The conditional operator works in a similar way. Of the second and third values, only the one that is selected is evaluated.

Summary

We looked at four types of JavaScript values in this chapter: numbers, strings, Booleans, and undefined values.

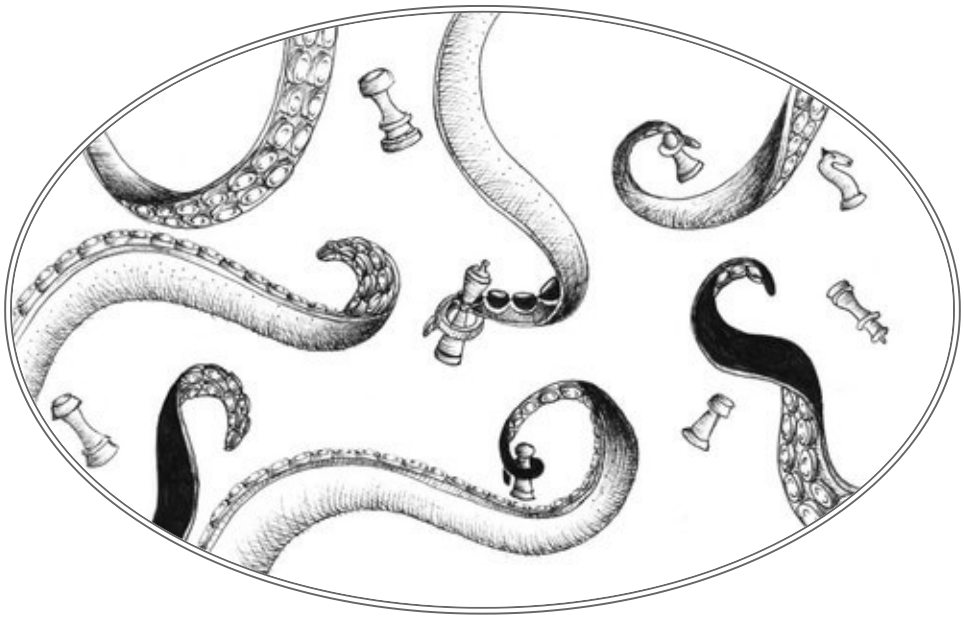
Such values are created by typing in their name (`true`, `null`) or value (`13`, `"abc"`). You can combine and transform values with operators. We saw binary operators for arithmetic (`+`, `-`, `*`, `/`, and `%`), string concatenation (`+`), comparison (`==`, `!=`, `===`, `!==`, `<`, `>`, `<=`, `>=`), and logic (`&&`, `||`), as well as several unary operators (`-` to negate a number, `!` to negate logically, and `typeof` to find a value's type) and a ternary operator (`?:`) to pick one of two values based on a third value.

This gives you enough information to use JavaScript as a pocket calculator but not much more. The next chapter will start tying these expressions together into basic programs.

CHAPTER 2

PROGRAM STRUCTURE

> "And my heart glows bright red under my filmy, translucent skin and they have to administer 10cc of JavaScript to get me to come back. (I respond well to toxins in the blood.) Man, that stuff will kick the peaches right out your gills!"
>
> — _why, *Why's (Poignant) Guide to Ruby*



In this chapter, we will start to do things that can actually be called *programming*. We will expand our command of the JavaScript language beyond the nouns and sentence fragments we've seen so far, to the point where we can express meaningful prose.

EXPRESSIONS AND STATEMENTS

In Chapter 1, we made values and applied operators to them to get new values. Creating values like this is the main substance of any JavaScript program. But that substance has to be framed in a larger structure to be useful. So that's what we'll cover next.

A fragment of code that produces a value is called an *expression*. Every value that is written literally (such as `22` or `"psychoanalysis"`) is an expression. An expression between parentheses is also an expression, as is a binary operator applied to two expressions or a unary operator applied to one.

This shows part of the beauty of a language-based interface. Expressions can contain other expressions in a way similar to how subsentences in human languages are nested—a subsentence can contain its own subsentences, and so on. This allows us to build expressions that describe arbitrarily complex computations.

If an expression corresponds to a sentence fragment, a JavaScript *statement* corresponds to a full sentence. A program is a list of statements.

The simplest kind of statement is an expression with a semicolon after it. This is a program:

```
1;
!false;
```

edit & run code by clicking it

It is a useless program, though. An expression can be content to just produce a value, which can then be used by the enclosing code. A statement stands on its own, so it amounts to something only if it affects the world. It could display something on the screen—that counts as changing the world—or it could change the internal state of the machine in a way that will affect the statements that come after it. These changes are called *side effects*. The statements in the previous example just produce the values `1` and `true` and then immediately throw them away. This leaves no impression on the world at all. When you run this program, nothing observable happens.

In some cases, JavaScript allows you to omit the semicolon at the end of a statement. In other cases, it has to be there, or the next line will be treated as part of the same statement. The rules for when it can be safely omitted are somewhat complex and error-prone. So in this book, every statement that needs a semicolon will always get one. I recommend you do the same, at least until you've learned more about the subtleties of missing semicolons.

Bindings

How does a program keep an internal state? How does it remember things? We have seen how to produce new values from old values, but this does not change the old values, and the new value has to be immediately used or it will dissipate again. To catch and hold values, JavaScript provides a thing called a *binding*, or *variable*:

```
let caught = 5 * 5;
```

That's a second kind of statement. The special word (*keyword*) `let` indicates that this sentence is going to define a binding. It is followed by the name of the binding and, if we want to immediately give it a value, by an `=` operator and an expression.

The previous statement creates a binding called `caught` and uses it to grab hold of the number that is produced by multiplying 5 by 5.

After a binding has been defined, its name can be used as an expression. The value of such an expression is the value the binding currently holds. Here's an example:

```
let ten = 10;
console.log(ten * ten);
// → 100
```

When a binding points at a value, that does not mean it is tied to that value forever. The `=` operator can be used at any time on existing bindings to disconnect them from their current value and have them point to a new one.

```
let mood = "light";
console.log(mood);
// → light
mood = "dark";
console.log(mood);
// → dark
```

You should imagine bindings as tentacles, rather than boxes. They do not *contain* values; they *grasp* them—two bindings can refer to the same value. A program can access only the values that it still has a reference to. When you need to remember something, you grow a tentacle to hold on to it or you reattach one of your existing tentacles to it.

Let's look at another example. To remember the number of dollars that Luigi still owes you, you create a binding. And then when he pays back $35, you give this binding a new value.

```
let luigisDebt = 140;
luigisDebt = luigisDebt - 35;
console.log(luigisDebt);
// → 105
```

When you define a binding without giving it a value, the tentacle has nothing to grasp, so it ends in thin air. If you ask for the value of an empty binding, you'll get the value `undefined`.

A single `let` statement may define multiple bindings. The definitions must be separated by commas.

```
let one = 1, two = 2;
console.log(one + two);
// → 3
```

The words `var` and `const` can also be used to create bindings, in a way similar to `let`.

```
var name = "Ayda";
const greeting = "Hello ";
console.log(greeting + name);
// → Hello Ayda
```

The first, `var` (short for "variable"), is the way bindings were declared in pre-2015 JavaScript. I'll get back to the precise way it differs from `let` in the next

chapter. For now, remember that it mostly does the same thing, but we'll rarely use it in this book because it has some confusing properties.

The word `const` stands for *constant*. It defines a constant binding, which points at the same value for as long as it lives. This is useful for bindings that give a name to a value so that you can easily refer to it later.

Binding names

Binding names can be any word. Digits can be part of binding names—`catch22` is a valid name, for example—but the name must not start with a digit. A binding name may include dollar signs (`$`) or underscores (`_`) but no other punctuation or special characters.

Words with a special meaning, such as `let`, are *keywords*, and they may not be used as binding names. There are also a number of words that are "reserved for use" in future versions of JavaScript, which also can't be used as binding names. The full list of keywords and reserved words is rather long.

```
break case catch class const continue debugger default
delete do else enum export extends false finally for
function if implements import interface in instanceof let
new package private protected public return static super
switch this throw true try typeof var void while with yield
```

Don't worry about memorizing this list. When creating a binding produces an unexpected syntax error, see whether you're trying to define a reserved word.

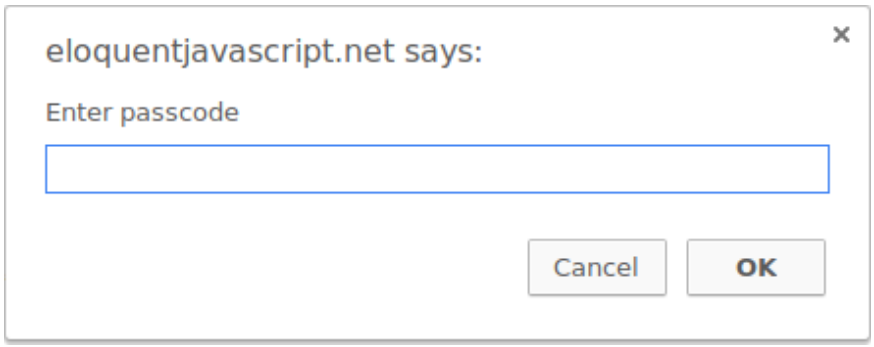
The environment

The collection of bindings and their values that exist at a given time is called the *environment*. When a program starts up, this environment is not empty. It always contains bindings that are part of the language standard, and most of the time, it also has bindings that provide ways to interact with the surrounding system. For example, in a browser, there are functions to interact with the currently loaded website and to read mouse and keyboard input.

Functions

A lot of the values provided in the default environment have the type *function*. A function is a piece of program wrapped in a value. Such values can be *applied* in order to run the wrapped program. For example, in a browser environment, the binding `prompt` holds a function that shows a little dialog box asking for user input. It is used like this:

```
prompt("Enter passcode");
```



Executing a function is called *invoking*, *calling*, or *applying* it. You can call a function by putting parentheses after an expression that produces a function value. Usually you'll directly use the name of the binding that holds the function. The values between the parentheses are given to the program inside the function. In the example, the `prompt` function uses the string that we give it as the text to show in the dialog box. Values given to functions are called *arguments*. Different functions might need a different number or different types of arguments.

The `prompt` function isn't used much in modern web programming, mostly because you have no control over the way the resulting dialog looks, but can be helpful in toy programs and experiments.

The console.log function

In the examples, I used `console.log` to output values. Most JavaScript systems (including all modern web browsers and Node.js) provide a `console.log` function that writes out its arguments to *some* text output device. In browsers, the output lands in the JavaScript console. This part of the

browser interface is hidden by default, but most browsers open it when you press F12 or, on a Mac, COMMAND-OPTION-I. If that does not work, search through the menus for an item named Developer Tools or similar.

When running the examples (or your own code) on the pages of this book, `console.log` output will be shown after the example, instead of in the browser's JavaScript console.

```
let x = 30;
console.log("the value of x is", x);
// → the value of x is 30
```

Though binding names cannot contain period characters, `console.log` does have one. This is because `console.log` isn't a simple binding. It is actually an expression that retrieves the `log` property from the value held by the `console` binding. We'll find out exactly what this means in Chapter 4.

Return values

Showing a dialog box or writing text to the screen is a *side effect*. A lot of functions are useful because of the side effects they produce. Functions may also produce values, in which case they don't need to have a side effect to be useful. For example, the function `Math.max` takes any amount of number arguments and gives back the greatest.

```
console.log(Math.max(2, 4));
// → 4
```

When a function produces a value, it is said to *return* that value. Anything that produces a value is an expression in JavaScript, which means function calls can be used within larger expressions. Here a call to `Math.min`, which is the opposite of `Math.max`, is used as part of a plus expression:

```
console.log(Math.min(2, 4) + 100);
// → 102
```

The next chapter explains how to write your own functions.

Control flow

When your program contains more than one statement, the statements are executed as if they are a story, from top to bottom. This example program has two statements. The first one asks the user for a number, and the second, which is executed after the first, shows the square of that number.

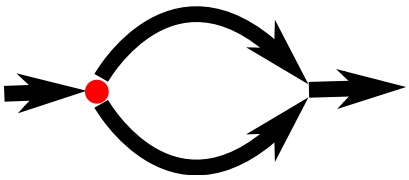
```
let theNumber = Number(prompt("Pick a number"));
console.log("Your number is the square root of " +
            theNumber * theNumber);
```

The function `Number` converts a value to a number. We need that conversion because the result of `prompt` is a string value, and we want a number. There are similar functions called `String` and `Boolean` that convert values to those types.

Here is the rather trivial schematic representation of straight-line control flow:

Conditional execution

Not all programs are straight roads. We may, for example, want to create a branching road, where the program takes the proper branch based on the situation at hand. This is called *conditional execution*.



Conditional execution is created with the `if` keyword in JavaScript. In the simple case, we want some code to be executed if, and only if, a certain condition holds. We might, for example, want to show the square of the input only if the input is actually a number.

```
let theNumber = Number(prompt("Pick a number"));
if (!Number.isNaN(theNumber)) {
```

```
  console.log("Your number is the square root of " +
              theNumber * theNumber);
}
```

With this modification, if you enter “parrot”, no output is shown.

The `if` keyword executes or skips a statement depending on the value of a Boolean expression. The deciding expression is written after the keyword, between parentheses, followed by the statement to execute.

The `Number.isNaN` function is a standard JavaScript function that returns `true` only if the argument it is given is `NaN`. The `Number` function happens to return `NaN` when you give it a string that doesn’t represent a valid number. Thus, the condition translates to “unless `theNumber` is not-a-number, do this”.

The statement after the `if` is wrapped in braces (`{` and `}`) in this example. The braces can be used to group any number of statements into a single statement, called a *block*. You could also have omitted them in this case, since they hold only a single statement, but to avoid having to think about whether they are needed, most JavaScript programmers use them in every wrapped statement like this. We’ll mostly follow that convention in this book, except for the occasional one-liner.

```
if (1 + 1 == 2) console.log("It's true");
// → It's true
```

You often won’t just have code that executes when a condition holds true, but also code that handles the other case. This alternate path is represented by the second arrow in the diagram. You can use the `else` keyword, together with `if`, to create two separate, alternative execution paths.

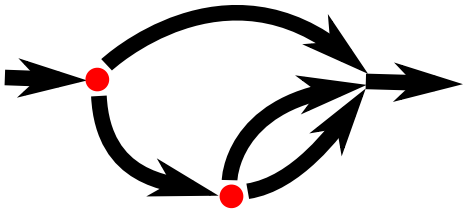
```
let theNumber = Number(prompt("Pick a number"));
if (!Number.isNaN(theNumber)) {
  console.log("Your number is the square root of " +
              theNumber * theNumber);
} else {
  console.log("Hey. Why didn't you give me a number?");
}
```

If you have more than two paths to choose from, you can "chain" multiple `if`/`else` pairs together. Here's an example:

```
let num = Number(prompt("Pick a number"));

if (num < 10) {
  console.log("Small");
} else if (num < 100) {
  console.log("Medium");
} else {
  console.log("Large");
}
```

The program will first check whether `num` is less than 10. If it is, it chooses that branch, shows `"Small"`, and is done. If it isn't, it takes the `else` branch, which itself contains a second `if`. If the second condition (`< 100`) holds, that means the number is between 10 and 100, and `"Medium"` is shown. If it doesn't, the second and last `else` branch is chosen.

The schema for this program looks something like this:

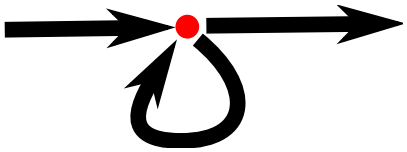


WHILE AND DO LOOPS

Consider a program that outputs all even numbers from 0 to 12. One way to write this is as follows:

```
console.log(0);
console.log(2);
console.log(4);
console.log(6);
console.log(8);
console.log(10);
console.log(12);
```

That works, but the idea of writing a program is to make something *less* work, not more. If we needed all even numbers less than 1,000, this approach would be unworkable. What we need is a way to run a piece of code multiple times. This form of control flow is called a *loop*.



Looping control flow allows us to go back to some point in the program where we were before and repeat it with our current program state. If we combine this with a binding that counts, we can do something like this:

```
let number = 0;
while (number <= 12) {
  console.log(number);
  number = number + 2;
}
// → 0
// → 2
//   … etcetera
```

A statement starting with the keyword `while` creates a loop. The word `while` is followed by an expression in parentheses and then a statement, much like `if`. The loop keeps entering that statement as long as the expression produces a value that gives `true` when converted to Boolean.

The `number` binding demonstrates the way a binding can track the progress of a program. Every time the loop repeats, `number` gets a value that is 2 more than its previous value. At the beginning of every repetition, it is compared with the number 12 to decide whether the program's work is finished.

As an example that actually does something useful, we can now write a program that calculates and shows the value of 2^{10} (2 to the 10th power). We use two bindings: one to keep track of our result and one to count how often we

have multiplied this result by 2. The loop tests whether the second binding has reached 10 yet and, if not, updates both bindings.

```
let result = 1;
let counter = 0;
while (counter < 10) {
  result = result * 2;
  counter = counter + 1;
}
console.log(result);
// → 1024
```

The counter could also have started at `1` and checked for `<= 10`, but for reasons that will become apparent in Chapter 4, it is a good idea to get used to counting from 0.

A `do` loop is a control structure similar to a `while` loop. It differs only on one point: a `do` loop always executes its body at least once, and it starts testing whether it should stop only after that first execution. To reflect this, the test appears after the body of the loop.

```
let yourName;
do {
  yourName = prompt("Who are you?");
} while (!yourName);
console.log(yourName);
```

This program will force you to enter a name. It will ask again and again until it gets something that is not an empty string. Applying the `!` operator will convert a value to Boolean type before negating it, and all strings except `""` convert to `true`. This means the loop continues going round until you provide a non-empty name.

Indenting Code

In the examples, I've been adding spaces in front of statements that are part of some larger statement. These spaces are not required—the computer will accept the program just fine without them. In fact, even the line breaks in

programs are optional. You could write a program as a single long line if you felt like it.

The role of this indentation inside blocks is to make the structure of the code stand out. In code where new blocks are opened inside other blocks, it can become hard to see where one block ends and another begins. With proper indentation, the visual shape of a program corresponds to the shape of the blocks inside it. I like to use two spaces for every open block, but tastes differ—some people use four spaces, and some people use tab characters. The important thing is that each new block adds the same amount of space.

```
if (false != true) {
  console.log("That makes sense.");
  if (1 < 2) {
    console.log("No surprise there.");
  }
}
```

Most code editor programs (including the one in this book) will help by automatically indenting new lines the proper amount.

FOR LOOPS

Many loops follow the pattern shown in the `while` examples. First a "counter" binding is created to track the progress of the loop. Then comes a `while` loop, usually with a test expression that checks whether the counter has reached its end value. At the end of the loop body, the counter is updated to track progress.

Because this pattern is so common, JavaScript and similar languages provide a slightly shorter and more comprehensive form, the `for` loop.

```
for (let number = 0; number <= 12; number = number + 2) {
  console.log(number);
}
// → 0
// → 2
//   … etcetera
```

This program is exactly equivalent to the earlier even-number-printing example. The only change is that all the statements that are related to the "state" of the loop are grouped together after `for`.

The parentheses after a `for` keyword must contain two semicolons. The part before the first semicolon *initializes* the loop, usually by defining a binding. The second part is the expression that *checks* whether the loop must continue. The final part *updates* the state of the loop after every iteration. In most cases, this is shorter and clearer than a `while` construct.

This is the code that computes 2^{10} using `for` instead of `while`:

```
let result = 1;
for (let counter = 0; counter < 10; counter = counter + 1) {
  result = result * 2;
}
console.log(result);
// → 1024
```

Breaking Out of a Loop

Having the looping condition produce `false` is not the only way a loop can finish. There is a special statement called `break` that has the effect of immediately jumping out of the enclosing loop.

This program illustrates the `break` statement. It finds the first number that is both greater than or equal to 20 and divisible by 7.

```
for (let current = 20; ; current = current + 1) {
  if (current % 7 == 0) {
    console.log(current);
    break;
  }
}
// → 21
```

Using the remainder (`%`) operator is an easy way to test whether a number is divisible by another number. If it is, the remainder of their division is zero.

The `for` construct in the example does not have a part that checks for the end of the loop. This means that the loop will never stop unless the `break` statement inside is executed.

If you were to remove that `break` statement or you accidentally write an end condition that always produces `true`, your program would get stuck in an *infinite loop*. A program stuck in an infinite loop will never finish running, which is usually a bad thing.

If you create an infinite loop in one of the examples on these pages, you'll usually be asked whether you want to stop the script after a few seconds. If that fails, you will have to close the tab that you're working in, or on some browsers close your whole browser, to recover.

The `continue` keyword is similar to `break`, in that it influences the progress of a loop. When `continue` is encountered in a loop body, control jumps out of the body and continues with the loop's next iteration.

Updating bindings succinctly

Especially when looping, a program often needs to "update" a binding to hold a value based on that binding's previous value.

```
counter = counter + 1;
```

JavaScript provides a shortcut for this.

```
counter += 1;
```

Similar shortcuts work for many other operators, such as `result *= 2` to double `result` or `counter -= 1` to count downward.

This allows us to shorten our counting example a little more.

```
for (let number = 0; number <= 12; number += 2) {
  console.log(number);
}
```

For `counter += 1` and `counter -= 1`, there are even shorter equivalents: `counter++` and `counter--`.

Dispatching on a value with switch

It is not uncommon for code to look like this:

```
if (x == "value1") action1();
else if (x == "value2") action2();
else if (x == "value3") action3();
else defaultAction();
```

There is a construct called `switch` that is intended to express such a "dispatch" in a more direct way. Unfortunately, the syntax JavaScript uses for this (which it inherited from the C/Java line of programming languages) is somewhat awkward—a chain of `if` statements may look better. Here is an example:

```
switch (prompt("What is the weather like?")) {
  case "rainy":
    console.log("Remember to bring an umbrella.");
    break;
  case "sunny":
    console.log("Dress lightly.");
  case "cloudy":
    console.log("Go outside.");
    break;
  default:
    console.log("Unknown weather type!");
    break;
}
```

You may put any number of `case` labels inside the block opened by `switch`. The program will start executing at the label that corresponds to the value that `switch` was given, or at `default` if no matching value is found. It will continue executing, even across other labels, until it reaches a `break` statement. In some cases, such as the `"sunny"` case in the example, this can be used to share some code between cases (it recommends going outside for both sunny and cloudy

weather). But be careful—it is easy to forget such a `break`, which will cause the program to execute code you do not want executed.

Capitalization

Binding names may not contain spaces, yet it is often helpful to use multiple words to clearly describe what the binding represents. These are pretty much your choices for writing a binding name with several words in it:

```
fuzzylittleturtle
fuzzy_little_turtle
FuzzyLittleTurtle
fuzzyLittleTurtle
```

The first style can be hard to read. I rather like the look of the underscores, though that style is a little painful to type. The standard JavaScript functions, and most JavaScript programmers, follow the bottom style—they capitalize every word except the first. It is not hard to get used to little things like that, and code with mixed naming styles can be jarring to read, so we follow this convention.

In a few cases, such as the `Number` function, the first letter of a binding is also capitalized. This was done to mark this function as a constructor. What a constructor is will become clear in Chapter 6. For now, the important thing is not to be bothered by this apparent lack of consistency.

Comments

Often, raw code does not convey all the information you want a program to convey to human readers, or it conveys it in such a cryptic way that people might not understand it. At other times, you might just want to include some related thoughts as part of your program. This is what *comments* are for.

A comment is a piece of text that is part of a program but is completely ignored by the computer. JavaScript has two ways of writing comments. To write a single-line comment, you can use two slash characters (`//`) and then the comment text after it.

```
let accountBalance = calculateBalance(account);
// It's a green hollow where a river sings
accountBalance.adjust();
// Madly catching white tatters in the grass.
let report = new Report();
// Where the sun on the proud mountain rings:
addToReport(accountBalance, report);
// It's a little valley, foaming like light in a glass.
```

A `//` comment goes only to the end of the line. A section of text between `/*` and `*/` will be ignored in its entirety, regardless of whether it contains line breaks. This is useful for adding blocks of information about a file or a chunk of program.

```
/*
  I first found this number scrawled on the back of an old noteboo
  Since then, it has often dropped by, showing up in phone numbers
  and the serial numbers of products that I've bought. It obviousl
  likes me, so I've decided to keep it.
*/
const myNumber = 11213;
```

Summary

You now know that a program is built out of statements, which themselves sometimes contain more statements. Statements tend to contain expressions, which themselves can be built out of smaller expressions.

Putting statements after one another gives you a program that is executed from top to bottom. You can introduce disturbances in the flow of control by using conditional (`if`, `else`, and `switch`) and looping (`while`, `do`, and `for`) statements.

Bindings can be used to file pieces of data under a name, and they are useful for tracking state in your program. The environment is the set of bindings that are defined. JavaScript systems always put a number of useful standard bindings into your environment.

Functions are special values that encapsulate a piece of program. You can invoke them by writing `functionName(argument1, argument2)`. Such a function call is an expression and may produce a value.

Exercises

If you are unsure how to test your solutions to the exercises, refer to the Introduction.

Each exercise starts with a problem description. Read this description and try to solve the exercise. If you run into problems, consider reading the hints after the exercise. Full solutions to the exercises are not included in this book, but you can find them online at *https://eloquentjavascript.net/code*. If you want to learn something from the exercises, I recommend looking at the solutions only after you've solved the exercise, or at least after you've attacked it long and hard enough to have a slight headache.

Looping a triangle

Write a loop that makes seven calls to `console.log` to output the following triangle:

```
#
##
###
####
#####
######
#######
```

It may be useful to know that you can find the length of a string by writing `.length` after it.

```
let abc = "abc";
console.log(abc.length);
// → 3
```

Most exercises contain a piece of code that you can modify to solve the exercise. Remember that you can click code blocks to edit them.

```
// Your code here.
```

» Display hints...

FizzBuzz

Write a program that uses `console.log` to print all the numbers from 1 to 100, with two exceptions. For numbers divisible by 3, print `"Fizz"` instead of the number, and for numbers divisible by 5 (and not 3), print `"Buzz"` instead.

When you have that working, modify your program to print `"FizzBuzz"` for numbers that are divisible by both 3 and 5 (and still print `"Fizz"` or `"Buzz"` for numbers divisible by only one of those).

(This is actually an interview question that has been claimed to weed out a significant percentage of programmer candidates. So if you solved it, your labor market value just went up.)

```
// Your code here.
```

» Display hints...

Chessboard

Write a program that creates a string that represents an 8×8 grid, using newline characters to separate lines. At each position of the grid there is either a space or a "#" character. The characters should form a chessboard.

Passing this string to `console.log` should show something like this:

```
 # # # #
# # # #
 # # # #
# # # #
 # # # #
# # # #
```

```
 # # # #
# # # #
```

When you have a program that generates this pattern, define a binding `size = 8` and change the program so that it works for any `size`, outputting a grid of the given width and height.

```
// Your code here.
```

» Display hints...

CHAPTER 3

FUNCTIONS

> "People think that computer science is the art of geniuses but the actual reality is the opposite, just many people doing things that build on each other, like a wall of mini stones."
>
> — Donald Knuth

Functions are the bread and butter of JavaScript programming. The concept of wrapping a piece of program in a value has many uses. It gives us a way to structure larger programs, to reduce repetition, to associate names with subprograms, and to isolate these subprograms from each other.

The most obvious application of functions is defining new vocabulary. Creating new words in prose is usually bad style. But in programming, it is indispensable.

Typical adult English speakers have some 20,000 words in their vocabulary. Few programming languages come with 20,000 commands built in. And the vocabulary that *is* available tends to be more precisely defined, and thus less

flexible, than in human language. Therefore, we usually *have* to introduce new concepts to avoid repeating ourselves too much.

Defining a function

A function definition is a regular binding where the value of the binding is a function. For example, this code defines `square` to refer to a function that produces the square of a given number:

```
const square = function(x) {
  return x * x;
};

console.log(square(12));
// → 144
```

edit & run code by clicking it

A function is created with an expression that starts with the keyword `function`. Functions have a set of *parameters* (in this case, only `x`) and a *body*, which contains the statements that are to be executed when the function is called. The function body of a function created this way must always be wrapped in braces, even when it consists of only a single statement.

A function can have multiple parameters or no parameters at all. In the following example, `makeNoise` does not list any parameter names, whereas `power` lists two:

```
const makeNoise = function() {
  console.log("Pling!");
};

makeNoise();
// → Pling!

const power = function(base, exponent) {
  let result = 1;
  for (let count = 0; count < exponent; count++) {
    result *= base;
  }
```

```
  return result;
};

console.log(power(2, 10));
// → 1024
```

Some functions produce a value, such as `power` and `square`, and some don't, such as `makeNoise`, whose only result is a side effect. A `return` statement determines the value the function returns. When control comes across such a statement, it immediately jumps out of the current function and gives the returned value to the code that called the function. A `return` keyword without an expression after it will cause the function to return `undefined`. Functions that don't have a `return` statement at all, such as `makeNoise`, similarly return `undefined`.

Parameters to a function behave like regular bindings, but their initial values are given by the *caller* of the function, not the code in the function itself.

Bindings and scopes

Each binding has a *scope*, which is the part of the program in which the binding is visible. For bindings defined outside of any function or block, the scope is the whole program—you can refer to such bindings wherever you want. These are called *global*.

But bindings created for function parameters or declared inside a function can be referenced only in that function, so they are known as *local* bindings. Every time the function is called, new instances of these bindings are created. This provides some isolation between functions—each function call acts in its own little world (its local environment) and can often be understood without knowing a lot about what's going on in the global environment.

Bindings declared with `let` and `const` are in fact local to the *block* that they are declared in, so if you create one of those inside of a loop, the code before and after the loop cannot "see" it. In pre-2015 JavaScript, only functions created new scopes, so old-style bindings, created with the `var` keyword, are

visible throughout the whole function that they appear in—or throughout the global scope, if they are not in a function.

```
let x = 10;
if (true) {
  let y = 20;
  var z = 30;
  console.log(x + y + z);
  // → 60
}
// y is not visible here
console.log(x + z);
// → 40
```

Each scope can "look out" into the scope around it, so `x` is visible inside the block in the example. The exception is when multiple bindings have the same name—in that case, code can see only the innermost one. For example, when the code inside the `halve` function refers to `n`, it is seeing its *own* `n`, not the global `n`.

```
const halve = function(n) {
  return n / 2;
};

let n = 10;
console.log(halve(100));
// → 50
console.log(n);
// → 10
```

Nested scope

JavaScript distinguishes not just *global* and *local* bindings. Blocks and functions can be created inside other blocks and functions, producing multiple degrees of locality.

For example, this function—which outputs the ingredients needed to make a batch of hummus—has another function inside it:

```
const hummus = function(factor) {
  const ingredient = function(amount, unit, name) {
    let ingredientAmount = amount * factor;
    if (ingredientAmount > 1) {
      unit += "s";
    }
    console.log(`${ingredientAmount} ${unit} ${name}`);
  };
  ingredient(1, "can", "chickpeas");
  ingredient(0.25, "cup", "tahini");
  ingredient(0.25, "cup", "lemon juice");
  ingredient(1, "clove", "garlic");
  ingredient(2, "tablespoon", "olive oil");
  ingredient(0.5, "teaspoon", "cumin");
};
```

The code inside the `ingredient` function can see the `factor` binding from the outer function. But its local bindings, such as `unit` or `ingredientAmount`, are not visible in the outer function.

The set of bindings visible inside a block is determined by the place of that block in the program text. Each local scope can also see all the local scopes that contain it, and all scopes can see the global scope. This approach to binding visibility is called *lexical scoping*.

Functions as values

A function binding usually simply acts as a name for a specific piece of the program. Such a binding is defined once and never changed. This makes it easy to confuse the function and its name.

But the two are different. A function value can do all the things that other values can do—you can use it in arbitrary expressions, not just call it. It is possible to store a function value in a new binding, pass it as an argument to a function, and so on. Similarly, a binding that holds a function is still just a regular binding and can, if not constant, be assigned a new value, like so:

```
let launchMissiles = function() {
  missileSystem.launch("now");
};
if (safeMode) {
  launchMissiles = function() {/* do nothing */};
}
```

In Chapter 5, we will discuss the interesting things that can be done by passing around function values to other functions.

Declaration notation

There is a slightly shorter way to create a function binding. When the `function` keyword is used at the start of a statement, it works differently.

```
function square(x) {
  return x * x;
}
```

This is a function *declaration*. The statement defines the binding `square` and points it at the given function. It is slightly easier to write and doesn't require a semicolon after the function.

There is one subtlety with this form of function definition.

```
console.log("The future says:", future());

function future() {
  return "You'll never have flying cars";
}
```

The preceding code works, even though the function is defined *below* the code that uses it. Function declarations are not part of the regular top-to-bottom flow of control. They are conceptually moved to the top of their scope and can be used by all the code in that scope. This is sometimes useful because it offers the freedom to order code in a way that seems meaningful, without worrying about having to define all functions before they are used.

Arrow functions

There's a third notation for functions, which looks very different from the others. Instead of the `function` keyword, it uses an arrow (`=>`) made up of an equal sign and a greater-than character (not to be confused with the greater-than-or-equal operator, which is written `>=`).

```
const power = (base, exponent) => {
  let result = 1;
  for (let count = 0; count < exponent; count++) {
    result *= base;
  }
  return result;
};
```

The arrow comes *after* the list of parameters and is followed by the function's body. It expresses something like "this input (the parameters) produces this result (the body)".

When there is only one parameter name, you can omit the parentheses around the parameter list. If the body is a single expression, rather than a block in braces, that expression will be returned from the function. So, these two definitions of `square` do the same thing:

```
const square1 = (x) => { return x * x; };
const square2 = x => x * x;
```

When an arrow function has no parameters at all, its parameter list is just an empty set of parentheses.

```
const horn = () => {
  console.log("Toot");
};
```

There's no deep reason to have both arrow functions and `function` expressions in the language. Apart from a minor detail, which we'll discuss in Chapter 6, they do the same thing. Arrow functions were added in 2015, mostly

to make it possible to write small function expressions in a less verbose way. We'll be using them a lot in Chapter 5.

The call stack

The way control flows through functions is somewhat involved. Let's take a closer look at it. Here is a simple program that makes a few function calls:

```
function greet(who) {
  console.log("Hello " + who);
}
greet("Harry");
console.log("Bye");
```

A run through this program goes roughly like this: the call to `greet` causes control to jump to the start of that function (line 2). The function calls `console.log`, which takes control, does its job, and then returns control to line 2. There it reaches the end of the `greet` function, so it returns to the place that called it, which is line 4. The line after that calls `console.log` again. After that returns, the program reaches its end.

We could show the flow of control schematically like this:

```
not in function
   in greet
        in console.log
   in greet
not in function
   in console.log
not in function
```

Because a function has to jump back to the place that called it when it returns, the computer must remember the context from which the call happened. In one case, `console.log` has to return to the `greet` function when it is done. In the other case, it returns to the end of the program.

The place where the computer stores this context is the *call stack*. Every time a function is called, the current context is stored on top of this stack. When a

function returns, it removes the top context from the stack and uses that context to continue execution.

Storing this stack requires space in the computer's memory. When the stack grows too big, the computer will fail with a message like "out of stack space" or "too much recursion". The following code illustrates this by asking the computer a really hard question that causes an infinite back-and-forth between two functions. Rather, it *would* be infinite, if the computer had an infinite stack. As it is, we will run out of space, or "blow the stack".

```
function chicken() {
  return egg();
}
function egg() {
  return chicken();
}
console.log(chicken() + " came first.");
// → ??
```

Optional Arguments

The following code is allowed and executes without any problem:

```
function square(x) { return x * x; }
console.log(square(4, true, "hedgehog"));
// → 16
```

We defined `square` with only one parameter. Yet when we call it with three, the language doesn't complain. It ignores the extra arguments and computes the square of the first one.

JavaScript is extremely broad-minded about the number of arguments you pass to a function. If you pass too many, the extra ones are ignored. If you pass too few, the missing parameters get assigned the value `undefined`.

The downside of this is that it is possible—likely, even—that you'll accidentally pass the wrong number of arguments to functions. And no one will tell you about it.

The upside is that this behavior can be used to allow a function to be called with different numbers of arguments. For example, this `minus` function tries to imitate the `-` operator by acting on either one or two arguments:

```
function minus(a, b) {
  if (b === undefined) return -a;
  else return a - b;
}

console.log(minus(10));
// → -10
console.log(minus(10, 5));
// → 5
```

If you write an `=` operator after a parameter, followed by an expression, the value of that expression will replace the argument when it is not given.

For example, this version of `power` makes its second argument optional. If you don't provide it or pass the value `undefined`, it will default to two, and the function will behave like `square`.

```
function power(base, exponent = 2) {
  let result = 1;
  for (let count = 0; count < exponent; count++) {
    result *= base;
  }
  return result;
}

console.log(power(4));
// → 16
console.log(power(2, 6));
// → 64
```

In the next chapter, we will see a way in which a function body can get at the whole list of arguments it was passed. This is helpful because it makes it possible for a function to accept any number of arguments. For example, `console.log` does this—it outputs all of the values it is given.

```
console.log("C", "O", 2);
// → C O 2
```

Closure

The ability to treat functions as values, combined with the fact that local bindings are re-created every time a function is called, brings up an interesting question. What happens to local bindings when the function call that created them is no longer active?

The following code shows an example of this. It defines a function, `wrapValue`, that creates a local binding. It then returns a function that accesses and returns this local binding.

```
function wrapValue(n) {
  let local = n;
  return () => local;
}

let wrap1 = wrapValue(1);
let wrap2 = wrapValue(2);
console.log(wrap1());
// → 1
console.log(wrap2());
// → 2
```

This is allowed and works as you'd hope—both instances of the binding can still be accessed. This situation is a good demonstration of the fact that local bindings are created anew for every call, and different calls can't trample on one another's local bindings.

This feature—being able to reference a specific instance of a local binding in an enclosing scope—is called *closure*. A function that references bindings from local scopes around it is called *a* closure. This behavior not only frees you from having to worry about lifetimes of bindings but also makes it possible to use function values in some creative ways.

With a slight change, we can turn the previous example into a way to create functions that multiply by an arbitrary amount.

```
function multiplier(factor) {
  return number => number * factor;
}

let twice = multiplier(2);
console.log(twice(5));
// → 10
```

The explicit `local` binding from the `wrapValue` example isn't really needed since a parameter is itself a local binding.

Thinking about programs like this takes some practice. A good mental model is to think of function values as containing both the code in their body and the environment in which they are created. When called, the function body sees the environment in which it was created, not the environment in which it is called.

In the example, `multiplier` is called and creates an environment in which its `factor` parameter is bound to 2. The function value it returns, which is stored in `twice`, remembers this environment. So when that is called, it multiplies its argument by 2.

Recursion

It is perfectly okay for a function to call itself, as long as it doesn't do it so often that it overflows the stack. A function that calls itself is called *recursive*. Recursion allows some functions to be written in a different style. Take, for example, this alternative implementation of `power`:

```
function power(base, exponent) {
  if (exponent == 0) {
    return 1;
  } else {
    return base * power(base, exponent - 1);
  }
}
```

```
console.log(power(2, 3));
// → 8
```

This is rather close to the way mathematicians define exponentiation and arguably describes the concept more clearly than the looping variant. The function calls itself multiple times with ever smaller exponents to achieve the repeated multiplication.

But this implementation has one problem: in typical JavaScript implementations, it's about three times slower than the looping version. Running through a simple loop is generally cheaper than calling a function multiple times.

The dilemma of speed versus elegance is an interesting one. You can see it as a kind of continuum between human-friendliness and machine-friendliness. Almost any program can be made faster by making it bigger and more convoluted. The programmer has to decide on an appropriate balance.

In the case of the `power` function, the inelegant (looping) version is still fairly simple and easy to read. It doesn't make much sense to replace it with the recursive version. Often, though, a program deals with such complex concepts that giving up some efficiency in order to make the program more straightforward is helpful.

Worrying about efficiency can be a distraction. It's yet another factor that complicates program design, and when you're doing something that's already difficult, that extra thing to worry about can be paralyzing.

Therefore, always start by writing something that's correct and easy to understand. If you're worried that it's too slow—which it usually isn't since most code simply isn't executed often enough to take any significant amount of time—you can measure afterward and improve it if necessary.

Recursion is not always just an inefficient alternative to looping. Some problems really are easier to solve with recursion than with loops. Most often

these are problems that require exploring or processing several “branches”, each of which might branch out again into even more branches.

Consider this puzzle: by starting from the number 1 and repeatedly either adding 5 or multiplying by 3, an infinite set of numbers can be produced. How would you write a function that, given a number, tries to find a sequence of such additions and multiplications that produces that number?

For example, the number 13 could be reached by first multiplying by 3 and then adding 5 twice, whereas the number 15 cannot be reached at all.

Here is a recursive solution:

```
function findSolution(target) {
  function find(current, history) {
    if (current == target) {
      return history;
    } else if (current > target) {
      return null;
    } else {
      return find(current + 5, `(${history} + 5)`) ||
             find(current * 3, `(${history} * 3)`);
    }
  }
  return find(1, "1");
}

console.log(findSolution(24));
// → (((1 * 3) + 5) * 3)
```

Note that this program doesn’t necessarily find the *shortest* sequence of operations. It is satisfied when it finds any sequence at all.

It is okay if you don’t see how it works right away. Let’s work through it, since it makes for a great exercise in recursive thinking.

The inner function `find` does the actual recursing. It takes two arguments: the current number and a string that records how we reached this number. If it

finds a solution, it returns a string that shows how to get to the target. If no solution can be found starting from this number, it returns `null`.

To do this, the function performs one of three actions. If the current number is the target number, the current history is a way to reach that target, so it is returned. If the current number is greater than the target, there's no sense in further exploring this branch because both adding and multiplying will only make the number bigger, so it returns `null`. Finally, if we're still below the target number, the function tries both possible paths that start from the current number by calling itself twice, once for addition and once for multiplication. If the first call returns something that is not `null`, it is returned. Otherwise, the second call is returned, regardless of whether it produces a string or `null`.

To better understand how this function produces the effect we're looking for, let's look at all the calls to `find` that are made when searching for a solution for the number 13.

```
find(1, "1")
  find(6, "(1 + 5)")
    find(11, "((1 + 5) + 5)")
      find(16, "(((1 + 5) + 5) + 5)")
        too big
      find(33, "(((1 + 5) + 5) * 3)")
        too big
    find(18, "((1 + 5) * 3)")
      too big
  find(3, "(1 * 3)")
    find(8, "((1 * 3) + 5)")
      find(13, "(((1 * 3) + 5) + 5)")
        found!
```

The indentation indicates the depth of the call stack. The first time `find` is called, it starts by calling itself to explore the solution that starts with `(1 + 5)`. That call will further recurse to explore *every* continued solution that yields a number less than or equal to the target number. Since it doesn't find one that hits the target, it returns `null` back to the first call. There the `||` operator

causes the call that explores `(1 * 3)` to happen. This search has more luck—its first recursive call, through yet *another* recursive call, hits upon the target number. That innermost call returns a string, and each of the `||` operators in the intermediate calls passes that string along, ultimately returning the solution.

Growing functions

There are two more or less natural ways for functions to be introduced into programs.

The first is that you find yourself writing similar code multiple times. You'd prefer not to do that. Having more code means more space for mistakes to hide and more material to read for people trying to understand the program. So you take the repeated functionality, find a good name for it, and put it into a function.

The second way is that you find you need some functionality that you haven't written yet and that sounds like it deserves its own function. You'll start by naming the function, and then you'll write its body. You might even start writing code that uses the function before you actually define the function itself.

How difficult it is to find a good name for a function is a good indication of how clear a concept it is that you're trying to wrap. Let's go through an example.

We want to write a program that prints two numbers: the numbers of cows and chickens on a farm, with the words `Cows` and `Chickens` after them and zeros padded before both numbers so that they are always three digits long.

```
007 Cows
011 Chickens
```

This asks for a function of two arguments—the number of cows and the number of chickens. Let's get coding.

```
function printFarmInventory(cows, chickens) {
  let cowString = String(cows);
  while (cowString.length < 3) {
    cowString = "0" + cowString;
  }
  console.log(`${cowString} Cows`);
  let chickenString = String(chickens);
  while (chickenString.length < 3) {
    chickenString = "0" + chickenString;
  }
  console.log(`${chickenString} Chickens`);
}
printFarmInventory(7, 11);
```

Writing `.length` after a string expression will give us the length of that string. Thus, the `while` loops keep adding zeros in front of the number strings until they are at least three characters long.

Mission accomplished! But just as we are about to send the farmer the code (along with a hefty invoice), she calls and tells us she's also started keeping pigs, and couldn't we please extend the software to also print pigs?

We sure can. But just as we're in the process of copying and pasting those four lines one more time, we stop and reconsider. There has to be a better way. Here's a first attempt:

```
function printZeroPaddedWithLabel(number, label) {
  let numberString = String(number);
  while (numberString.length < 3) {
    numberString = "0" + numberString;
  }
  console.log(`${numberString} ${label}`);
}

function printFarmInventory(cows, chickens, pigs) {
  printZeroPaddedWithLabel(cows, "Cows");
  printZeroPaddedWithLabel(chickens, "Chickens");
  printZeroPaddedWithLabel(pigs, "Pigs");
}
```

```
printFarmInventory(7, 11, 3);
```

It works! But that name, `printZeroPaddedWithLabel`, is a little awkward. It conflates three things—printing, zero-padding, and adding a label—into a single function.

Instead of lifting out the repeated part of our program wholesale, let's try to pick out a single *concept*.

```
function zeroPad(number, width) {
  let string = String(number);
  while (string.length < width) {
    string = "0" + string;
  }
  return string;
}

function printFarmInventory(cows, chickens, pigs) {
  console.log(`${zeroPad(cows, 3)} Cows`);
  console.log(`${zeroPad(chickens, 3)} Chickens`);
  console.log(`${zeroPad(pigs, 3)} Pigs`);
}

printFarmInventory(7, 16, 3);
```

A function with a nice, obvious name like `zeroPad` makes it easier for someone who reads the code to figure out what it does. And such a function is useful in more situations than just this specific program. For example, you could use it to help print nicely aligned tables of numbers.

How smart and versatile *should* our function be? We could write anything, from a terribly simple function that can only pad a number to be three characters wide to a complicated generalized number-formatting system that handles fractional numbers, negative numbers, alignment of decimal dots, padding with different characters, and so on.

A useful principle is to not add cleverness unless you are absolutely sure you're going to need it. It can be tempting to write general "frameworks" for every bit of functionality you come across. Resist that urge. You won't get any real work done—you'll just be writing code that you never use.

Functions and side effects

Functions can be roughly divided into those that are called for their side effects and those that are called for their return value. (Though it is definitely also possible to both have side effects and return a value.)

The first helper function in the farm example, `printZeroPaddedWithLabel`, is called for its side effect: it prints a line. The second version, `zeroPad`, is called for its return value. It is no coincidence that the second is useful in more situations than the first. Functions that create values are easier to combine in new ways than functions that directly perform side effects.

A *pure* function is a specific kind of value-producing function that not only has no side effects but also doesn't rely on side effects from other code—for example, it doesn't read global bindings whose value might change. A pure function has the pleasant property that, when called with the same arguments, it always produces the same value (and doesn't do anything else). A call to such a function can be substituted by its return value without changing the meaning of the code. When you are not sure that a pure function is working correctly, you can test it by simply calling it and know that if it works in that context, it will work in any context. Nonpure functions tend to require more scaffolding to test.

Still, there's no need to feel bad when writing functions that are not pure or to wage a holy war to purge them from your code. Side effects are often useful. There'd be no way to write a pure version of `console.log`, for example, and `console.log` is good to have. Some operations are also easier to express in an efficient way when we use side effects, so computing speed can be a reason to avoid purity.

Summary

This chapter taught you how to write your own functions. The `function` keyword, when used as an expression, can create a function value. When used as a statement, it can be used to declare a binding and give it a function as its value. Arrow functions are yet another way to create functions.

```
// Define f to hold a function value
const f = function(a) {
  console.log(a + 2);
};

// Declare g to be a function
function g(a, b) {
  return a * b * 3.5;
}

// A less verbose function value
let h = a => a % 3;
```

A key aspect in understanding functions is understanding scopes. Each block creates a new scope. Parameters and bindings declared in a given scope are local and not visible from the outside. Bindings declared with `var` behave differently—they end up in the nearest function scope or the global scope.

Separating the tasks your program performs into different functions is helpful. You won't have to repeat yourself as much, and functions can help organize a program by grouping code into pieces that do specific things.

Exercises

Minimum

The previous chapter introduced the standard function `Math.min` that returns its smallest argument. We can build something like that now. Write a function `min` that takes two arguments and returns their minimum.

```
// Your code here.

console.log(min(0, 10));
```

```
// → 0
console.log(min(0, -10));
// → -10
```

» Display hints...

Recursion

We've seen that `%` (the remainder operator) can be used to test whether a number is even or odd by using `% 2` to see whether it's divisible by two. Here's another way to define whether a positive whole number is even or odd:

- Zero is even.
- One is odd.
- For any other number *N*, its evenness is the same as *N* - 2.

Define a recursive function `isEven` corresponding to this description. The function should accept a single parameter (a positive, whole number) and return a Boolean.

Test it on 50 and 75. See how it behaves on -1. Why? Can you think of a way to fix this?

```
// Your code here.

console.log(isEven(50));
// → true
console.log(isEven(75));
// → false
console.log(isEven(-1));
// → ??
```

» Display hints...

Bean counting

You can get the Nth character, or letter, from a string by writing `"string"[N]`. The returned value will be a string containing only one character (for example,

`"b"`). The first character has position 0, which causes the last one to be found at position `string.length - 1`. In other words, a two-character string has length 2, and its characters have positions 0 and 1.

Write a function `countBs` that takes a string as its only argument and returns a number that indicates how many uppercase “B” characters there are in the string.

Next, write a function called `countChar` that behaves like `countBs`, except it takes a second argument that indicates the character that is to be counted (rather than counting only uppercase “B” characters). Rewrite `countBs` to make use of this new function.

```
// Your code here.

console.log(countBs("BBC"));
// → 2
console.log(countChar("kakkerlak", "k"));
// → 4
```

» Display hints...

CHAPTER 4

DATA STRUCTURES: OBJECTS AND ARRAYS

> "On two occasions I have been asked, 'Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?' [...] I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question."
>
> — Charles Babbage, *Passages from the Life of a Philosopher (1864)*

Numbers, Booleans, and strings are the atoms that data structures are built from. Many types of information require more than one atom, though. *Objects* allow us to group values—including other objects—to build more complex structures.

The programs we have built so far have been limited by the fact that they were operating only on simple data types. This chapter will introduce basic data structures. By the end of it, you'll know enough to start writing useful programs.

The chapter will work through a more or less realistic programming example, introducing concepts as they apply to the problem at hand. The example code

will often build on functions and bindings that were introduced earlier in the text.

The weresquirrel

Every now and then, usually between 8 p.m. and 10 p.m., Jacques finds himself transforming into a small furry rodent with a bushy tail.

On one hand, Jacques is quite glad that he doesn't have classic lycanthropy. Turning into a squirrel does cause fewer problems than turning into a wolf. Instead of having to worry about accidentally eating the neighbor (*that* would be awkward), he worries about being eaten by the neighbor's cat. After two occasions where he woke up on a precariously thin branch in the crown of an oak, naked and disoriented, he has taken to locking the doors and windows of his room at night and putting a few walnuts on the floor to keep himself busy.

That takes care of the cat and tree problems. But Jacques would prefer to get rid of his condition entirely. The irregular occurrences of the transformation make him suspect that they might be triggered by something. For a while, he believed that it happened only on days when he had been near oak trees. But avoiding oak trees did not stop the problem.

Switching to a more scientific approach, Jacques has started keeping a daily log of everything he does on a given day and whether he changed form. With this data he hopes to narrow down the conditions that trigger the transformations.

The first thing he needs is a data structure to store this information.

Data sets

To work with a chunk of digital data, we'll first have to find a way to represent it in our machine's memory. Say, for example, that we want to represent a collection of the numbers 2, 3, 5, 7, and 11.

We could get creative with strings—after all, strings can have any length, so we can put a lot of data into them—and use `"2 3 5 7 11"` as our representation.

But this is awkward. You'd have to somehow extract the digits and convert them back to numbers to access them.

Fortunately, JavaScript provides a data type specifically for storing sequences of values. It is called an *array* and is written as a list of values between square brackets, separated by commas.

```
let listOfNumbers = [2, 3, 5, 7, 11];
console.log(listOfNumbers[2]);
// → 5
console.log(listOfNumbers[0]);
// → 2
console.log(listOfNumbers[2 - 1]);
// → 3
```

edit & run code by clicking it

The notation for getting at the elements inside an array also uses square brackets. A pair of square brackets immediately after an expression, with another expression inside of them, will look up the element in the left-hand expression that corresponds to the *index* given by the expression in the brackets.

The first index of an array is zero, not one. So the first element is retrieved with `listOfNumbers[0]`. Zero-based counting has a long tradition in technology and in certain ways makes a lot of sense, but it takes some getting used to. Think of the index as the amount of items to skip, counting from the start of the array.

Properties

We've seen a few suspicious-looking expressions like `myString.length` (to get the length of a string) and `Math.max` (the maximum function) in past chapters. These are expressions that access a *property* of some value. In the first case, we access the `length` property of the value in `myString`. In the second, we access the property named `max` in the `Math` object (which is a collection of mathematics-related constants and functions).

Almost all JavaScript values have properties. The exceptions are `null` and `undefined`. If you try to access a property on one of these nonvalues, you get an error.

```
null.length;
// → TypeError: null has no properties
```

The two main ways to access properties in JavaScript are with a dot and with square brackets. Both `value.x` and `value[x]` access a property on `value`—but not necessarily the same property. The difference is in how `x` is interpreted. When using a dot, the word after the dot is the literal name of the property. When using square brackets, the expression between the brackets is *evaluated* to get the property name. Whereas `value.x` fetches the property of `value` named "x", `value[x]` tries to evaluate the expression `x` and uses the result, converted to a string, as the property name.

So if you know that the property you are interested in is called *color*, you say `value.color`. If you want to extract the property named by the value held in the binding `i`, you say `value[i]`. Property names are strings. They can be any string, but the dot notation works only with names that look like valid binding names. So if you want to access a property named *2* or *John Doe*, you must use square brackets: `value[2]` or `value["John Doe"]`.

The elements in an array are stored as the array's properties, using numbers as property names. Because you can't use the dot notation with numbers and usually want to use a binding that holds the index anyway, you have to use the bracket notation to get at them.

The `length` property of an array tells us how many elements it has. This property name is a valid binding name, and we know its name in advance, so to find the length of an array, you typically write `array.length` because that's easier to write than `array["length"]`.

Methods

Both string and array values contain, in addition to the `length` property, a number of properties that hold function values.

```
let doh = "Doh";
console.log(typeof doh.toUpperCase);
// → function
console.log(doh.toUpperCase());
// → DOH
```

Every string has a `toUpperCase` property. When called, it will return a copy of the string in which all letters have been converted to uppercase. There is also `toLowerCase`, going the other way.

Interestingly, even though the call to `toUpperCase` does not pass any arguments, the function somehow has access to the string `"Doh"`, the value whose property we called. How this works is described in Chapter 6.

Properties that contain functions are generally called *methods* of the value they belong to, as in "`toUpperCase` is a method of a string".

This example demonstrates two methods you can use to manipulate arrays:

```
let sequence = [1, 2, 3];
sequence.push(4);
sequence.push(5);
console.log(sequence);
// → [1, 2, 3, 4, 5]
console.log(sequence.pop());
// → 5
console.log(sequence);
// → [1, 2, 3, 4]
```

The `push` method adds values to the end of an array, and the `pop` method does the opposite, removing the last value in the array and returning it.

These somewhat silly names are the traditional terms for operations on a *stack*. A stack, in programming, is a data structure that allows you to push values into it and pop them out again in the opposite order so that the thing that was

added last is removed first. These are common in programming—you might remember the function call stack from the previous chapter, which is an instance of the same idea.

Objects

Back to the weresquirrel. A set of daily log entries can be represented as an array. But the entries do not consist of just a number or a string—each entry needs to store a list of activities and a Boolean value that indicates whether Jacques turned into a squirrel or not. Ideally, we would like to group these together into a single value and then put those grouped values into an array of log entries.

Values of the type *object* are arbitrary collections of properties. One way to create an object is by using braces as an expression.

```
let day1 = {
  squirrel: false,
  events: ["work", "touched tree", "pizza", "running"]
};
console.log(day1.squirrel);
// → false
console.log(day1.wolf);
// → undefined
day1.wolf = false;
console.log(day1.wolf);
// → false
```

Inside the braces, there is a list of properties separated by commas. Each property has a name followed by a colon and a value. When an object is written over multiple lines, indenting it like in the example helps with readability. Properties whose names aren't valid binding names or valid numbers have to be quoted.

```
let descriptions = {
  work: "Went to work",
  "touched tree": "Touched a tree"
};
```

This means that braces have *two* meanings in JavaScript. At the start of a statement, they start a block of statements. In any other position, they describe an object. Fortunately, it is rarely useful to start a statement with an object in braces, so the ambiguity between these two is not much of a problem.

Reading a property that doesn't exist will give you the value `undefined`.

It is possible to assign a value to a property expression with the `=` operator. This will replace the property's value if it already existed or create a new property on the object if it didn't.

To briefly return to our tentacle model of bindings—property bindings are similar. They *grasp* values, but other bindings and properties might be holding onto those same values. You may think of objects as octopuses with any number of tentacles, each of which has a name tattooed on it.

The `delete` operator cuts off a tentacle from such an octopus. It is a unary operator that, when applied to an object property, will remove the named property from the object. This is not a common thing to do, but it is possible.

```
let anObject = {left: 1, right: 2};
console.log(anObject.left);
// → 1
delete anObject.left;
console.log(anObject.left);
// → undefined
console.log("left" in anObject);
// → false
console.log("right" in anObject);
// → true
```

The binary `in` operator, when applied to a string and an object, tells you whether that object has a property with that name. The difference between setting a property to `undefined` and actually deleting it is that, in the first case, the object still *has* the property (it just doesn't have a very interesting value), whereas in the second case the property is no longer present and `in` will return `false`.

To find out what properties an object has, you can use the `Object.keys` function. You give it an object, and it returns an array of strings—the object's property names.

```
console.log(Object.keys({x: 0, y: 0, z: 2}));
// → ["x", "y", "z"]
```

There's an `Object.assign` function that copies all properties from one object into another.

```
let objectA = {a: 1, b: 2};
Object.assign(objectA, {b: 3, c: 4});
console.log(objectA);
// → {a: 1, b: 3, c: 4}
```

Arrays, then, are just a kind of object specialized for storing sequences of things. If you evaluate `typeof []`, it produces `"object"`. You can see them as long, flat octopuses with all their tentacles in a neat row, labeled with numbers.

We will represent the journal that Jacques keeps as an array of objects.

```
let journal = [
  {events: ["work", "touched tree", "pizza",
            "running", "television"],
   squirrel: false},
  {events: ["work", "ice cream", "cauliflower",
            "lasagna", "touched tree", "brushed teeth"],
   squirrel: false},
  {events: ["weekend", "cycling", "break", "peanuts",
            "beer"],
   squirrel: true},
  /* and so on... */
];
```

MUTABILITY

We will get to actual programming *real* soon now. First there's one more piece of theory to understand.

We saw that object values can be modified. The types of values discussed in earlier chapters, such as numbers, strings, and Booleans, are all *immutable*—it is impossible to change values of those types. You can combine them and derive new values from them, but when you take a specific string value, that value will always remain the same. The text inside it cannot be changed. If you have a string that contains `"cat"`, it is not possible for other code to change a character in your string to make it spell `"rat"`.

Objects work differently. You *can* change their properties, causing a single object value to have different content at different times.

When we have two numbers, 120 and 120, we can consider them precisely the same number, whether or not they refer to the same physical bits. With objects, there is a difference between having two references to the same object and having two different objects that contain the same properties. Consider the following code:

```
let object1 = {value: 10};
let object2 = object1;
let object3 = {value: 10};

console.log(object1 == object2);
// → true
console.log(object1 == object3);
// → false

object1.value = 15;
console.log(object2.value);
// → 15
console.log(object3.value);
// → 10
```

The `object1` and `object2` bindings grasp the *same* object, which is why changing `object1` also changes the value of `object2`. They are said to have the same *identity*. The binding `object3` points to a different object, which initially contains the same properties as `object1` but lives a separate life.

Bindings can also be changeable or constant, but this is separate from the way their values behave. Even though number values don't change, you can use a `let` binding to keep track of a changing number by changing the value the binding points at. Similarly, though a `const` binding to an object can itself not be changed and will continue to point at the same object, the *contents* of that object might change.

```
const score = {visitors: 0, home: 0};
// This is okay
score.visitors = 1;
// This isn't allowed
score = {visitors: 1, home: 1};
```

When you compare objects with JavaScript's `==` operator, it compares by identity: it will produce `true` only if both objects are precisely the same value. Comparing different objects will return `false`, even if they have identical properties. There is no "deep" comparison operation built into JavaScript, which compares objects by contents, but it is possible to write it yourself (which is one of the exercises at the end of this chapter).

The lycanthrope's log

So, Jacques starts up his JavaScript interpreter and sets up the environment he needs to keep his journal.

```
let journal = [];

function addEntry(events, squirrel) {
  journal.push({events, squirrel});
}
```

Note that the object added to the journal looks a little odd. Instead of declaring properties like `events: events`, it just gives a property name. This is shorthand that means the same thing—if a property name in brace notation isn't followed by a value, its value is taken from the binding with the same name.

So then, every evening at 10 p.m.—or sometimes the next morning, after climbing down from the top shelf of his bookcase—Jacques records the day.

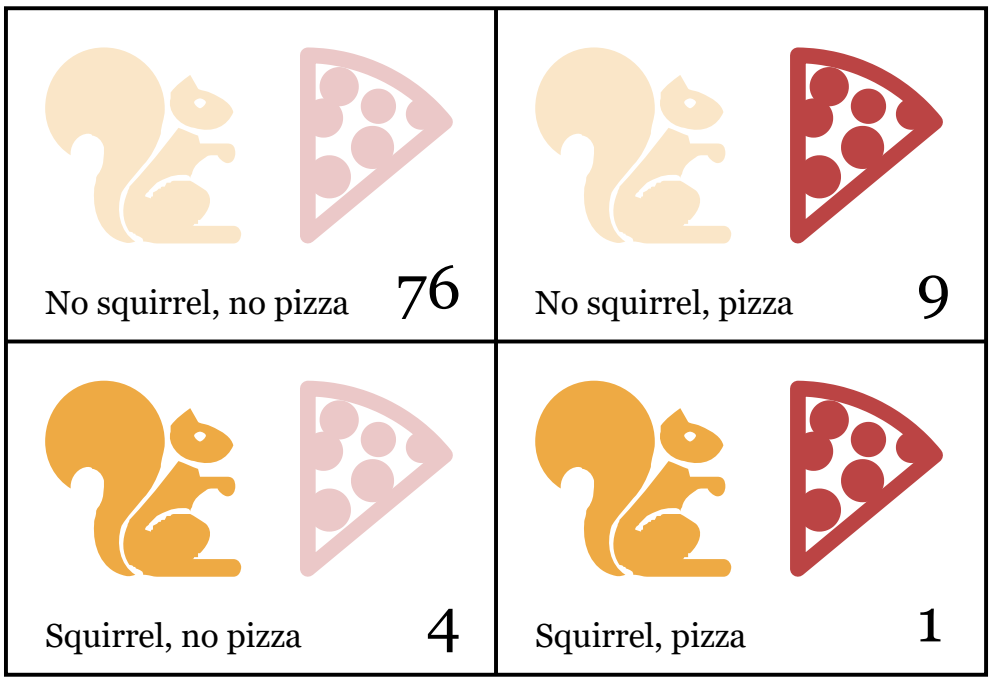
```
addEntry(["work", "touched tree", "pizza", "running",
          "television"], false);
addEntry(["work", "ice cream", "cauliflower", "lasagna",
          "touched tree", "brushed teeth"], false);
addEntry(["weekend", "cycling", "break", "peanuts",
          "beer"], true);
```

Once he has enough data points, he intends to use statistics to find out which of these events may be related to the squirrelifications.

Correlation is a measure of dependence between statistical variables. A statistical variable is not quite the same as a programming variable. In statistics you typically have a set of *measurements*, and each variable is measured for every measurement. Correlation between variables is usually expressed as a value that ranges from -1 to 1. Zero correlation means the variables are not related. A correlation of one indicates that the two are perfectly related—if you know one, you also know the other. Negative one also means that the variables are perfectly related but that they are opposites—when one is true, the other is false.

To compute the measure of correlation between two Boolean variables, we can use the *phi coefficient* (ϕ). This is a formula whose input is a frequency table containing the number of times the different combinations of the variables were observed. The output of the formula is a number between -1 and 1 that describes the correlation.

We could take the event of eating pizza and put that in a frequency table like this, where each number indicates the amount of times that combination occurred in our measurements:



If we call that table n, we can compute ϕ using the following formula:

$$\phi = \frac{n_{11}n_{00} - n_{10}n_{01}}{\sqrt{n_{1\bullet}n_{0\bullet}n_{\bullet 1}n_{\bullet 0}}}$$

(If at this point you're putting the book down to focus on a terrible flashback to 10th grade math class—hold on! I do not intend to torture you with endless pages of cryptic notation—it's just this one formula for now. And even with this one, all we do is turn it into JavaScript.)

The notation n_{01} indicates the number of measurements where the first variable (squirrelness) is false (0) and the second variable (pizza) is true (1). In the pizza table, n_{01} is 9.

The value $n_{1\bullet}$ refers to the sum of all measurements where the first variable is true, which is 5 in the example table. Likewise, $n_{\bullet 0}$ refers to the sum of the measurements where the second variable is false.

So for the pizza table, the part above the division line (the dividend) would be $1\times76-4\times9 = 40$, and the part below it (the divisor) would be the square root of $5\times85\times10\times80$, or $\sqrt{340000}$. This comes out to $\phi \approx 0.069$, which is tiny. Eating pizza does not appear to have influence on the transformations.

Computing correlation

We can represent a two-by-two table in JavaScript with a four-element array (`[76, 9, 4, 1]`). We could also use other representations, such as an array containing two two-element arrays (`[[76, 9], [4, 1]]`) or an object with property names like `"11"` and `"01"`, but the flat array is simple and makes the expressions that access the table pleasantly short. We'll interpret the indices to the array as two-bit binary numbers, where the leftmost (most significant) digit refers to the squirrel variable and the rightmost (least significant) digit refers to the event variable. For example, the binary number `10` refers to the case where Jacques did turn into a squirrel, but the event (say, "pizza") didn't occur. This happened four times. And since binary `10` is `2` in decimal notation, we will store this number at index `2` of the array.

This is the function that computes the ϕ coefficient from such an array:

```
function phi(table) {
  return (table[3] * table[0] - table[2] * table[1]) /
    Math.sqrt((table[2] + table[3]) *
              (table[0] + table[1]) *
              (table[1] + table[3]) *
              (table[0] + table[2]));
}

console.log(phi([76, 9, 4, 1]));
// → 0.068599434
```

This is a direct translation of the ϕ formula into JavaScript. `Math.sqrt` is the square root function, as provided by the `Math` object in a standard JavaScript environment. We have to add two fields from the table to get fields like n_1. because the sums of rows or columns are not stored directly in our data structure.

Jacques kept his journal for three months. The resulting data set is available in the coding sandbox for this chapter, where it is stored in the `JOURNAL` binding and in a downloadable file.

To extract a two-by-two table for a specific event from the journal, we must loop over all the entries and tally how many times the event occurs in relation to squirrel transformations.

```
function tableFor(event, journal) {
  let table = [0, 0, 0, 0];
  for (let i = 0; i < journal.length; i++) {
    let entry = journal[i], index = 0;
    if (entry.events.includes(event)) index += 1;
    if (entry.squirrel) index += 2;
    table[index] += 1;
  }
  return table;
}

console.log(tableFor("pizza", JOURNAL));
// → [76, 9, 4, 1]
```

Arrays have an `includes` method that checks whether a given value exists in the array. The function uses that to determine whether the event name it is interested in is part of the event list for a given day.

The body of the loop in `tableFor` figures out which box in the table each journal entry falls into by checking whether the entry contains the specific event it's interested in and whether the event happens alongside a squirrel incident. The loop then adds one to the correct box in the table.

We now have the tools we need to compute individual correlations. The only step remaining is to find a correlation for every type of event that was recorded and see whether anything stands out.

Array loops

In the `tableFor` function, there's a loop like this:

```
for (let i = 0; i < JOURNAL.length; i++) {
  let entry = JOURNAL[i];
```

```
  // Do something with entry
}
```

This kind of loop is common in classical JavaScript—going over arrays one element at a time is something that comes up a lot, and to do that you'd run a counter over the length of the array and pick out each element in turn.

There is a simpler way to write such loops in modern JavaScript.

```
for (let entry of JOURNAL) {
  console.log(`${entry.events.length} events.`);
}
```

When a `for` loop looks like this, with the word `of` after a variable definition, it will loop over the elements of the value given after `of`. This works not only for arrays but also for strings and some other data structures. We'll discuss *how* it works in Chapter 6.

The final analysis

We need to compute a correlation for every type of event that occurs in the data set. To do that, we first need to *find* every type of event.

```
function journalEvents(journal) {
  let events = [];
  for (let entry of journal) {
    for (let event of entry.events) {
      if (!events.includes(event)) {
        events.push(event);
      }
    }
  }
  return events;
}

console.log(journalEvents(JOURNAL));
// → ["carrot", "exercise", "weekend", "bread", …]
```

By going over all the events and adding those that aren't already in there to the `events` array, the function collects every type of event.

Using that, we can see all the correlations.

```
for (let event of journalEvents(JOURNAL)) {
  console.log(event + ":", phi(tableFor(event, JOURNAL)));
}
// → carrot:   0.0140970969
// → exercise: 0.0685994341
// → weekend:  0.1371988681
// → bread:   -0.0757554019
// → pudding: -0.0648203724
// and so on...
```

Most correlations seem to lie close to zero. Eating carrots, bread, or pudding apparently does not trigger squirrel-lycanthropy. It *does* seem to occur somewhat more often on weekends. Let's filter the results to show only correlations greater than 0.1 or less than -0.1.

```
for (let event of journalEvents(JOURNAL)) {
  let correlation = phi(tableFor(event, JOURNAL));
  if (correlation > 0.1 || correlation < -0.1) {
    console.log(event + ":", correlation);
  }
}
// → weekend:        0.1371988681
// → brushed teeth: -0.3805211953
// → candy:          0.1296407447
// → work:          -0.1371988681
// → spaghetti:      0.2425356250
// → reading:        0.1106828054
// → peanuts:        0.5902679812
```

Aha! There are two factors with a correlation that's clearly stronger than the others. Eating peanuts has a strong positive effect on the chance of turning into a squirrel, whereas brushing his teeth has a significant negative effect.

Interesting. Let's try something.

```
for (let entry of JOURNAL) {
  if (entry.events.includes("peanuts") &&
      !entry.events.includes("brushed teeth")) {
    entry.events.push("peanut teeth");
  }
}
console.log(phi(tableFor("peanut teeth", JOURNAL)));
// → 1
```

That's a strong result. The phenomenon occurs precisely when Jacques eats peanuts and fails to brush his teeth. If only he weren't such a slob about dental hygiene, he'd have never even noticed his affliction.

Knowing this, Jacques stops eating peanuts altogether and finds that his transformations don't come back.

For a few years, things go great for Jacques. But at some point he loses his job. Because he lives in a nasty country where having no job means having no medical services, he is forced to take employment with a circus where he performs as *The Incredible Squirrelman*, stuffing his mouth with peanut butter before every show.

One day, fed up with this pitiful existence, Jacques fails to change back into his human form, hops through a crack in the circus tent, and vanishes into the forest. He is never seen again.

Further arrayology

Before finishing the chapter, I want to introduce you to a few more object-related concepts. I'll start by introducing some generally useful array methods.

We saw `push` and `pop`, which add and remove elements at the end of an array, earlier in this chapter. The corresponding methods for adding and removing things at the start of an array are called `unshift` and `shift`.

```
let todoList = [];
function remember(task) {
  todoList.push(task);
```

```
}
function getTask() {
  return todoList.shift();
}
function rememberUrgently(task) {
  todoList.unshift(task);
}
```

That program manages a queue of tasks. You add tasks to the end of the queue by calling `remember("groceries")`, and when you're ready to do something, you call `getTask()` to get (and remove) the front item from the queue. The `rememberUrgently` function also adds a task but adds it to the front instead of the back of the queue.

To search for a specific value, arrays provide an `indexOf` method. The method searches through the array from the start to the end and returns the index at which the requested value was found—or -1 if it wasn't found. To search from the end instead of the start, there's a similar method called `lastIndexOf`.

```
console.log([1, 2, 3, 2, 1].indexOf(2));
// → 1
console.log([1, 2, 3, 2, 1].lastIndexOf(2));
// → 3
```

Both `indexOf` and `lastIndexOf` take an optional second argument that indicates where to start searching.

Another fundamental array method is `slice`, which takes start and end indices and returns an array that has only the elements between them. The start index is inclusive, the end index exclusive.

```
console.log([0, 1, 2, 3, 4].slice(2, 4));
// → [2, 3]
console.log([0, 1, 2, 3, 4].slice(2));
// → [2, 3, 4]
```

When the end index is not given, `slice` will take all of the elements after the start index. You can also omit the start index to copy the entire array.

The `concat` method can be used to glue arrays together to create a new array, similar to what the `+` operator does for strings.

The following example shows both `concat` and `slice` in action. It takes an array and an index, and it returns a new array that is a copy of the original array with the element at the given index removed.

```
function remove(array, index) {
  return array.slice(0, index)
    .concat(array.slice(index + 1));
}
console.log(remove(["a", "b", "c", "d", "e"], 2));
// → ["a", "b", "d", "e"]
```

If you pass `concat` an argument that is not an array, that value will be added to the new array as if it were a one-element array.

Strings and their properties

We can read properties like `length` and `toUpperCase` from string values. But if you try to add a new property, it doesn't stick.

```
let kim = "Kim";
kim.age = 88;
console.log(kim.age);
// → undefined
```

Values of type string, number, and Boolean are not objects, and though the language doesn't complain if you try to set new properties on them, it doesn't actually store those properties. As mentioned earlier, such values are immutable and cannot be changed.

But these types do have built-in properties. Every string value has a number of methods. Some very useful ones are `slice` and `indexOf`, which resemble the array methods of the same name.

```
console.log("coconuts".slice(4, 7));
// → nut
```

```
console.log("coconut".indexOf("u"));
// → 5
```

One difference is that a string's `indexOf` can search for a string containing more than one character, whereas the corresponding array method looks only for a single element.

```
console.log("one two three".indexOf("ee"));
// → 11
```

The `trim` method removes whitespace (spaces, newlines, tabs, and similar characters) from the start and end of a string.

```
console.log("  okay \n ".trim());
// → okay
```

The `zeroPad` function from the previous chapter also exists as a method. It is called `padStart` and takes the desired length and padding character as arguments.

```
console.log(String(6).padStart(3, "0"));
// → 006
```

You can split a string on every occurrence of another string with `split` and join it again with `join`.

```
let sentence = "Secretarybirds specialize in stomping";
let words = sentence.split(" ");
console.log(words);
// → ["Secretarybirds", "specialize", "in", "stomping"]
console.log(words.join(". "));
// → Secretarybirds. specialize. in. stomping
```

A string can be repeated with the `repeat` method, which creates a new string containing multiple copies of the original string, glued together.

```
console.log("LA".repeat(3));
// → LALALA
```

We have already seen the string type's `length` property. Accessing the individual characters in a string looks like accessing array elements (with a caveat that we'll discuss in Chapter 5).

```
let string = "abc";
console.log(string.length);
// → 3
console.log(string[1]);
// → b
```

Rest parameters

It can be useful for a function to accept any number of arguments. For example, `Math.max` computes the maximum of *all* the arguments it is given.

To write such a function, you put three dots before the function's last parameter, like this:

```
function max(...numbers) {
  let result = -Infinity;
  for (let number of numbers) {
    if (number > result) result = number;
  }
  return result;
}
console.log(max(4, 1, 9, -2));
// → 9
```

When such a function is called, the *rest parameter* is bound to an array containing all further arguments. If there are other parameters before it, their values aren't part of that array. When, as in `max`, it is the only parameter, it will hold all arguments.

You can use a similar three-dot notation to *call* a function with an array of arguments.

```
let numbers = [5, 1, 7];
console.log(max(...numbers));
// → 7
```

This "spreads" out the array into the function call, passing its elements as separate arguments. It is possible to include an array like that along with other arguments, as in `max(9, ...numbers, 2)`.

Square bracket array notation similarly allows the triple-dot operator to spread another array into the new array.

```
let words = ["never", "fully"];
console.log(["will", ...words, "understand"]);
// → ["will", "never", "fully", "understand"]
```

The Math object

As we've seen, `Math` is a grab bag of number-related utility functions, such as `Math.max` (maximum), `Math.min` (minimum), and `Math.sqrt` (square root).

The `Math` object is used as a container to group a bunch of related functionality. There is only one `Math` object, and it is almost never useful as a value. Rather, it provides a *namespace* so that all these functions and values do not have to be global bindings.

Having too many global bindings "pollutes" the namespace. The more names have been taken, the more likely you are to accidentally overwrite the value of some existing binding. For example, it's not unlikely to want to name something `max` in one of your programs. Since JavaScript's built-in `max` function is tucked safely inside the `Math` object, we don't have to worry about overwriting it.

Many languages will stop you, or at least warn you, when you are defining a binding with a name that is already taken. JavaScript does this for bindings you declared with `let` or `const` but—perversely—not for standard bindings nor for bindings declared with `var` or `function`.

Back to the `Math` object. If you need to do trigonometry, `Math` can help. It contains `cos` (cosine), `sin` (sine), and `tan` (tangent), as well as their inverse functions, `acos`, `asin`, and `atan`, respectively. The number π (pi)—or at least

the closest approximation that fits in a JavaScript number—is available as `Math.PI`. There is an old programming tradition of writing the names of constant values in all caps.

```
function randomPointOnCircle(radius) {
  let angle = Math.random() * 2 * Math.PI;
  return {x: radius * Math.cos(angle),
          y: radius * Math.sin(angle)};
}
console.log(randomPointOnCircle(2));
// → {x: 0.3667, y: 1.966}
```

If sines and cosines are not something you are familiar with, don't worry. When they are used in this book, in Chapter 14, I'll explain them.

The previous example used `Math.random`. This is a function that returns a new pseudorandom number between zero (inclusive) and one (exclusive) every time you call it.

```
console.log(Math.random());
// → 0.36993729369714856
console.log(Math.random());
// → 0.727367032552138
console.log(Math.random());
// → 0.40180766698904335
```

Though computers are deterministic machines—they always react the same way if given the same input—it is possible to have them produce numbers that appear random. To do that, the machine keeps some hidden value, and whenever you ask for a new random number, it performs complicated computations on this hidden value to create a new value. It stores a new value and returns some number derived from it. That way, it can produce ever new, hard-to-predict numbers in a way that *seems* random.

If we want a whole random number instead of a fractional one, we can use `Math.floor` (which rounds down to the nearest whole number) on the result of `Math.random`.

```
console.log(Math.floor(Math.random() * 10));
// → 2
```

Multiplying the random number by 10 gives us a number greater than or equal to 0 and below 10. Since `Math.floor` rounds down, this expression will produce, with equal chance, any number from 0 through 9.

There are also the functions `Math.ceil` (for "ceiling", which rounds up to a whole number), `Math.round` (to the nearest whole number), and `Math.abs`, which takes the absolute value of a number, meaning it negates negative values but leaves positive ones as they are.

Destructuring

Let's go back to the `phi` function for a moment.

```
function phi(table) {
  return (table[3] * table[0] - table[2] * table[1]) /
    Math.sqrt((table[2] + table[3]) *
              (table[0] + table[1]) *
              (table[1] + table[3]) *
              (table[0] + table[2]));
}
```

One of the reasons this function is awkward to read is that we have a binding pointing at our array, but we'd much prefer to have bindings for the *elements* of the array, that is, `let n00 = table[0]` and so on. Fortunately, there is a succinct way to do this in JavaScript.

```
function phi([n00, n01, n10, n11]) {
  return (n11 * n00 - n10 * n01) /
    Math.sqrt((n10 + n11) * (n00 + n01) *
              (n01 + n11) * (n00 + n10));
}
```

This also works for bindings created with `let`, `var`, or `const`. If you know the value you are binding is an array, you can use square brackets to "look inside" of the value, binding its contents.

A similar trick works for objects, using braces instead of square brackets.

```
let {name} = {name: "Faraji", age: 23};
console.log(name);
// → Faraji
```

Note that if you try to destructure `null` or `undefined`, you get an error, much as you would if you directly try to access a property of those values.

JSON

Because properties only grasp their value, rather than contain it, objects and arrays are stored in the computer's memory as sequences of bits holding the *addresses*—the place in memory—of their contents. So an array with another array inside of it consists of (at least) one memory region for the inner array, and another for the outer array, containing (among other things) a binary number that represents the position of the inner array.

If you want to save data in a file for later or send it to another computer over the network, you have to somehow convert these tangles of memory addresses to a description that can be stored or sent. You *could* send over your entire computer memory along with the address of the value you're interested in, I suppose, but that doesn't seem like the best approach.

What we can do is *serialize* the data. That means it is converted into a flat description. A popular serialization format is called *JSON* (pronounced "Jason"), which stands for JavaScript Object Notation. It is widely used as a data storage and communication format on the Web, even in languages other than JavaScript.

JSON looks similar to JavaScript's way of writing arrays and objects, with a few restrictions. All property names have to be surrounded by double quotes, and only simple data expressions are allowed—no function calls, bindings, or anything that involves actual computation. Comments are not allowed in JSON.

A journal entry might look like this when represented as JSON data:

```
{
  "squirrel": false,
  "events": ["work", "touched tree", "pizza", "running"]
}
```

JavaScript gives us the functions `JSON.stringify` and `JSON.parse` to convert data to and from this format. The first takes a JavaScript value and returns a JSON-encoded string. The second takes such a string and converts it to the value it encodes.

```
let string = JSON.stringify({squirrel: false,
                             events: ["weekend"]});
console.log(string);
// → {"squirrel":false,"events":["weekend"]}
console.log(JSON.parse(string).events);
// → ["weekend"]
```

Summary

Objects and arrays (which are a specific kind of object) provide ways to group several values into a single value. Conceptually, this allows us to put a bunch of related things in a bag and run around with the bag, instead of wrapping our arms around all of the individual things and trying to hold on to them separately.

Most values in JavaScript have properties, the exceptions being `null` and `undefined`. Properties are accessed using `value.prop` or `value["prop"]`. Objects tend to use names for their properties and store more or less a fixed set of them. Arrays, on the other hand, usually contain varying amounts of conceptually identical values and use numbers (starting from 0) as the names of their properties.

There *are* some named properties in arrays, such as `length` and a number of methods. Methods are functions that live in properties and (usually) act on the value they are a property of.

You can iterate over arrays using a special kind of `for` loop—`for (let element of array)`.

Exercises

The sum of a range

The introduction of this book alluded to the following as a nice way to compute the sum of a range of numbers:

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

Write a `range` function that takes two arguments, `start` and `end`, and returns an array containing all the numbers from `start` up to (and including) `end`.

Next, write a `sum` function that takes an array of numbers and returns the sum of these numbers. Run the example program and see whether it does indeed return 55.

As a bonus assignment, modify your `range` function to take an optional third argument that indicates the "step" value used when building the array. If no step is given, the elements go up by increments of one, corresponding to the old behavior. The function call `range(1, 10, 2)` should return `[1, 3, 5, 7, 9]`. Make sure it also works with negative step values so that `range(5, 2, -1)` produces `[5, 4, 3, 2]`.

```
// Your code here.

console.log(range(1, 10));
// → [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
console.log(range(5, 2, -1));
// → [5, 4, 3, 2]
console.log(sum(range(1, 10)));
// → 55
```

» Display hints...

Reversing an array

Arrays have a `reverse` method that changes the array by inverting the order in which its elements appear. For this exercise, write two functions, `reverseArray` and `reverseArrayInPlace`. The first, `reverseArray`, takes an array as argument and produces a *new* array that has the same elements in the inverse order. The second, `reverseArrayInPlace`, does what the `reverse` method does: it *modifies* the array given as argument by reversing its elements. Neither may use the standard `reverse` method.

Thinking back to the notes about side effects and pure functions in the previous chapter, which variant do you expect to be useful in more situations? Which one runs faster?

```
// Your code here.

console.log(reverseArray(["A", "B", "C"]));
// → ["C", "B", "A"];
let arrayValue = [1, 2, 3, 4, 5];
reverseArrayInPlace(arrayValue);
console.log(arrayValue);
// → [5, 4, 3, 2, 1]
```

» Display hints...

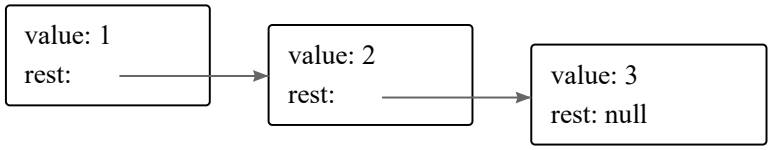
A LIST

Objects, as generic blobs of values, can be used to build all sorts of data structures. A common data structure is the *list* (not to be confused with array). A list is a nested set of objects, with the first object holding a reference to the second, the second to the third, and so on.

```
let list = {
  value: 1,
  rest: {
    value: 2,
    rest: {
      value: 3,
      rest: null
    }
```

```
  }
};
```

The resulting objects form a chain, like this:



A nice thing about lists is that they can share parts of their structure. For example, if I create two new values `{value: 0, rest: list}` and `{value: -1, rest: list}` (with `list` referring to the binding defined earlier), they are both independent lists, but they share the structure that makes up their last three elements. The original list is also still a valid three-element list.

Write a function `arrayToList` that builds up a list structure like the one shown when given `[1, 2, 3]` as argument. Also write a `listToArray` function that produces an array from a list. Then add a helper function `prepend`, which takes an element and a list and creates a new list that adds the element to the front of the input list, and `nth`, which takes a list and a number and returns the element at the given position in the list (with zero referring to the first element) or `undefined` when there is no such element.

If you haven't already, also write a recursive version of `nth`.

```
// Your code here.

console.log(arrayToList([10, 20]));
// → {value: 10, rest: {value: 20, rest: null}}
console.log(listToArray(arrayToList([10, 20, 30])));
// → [10, 20, 30]
console.log(prepend(10, prepend(20, null)));
// → {value: 10, rest: {value: 20, rest: null}}
console.log(nth(arrayToList([10, 20, 30]), 1));
// → 20
```

» Display hints...

Deep comparison

The `==` operator compares objects by identity. But sometimes you'd prefer to compare the values of their actual properties.

Write a function `deepEqual` that takes two values and returns true only if they are the same value or are objects with the same properties, where the values of the properties are equal when compared with a recursive call to `deepEqual`.

To find out whether values should be compared directly (use the `===` operator for that) or have their properties compared, you can use the `typeof` operator. If it produces `"object"` for both values, you should do a deep comparison. But you have to take one silly exception into account: because of a historical accident, `typeof null` also produces `"object"`.

The `Object.keys` function will be useful when you need to go over the properties of objects to compare them.

```
// Your code here.

let obj = {here: {is: "an"}, object: 2};
console.log(deepEqual(obj, obj));
// → true
console.log(deepEqual(obj, {here: 1, object: 2}));
// → false
console.log(deepEqual(obj, {here: {is: "an"}, object: 2}));
// → true
```

» Display hints...

CHAPTER 5

HIGHER-ORDER FUNCTIONS

> "Tzu-li and Tzu-ssu were boasting about the size of their latest programs. 'Two-hundred thousand lines,' said Tzu-li, 'not counting comments!' Tzu-ssu responded, 'Pssh, mine is almost a *million* lines already.' Master Yuan-Ma said, 'My best program has five hundred lines.' Hearing this, Tzu-li and Tzu-ssu were enlightened."
>
> — Master Yuan-Ma, *The Book of Programming*

> "There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies." ¶
>
> — C.A.R. Hoare, *1980 ACM Turing Award Lecture*

A large program is a costly program, and not just because of the time it takes to build. Size almost always involves complexity, and complexity confuses programmers. Confused programmers, in turn, introduce mistakes (*bugs*) into programs. A large program then provides a lot of space for these bugs to hide, making them hard to find.

Let's briefly go back to the final two example programs in the introduction. The first is self-contained and six lines long.

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


The second relies on two external functions and is one line long.

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

Which one is more likely to contain a bug?

If we count the size of the definitions of `sum` and `range`, the second program is also big—even bigger than the first. But still, I'd argue that it is more likely to be correct.

It is more likely to be correct because the solution is expressed in a vocabulary that corresponds to the problem being solved. Summing a range of numbers isn't about loops and counters. It is about ranges and sums.

The definitions of this vocabulary (the functions `sum` and `range`) will still involve loops, counters, and other incidental details. But because they are expressing simpler concepts than the program as a whole, they are easier to get right.

Abstraction

In the context of programming, these kinds of vocabularies are usually called *abstractions*. Abstractions hide details and give us the ability to talk about problems at a higher (or more abstract) level.

As an analogy, compare these two recipes for pea soup. The first one goes like this:

> "Put 1 cup of dried peas per person into a container. Add water until the peas are well covered. Leave the peas in water for at least 12 hours. Take the peas out of the water and put them in a cooking pan. Add 4 cups of water per person. Cover the pan and

> keep the peas simmering for two hours. Take half an onion per person. Cut it into pieces with a knife. Add it to the peas. Take a stalk of celery per person. Cut it into pieces with a knife. Add it to the peas. Take a carrot per person. Cut it into pieces. With a knife! Add it to the peas. Cook for 10 more minutes."

And this is the second recipe:

> "Per person: 1 cup dried split peas, half a chopped onion, a stalk of celery, and a carrot.
>
> Soak peas for 12 hours. Simmer for 2 hours in 4 cups of water (per person). Chop and add vegetables. Cook for 10 more minutes."

The second is shorter and easier to interpret. But you do need to understand a few more cooking-related words such as *soak*, *simmer*, *chop*, and, I guess, *vegetable*.

When programming, we can't rely on all the words we need to be waiting for us in the dictionary. Thus, we might fall into the pattern of the first recipe—work out the precise steps the computer has to perform, one by one, blind to the higher-level concepts that they express.

It is a useful skill, in programming, to notice when you are working at too low a level of abstraction.

Abstracting Repetition

Plain functions, as we've seen them so far, are a good way to build abstractions. But sometimes they fall short.

It is common for a program to do something a given number of times. You can write a `for` loop for that, like this:

```
for (let i = 0; i < 10; i++) {
  console.log(i);
}
```

Can we abstract "doing something *N* times" as a function? Well, it's easy to write a function that calls `console.log` *N* times.

```
function repeatLog(n) {
  for (let i = 0; i < n; i++) {
    console.log(i);
  }
}
```

But what if we want to do something other than logging the numbers? Since "doing something" can be represented as a function and functions are just values, we can pass our action as a function value.

```
function repeat(n, action) {
  for (let i = 0; i < n; i++) {
    action(i);
  }
}

repeat(3, console.log);
// → 0
// → 1
// → 2
```

We don't have to pass a predefined function to `repeat`. Often, it is easier to create a function value on the spot instead.

```
let labels = [];
repeat(5, i => {
  labels.push(`Unit ${i + 1}`);
});
console.log(labels);
// → ["Unit 1", "Unit 2", "Unit 3", "Unit 4", "Unit 5"]
```

This is structured a little like a `for` loop—it first describes the kind of loop and then provides a body. However, the body is now written as a function value, which is wrapped in the parentheses of the call to `repeat`. This is why it has to be closed with the closing brace *and* closing parenthesis. In cases like this example, where the body is a single small expression, you could also omit the braces and write the loop on a single line.

Higher-order functions

Functions that operate on other functions, either by taking them as arguments or by returning them, are called *higher-order functions*. Since we have already seen that functions are regular values, there is nothing particularly remarkable about the fact that such functions exist. The term comes from mathematics, where the distinction between functions and other values is taken more seriously.

Higher-order functions allow us to abstract over *actions*, not just values. They come in several forms. For example, we can have functions that create new functions.

```
function greaterThan(n) {
  return m => m > n;
}
let greaterThan10 = greaterThan(10);
console.log(greaterThan10(11));
// → true
```

And we can have functions that change other functions.

```
function noisy(f) {
  return (...args) => {
    console.log("calling with", args);
    let result = f(...args);
    console.log("called with", args, ", returned", result);
    return result;
  };
}
noisy(Math.min)(3, 2, 1);
// → calling with [3, 2, 1]
// → called with [3, 2, 1] , returned 1
```

We can even write functions that provide new types of control flow.

```
function unless(test, then) {
  if (!test) then();
}
```

```
repeat(3, n => {
  unless(n % 2 == 1, () => {
    console.log(n, "is even");
  });
});
// → 0 is even
// → 2 is even
```

There is a built-in array method, `forEach`, that provides something like a `for/of` loop as a higher-order function.

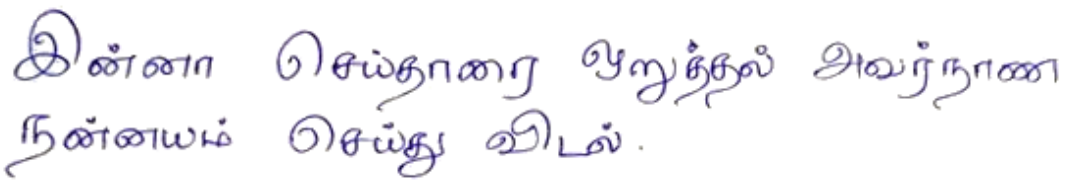
```
["A", "B"].forEach(l => console.log(l));
// → A
// → B
```

Script data set

One area where higher-order functions shine is data processing. To process data, we'll need some actual data. This chapter will use a data set about scripts—writing systems such as Latin, Cyrillic, or Arabic.

Remember Unicode from Chapter 1, the system that assigns a number to each character in written language? Most of these characters are associated with a specific script. The standard contains 140 different scripts—81 are still in use today, and 59 are historic.

Though I can fluently read only Latin characters, I appreciate the fact that people are writing texts in at least 80 other writing systems, many of which I wouldn't even recognize. For example, here's a sample of Tamil handwriting:



The example data set contains some pieces of information about the 140 scripts defined in Unicode. It is available in the coding sandbox for this chapter as the

`SCRIPTS` binding. The binding contains an array of objects, each of which describes a script.

```
{
  name: "Coptic",
  ranges: [[994, 1008], [11392, 11508], [11513, 11520]],
  direction: "ltr",
  year: -200,
  living: false,
  link: "https://en.wikipedia.org/wiki/Coptic_alphabet"
}
```

Such an object tells us the name of the script, the Unicode ranges assigned to it, the direction in which it is written, the (approximate) origin time, whether it is still in use, and a link to more information. The direction may be `"ltr"` for left to right, `"rtl"` for right to left (the way Arabic and Hebrew text are written), or `"ttb"` for top to bottom (as with Mongolian writing).

The `ranges` property contains an array of Unicode character ranges, each of which is a two-element array containing a lower bound and an upper bound. Any character codes within these ranges are assigned to the script. The lower bound is inclusive (code 994 is a Coptic character), and the upper bound is non-inclusive (code 1008 isn't).

Filtering arrays

To find the scripts in the data set that are still in use, the following function might be helpful. It filters out the elements in an array that don't pass a test.

```
function filter(array, test) {
  let passed = [];
  for (let element of array) {
    if (test(element)) {
      passed.push(element);
    }
  }
  return passed;
}
```

```
console.log(filter(SCRIPTS, script => script.living));
// → [{name: "Adlam", …}, …]
```

The function uses the argument named `test`, a function value, to fill a "gap" in the computation—the process of deciding which elements to collect.

Note how the `filter` function, rather than deleting elements from the existing array, builds up a new array with only the elements that pass the test. This function is *pure*. It does not modify the array it is given.

Like `forEach`, `filter` is a standard array method. The example defined the function only to show what it does internally. From now on, we'll use it like this instead:

```
console.log(SCRIPTS.filter(s => s.direction == "ttb"));
// → [{name: "Mongolian", …}, …]
```

Transforming with map

Say we have an array of objects representing scripts, produced by filtering the `SCRIPTS` array somehow. But we want an array of names, which is easier to inspect.

The `map` method transforms an array by applying a function to all of its elements and building a new array from the returned values. The new array will have the same length as the input array, but its content will have been *mapped* to a new form by the function.

```
function map(array, transform) {
  let mapped = [];
  for (let element of array) {
    mapped.push(transform(element));
  }
  return mapped;
}

let rtlScripts = SCRIPTS.filter(s => s.direction == "rtl");
```

```
console.log(map(rtlScripts, s => s.name));
// → ["Adlam", "Arabic", "Imperial Aramaic", …]
```

Like `forEach` and `filter`, `map` is a standard array method.

Summarizing with reduce

Another common thing to do with arrays is to compute a single value from them. Our recurring example, summing a collection of numbers, is an instance of this. Another example is finding the script with the most characters.

The higher-order operation that represents this pattern is called *reduce* (sometimes also called *fold*). It builds a value by repeatedly taking a single element from the array and combining it with the current value. When summing numbers, you'd start with the number zero and, for each element, add that to the sum.

The parameters to `reduce` are, apart from the array, a combining function and a start value. This function is a little less straightforward than `filter` and `map`, so take a close look at it:

```
function reduce(array, combine, start) {
  let current = start;
  for (let element of array) {
    current = combine(current, element);
  }
  return current;
}

console.log(reduce([1, 2, 3, 4], (a, b) => a + b, 0));
// → 10
```

The standard array method `reduce`, which of course corresponds to this function, has an added convenience. If your array contains at least one element, you are allowed to leave off the `start` argument. The method will take the first element of the array as its start value and start reducing at the second element.

```
console.log([1, 2, 3, 4].reduce((a, b) => a + b));
// → 10
```

To use `reduce` (twice) to find the script with the most characters, we can write something like this:

```
function characterCount(script) {
  return script.ranges.reduce((count, [from, to]) => {
    return count + (to - from);
  }, 0);
}

console.log(SCRIPTS.reduce((a, b) => {
  return characterCount(a) < characterCount(b) ? b : a;
}));
// → {name: "Han", …}
```

The `characterCount` function reduces the ranges assigned to a script by summing their sizes. Note the use of destructuring in the parameter list of the reducer function. The second call to `reduce` then uses this to find the largest script by repeatedly comparing two scripts and returning the larger one.

The Han script has more than 89,000 characters assigned to it in the Unicode standard, making it by far the biggest writing system in the data set. Han is a script (sometimes) used for Chinese, Japanese, and Korean text. Those languages share a lot of characters, though they tend to write them differently. The (U.S.-based) Unicode Consortium decided to treat them as a single writing system to save character codes. This is called *Han unification* and still makes some people very angry.

Composability

Consider how we would have written the previous example (finding the biggest script) without higher-order functions. The code is not that much worse.

```
let biggest = null;
for (let script of SCRIPTS) {
  if (biggest == null ||
```

```
      characterCount(biggest) < characterCount(script)) {
    biggest = script;
  }
}
console.log(biggest);
// → {name: "Han", …}
```

There are a few more bindings, and the program is four lines longer. But it is still very readable.

Higher-order functions start to shine when you need to *compose* operations. As an example, let's write code that finds the average year of origin for living and dead scripts in the data set.

```
function average(array) {
  return array.reduce((a, b) => a + b) / array.length;
}

console.log(Math.round(average(
  SCRIPTS.filter(s => s.living).map(s => s.year))));
// → 1165
console.log(Math.round(average(
  SCRIPTS.filter(s => !s.living).map(s => s.year))));
// → 204
```

So the dead scripts in Unicode are, on average, older than the living ones. This is not a terribly meaningful or surprising statistic. But I hope you'll agree that the code used to compute it isn't hard to read. You can see it as a pipeline: we start with all scripts, filter out the living (or dead) ones, take the years from those, average them, and round the result.

You could definitely also write this computation as one big loop.

```
let total = 0, count = 0;
for (let script of SCRIPTS) {
  if (script.living) {
    total += script.year;
    count += 1;
  }
```

```
}
console.log(Math.round(total / count));
// → 1165
```

But it is harder to see what was being computed and how. And because intermediate results aren't represented as coherent values, it'd be a lot more work to extract something like `average` into a separate function.

In terms of what the computer is actually doing, these two approaches are also quite different. The first will build up new arrays when running `filter` and `map`, whereas the second computes only some numbers, doing less work. You can usually afford the readable approach, but if you're processing huge arrays, and doing so many times, the less abstract style might be worth the extra speed.

Strings and character codes

One use of the data set would be figuring out what script a piece of text is using. Let's go through a program that does this.

Remember that each script has an array of character code ranges associated with it. So given a character code, we could use a function like this to find the corresponding script (if any):

```
function characterScript(code) {
  for (let script of SCRIPTS) {
    if (script.ranges.some(([from, to]) => {
      return code >= from && code < to;
    })) {
      return script;
    }
  }
  return null;
}

console.log(characterScript(121));
// → {name: "Latin", …}
```

The `some` method is another higher-order function. It takes a test function and tells you whether that function returns true for any of the elements in the array.

But how do we get the character codes in a string?

In Chapter 1 I mentioned that JavaScript strings are encoded as a sequence of 16-bit numbers. These are called *code units*. A Unicode character code was initially supposed to fit within such a unit (which gives you a little over 65,000 characters). When it became clear that wasn't going to be enough, many people balked at the need to use more memory per character. To address these concerns, UTF-16, the format used by JavaScript strings, was invented. It describes most common characters using a single 16-bit code unit but uses a pair of two such units for others.

UTF-16 is generally considered a bad idea today. It seems almost intentionally designed to invite mistakes. It's easy to write programs that pretend code units and characters are the same thing. And if your language doesn't use two-unit characters, that will appear to work just fine. But as soon as someone tries to use such a program with some less common Chinese characters, it breaks. Fortunately, with the advent of emoji, everybody has started using two-unit characters, and the burden of dealing with such problems is more fairly distributed.

Unfortunately, obvious operations on JavaScript strings, such as getting their length through the `length` property and accessing their content using square brackets, deal only with code units.

```
// Two emoji characters, horse and shoe
let horseShoe = "🏇👟";
console.log(horseShoe.length);
// → 4
console.log(horseShoe[0]);
// → (Invalid half-character)
console.log(horseShoe.charCodeAt(0));
// → 55357 (Code of the half-character)
console.log(horseShoe.codePointAt(0));
// → 128052 (Actual code for horse emoji)
```

JavaScript's `charCodeAt` method gives you a code unit, not a full character code. The `codePointAt` method, added later, does give a full Unicode character. So we could use that to get characters from a string. But the argument passed to `codePointAt` is still an index into the sequence of code units. So to run over all characters in a string, we'd still need to deal with the question of whether a character takes up one or two code units.

In the previous chapter, I mentioned that a `for/of` loop can also be used on strings. Like `codePointAt`, this type of loop was introduced at a time where people were acutely aware of the problems with UTF-16. When you use it to loop over a string, it gives you real characters, not code units.

```
let roseDragon = "🌹🐉";
for (let char of roseDragon) {
  console.log(char);
}
// → 🌹
// → 🐉
```

If you have a character (which will be a string of one or two code units), you can use `codePointAt(0)` to get its code.

Recognizing text

We have a `characterScript` function and a way to correctly loop over characters. The next step is to count the characters that belong to each script. The following counting abstraction will be useful there:

```
function countBy(items, groupName) {
  let counts = [];
  for (let item of items) {
    let name = groupName(item);
    let known = counts.findIndex(c => c.name == name);
    if (known == -1) {
      counts.push({name, count: 1});
    } else {
      counts[known].count++;
    }
```

```
  }
  return counts;
}

console.log(countBy([1, 2, 3, 4, 5], n => n > 2));
// → [{name: false, count: 2}, {name: true, count: 3}]
```

The `countBy` function expects a collection (anything that we can loop over with `for/of`) and a function that computes a group name for a given element. It returns an array of objects, each of which names a group and tells you the number of elements that were found in that group.

It uses another array method—`findIndex`. This method is somewhat like `indexOf`, but instead of looking for a specific value, it finds the first value for which the given function returns true. Like `indexOf`, it returns `-1` when no such element is found.

Using `countBy`, we can write the function that tells us which scripts are used in a piece of text.

```
function textScripts(text) {
  let scripts = countBy(text, char => {
    let script = characterScript(char.codePointAt(0));
    return script ? script.name : "none";
  }).filter(({name}) => name != "none");

  let total = scripts.reduce((n, {count}) => n + count, 0);
  if (total == 0) return "No scripts found";

  return scripts.map(({name, count}) => {
    return `${Math.round(count * 100 / total)}% ${name}`;
  }).join(", ");
}

console.log(textScripts('英国的狗说"woof", 俄罗斯的狗说"тяв"'));
// → 61% Han, 22% Latin, 17% Cyrillic
```

The function first counts the characters by name, using `characterScript` to assign them a name and falling back to the string `"none"` for characters that

aren't part of any script. The `filter` call drops the entry for `"none"` from the resulting array since we aren't interested in those characters.

To be able to compute percentages, we first need the total number of characters that belong to a script, which we can compute with `reduce`. If no such characters are found, the function returns a specific string. Otherwise, it transforms the counting entries into readable strings with `map` and then combines them with `join`.

Summary

Being able to pass function values to other functions is a deeply useful aspect of JavaScript. It allows us to write functions that model computations with "gaps" in them. The code that calls these functions can fill in the gaps by providing function values.

Arrays provide a number of useful higher-order methods. You can use `forEach` to loop over the elements in an array. The `filter` method returns a new array containing only the elements that pass the predicate function. Transforming an array by putting each element through a function is done with `map`. You can use `reduce` to combine all the elements in an array into a single value. The `some` method tests whether any element matches a given predicate function. And `findIndex` finds the position of the first element that matches a predicate.

Exercises

Flattening

Use the `reduce` method in combination with the `concat` method to "flatten" an array of arrays into a single array that has all the elements of the original arrays.

```
let arrays = [[1, 2, 3], [4, 5], [6]];
// Your code here.
// → [1, 2, 3, 4, 5, 6]
```

Your own loop

Write a higher-order function `loop` that provides something like a `for` loop statement. It takes a value, a test function, an update function, and a body function. Each iteration, it first runs the test function on the current loop value and stops if that returns false. Then it calls the body function, giving it the current value. Finally, it calls the update function to create a new value and starts from the beginning.

When defining the function, you can use a regular loop to do the actual looping.

```
// Your code here.

loop(3, n => n > 0, n => n - 1, console.log);
// → 3
// → 2
// → 1
```

Everything

Analogous to the `some` method, arrays also have an `every` method. This one returns true when the given function returns true for *every* element in the array. In a way, `some` is a version of the `||` operator that acts on arrays, and `every` is like the `&&` operator.

Implement `every` as a function that takes an array and a predicate function as parameters. Write two versions, one using a loop and one using the `some` method.

```
function every(array, test) {
  // Your code here.
}

console.log(every([1, 3, 5], n => n < 10));
// → true
console.log(every([2, 4, 16], n => n < 10));
// → false
console.log(every([], n => n < 10));
// → true
```

» Display hints...

Dominant writing direction

Write a function that computes the dominant writing direction in a string of text. Remember that each script object has a `direction` property that can be `"ltr"` (left to right), `"rtl"` (right to left), or `"ttb"` (top to bottom).

The dominant direction is the direction of a majority of the characters that have a script associated with them. The `characterScript` and `countBy` functions defined earlier in the chapter are probably useful here.

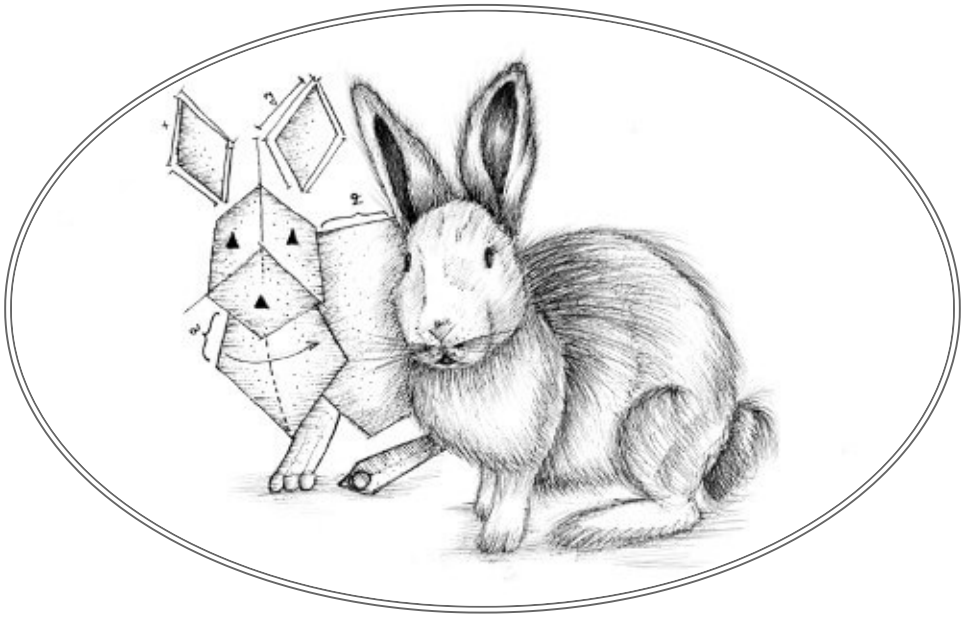
```
function dominantDirection(text) {
  // Your code here.
}

console.log(dominantDirection("Hello!"));
// → ltr
console.log(dominantDirection("Hey, مساء الخير"));
// → rtl
```

» Display hints...

CHAPTER 6

THE SECRET LIFE OF OBJECTS

> "An abstract data type is realized by writing a special kind of program [...] which defines the type in terms of the operations which can be performed on it."
>
> — Barbara Liskov, *Programming with Abstract Data Types*



Chapter 4 introduced JavaScript's objects. In programming culture, we have a thing called *object-oriented programming*, a set of techniques that use objects (and related concepts) as the central principle of program organization.

Though no one really agrees on its precise definition, object-oriented programming has shaped the design of many programming languages, including JavaScript. This chapter will describe the way these ideas can be applied in JavaScript.

ENCAPSULATION

The core idea in object-oriented programming is to divide programs into smaller pieces and make each piece responsible for managing its own state.

This way, some knowledge about the way a piece of the program works can be kept *local* to that piece. Someone working on the rest of the program does not have to remember or even be aware of that knowledge. Whenever these local details change, only the code directly around it needs to be updated.

Different pieces of such a program interact with each other through *interfaces*, limited sets of functions or bindings that provide useful functionality at a more abstract level, hiding their precise implementation.

Such program pieces are modeled using objects. Their interface consists of a specific set of methods and properties. Properties that are part of the interface are called *public*. The others, which outside code should not be touching, are called *private*.

Many languages provide a way to distinguish public and private properties and prevent outside code from accessing the private ones altogether. JavaScript, once again taking the minimalist approach, does not—not yet at least. There is work underway to add this to the language.

Even though the language doesn't have this distinction built in, JavaScript programmers *are* successfully using this idea. Typically, the available interface is described in documentation or comments. It is also common to put an underscore (`_`) character at the start of property names to indicate that those properties are private.

Separating interface from implementation is a great idea. It is usually called *encapsulation*.

Methods

Methods are nothing more than properties that hold function values. This is a simple method:

```
let rabbit = {};
rabbit.speak = function(line) {
  console.log(`The rabbit says '${line}'`);
};
```

edit & run code by clicking it

```
rabbit.speak("I'm alive.");
// → The rabbit says 'I'm alive.'
```

Usually a method needs to do something with the object it was called on. When a function is called as a method—looked up as a property and immediately called, as in `object.method()`—the binding called `this` in its body automatically points at the object that it was called on.

```
function speak(line) {
  console.log(`The ${this.type} rabbit says '${line}'`);
}
let whiteRabbit = {type: "white", speak};
let hungryRabbit = {type: "hungry", speak};

whiteRabbit.speak("Oh my ears and whiskers, " +
                  "how late it's getting!");
// → The white rabbit says 'Oh my ears and whiskers, how
//   late it's getting!'
hungryRabbit.speak("I could use a carrot right now.");
// → The hungry rabbit says 'I could use a carrot right now.'
```

You can think of `this` as an extra parameter that is passed in a different way. If you want to pass it explicitly, you can use a function's `call` method, which takes the `this` value as its first argument and treats further arguments as normal parameters.

```
speak.call(hungryRabbit, "Burp!");
// → The hungry rabbit says 'Burp!'
```

Since each function has its own `this` binding, whose value depends on the way it is called, you cannot refer to the `this` of the wrapping scope in a regular function defined with the `function` keyword.

Arrow functions are different—they do not bind their own `this` but can see the `this` binding of the scope around them. Thus, you can do something like the following code, which references `this` from inside a local function:

```
function normalize() {
  console.log(this.coords.map(n => n / this.length));
}
normalize.call({coords: [0, 2, 3], length: 5});
// → [0, 0.4, 0.6]
```

If I had written the argument to `map` using the `function` keyword, the code wouldn't work.

Prototypes

Watch closely.

```
let empty = {};
console.log(empty.toString);
// → function toString(){…}
console.log(empty.toString());
// → [object Object]
```

I pulled a property out of an empty object. Magic!

Well, not really. I have simply been withholding information about the way JavaScript objects work. In addition to their set of properties, most objects also have a *prototype*. A prototype is another object that is used as a fallback source of properties. When an object gets a request for a property that it does not have, its prototype will be searched for the property, then the prototype's prototype, and so on.

So who is the prototype of that empty object? It is the great ancestral prototype, the entity behind almost all objects, `Object.prototype`.

```
console.log(Object.getPrototypeOf({}) ==
            Object.prototype);
// → true
console.log(Object.getPrototypeOf(Object.prototype));
// → null
```

As you guess, `Object.getPrototypeOf` returns the prototype of an object.

The prototype relations of JavaScript objects form a tree-shaped structure, and at the root of this structure sits `Object.prototype`. It provides a few methods that show up in all objects, such as `toString`, which converts an object to a string representation.

Many objects don't directly have `Object.prototype` as their prototype but instead have another object that provides a different set of default properties. Functions derive from `Function.prototype`, and arrays derive from `Array.prototype`.

```
console.log(Object.getPrototypeOf(Math.max) ==
            Function.prototype);
// → true
console.log(Object.getPrototypeOf([]) ==
            Array.prototype);
// → true
```

Such a prototype object will itself have a prototype, often `Object.prototype`, so that it still indirectly provides methods like `toString`.

You can use `Object.create` to create an object with a specific prototype.

```
let protoRabbit = {
  speak(line) {
    console.log(`The ${this.type} rabbit says '${line}'`);
  }
};
let killerRabbit = Object.create(protoRabbit);
killerRabbit.type = "killer";
killerRabbit.speak("SKREEEE!");
// → The killer rabbit says 'SKREEEE!'
```

A property like `speak(line)` in an object expression is a shorthand way of defining a method. It creates a property called `speak` and gives it a function as its value.

The "proto" rabbit acts as a container for the properties that are shared by all rabbits. An individual rabbit object, like the killer rabbit, contains properties

that apply only to itself—in this case its type—and derives shared properties from its prototype.

Classes

JavaScript's prototype system can be interpreted as a somewhat informal take on an object-oriented concept called *classes*. A class defines the shape of a type of object—what methods and properties it has. Such an object is called an *instance* of the class.

Prototypes are useful for defining properties for which all instances of a class share the same value, such as methods. Properties that differ per instance, such as our rabbits' `type` property, need to be stored directly in the objects themselves.

So to create an instance of a given class, you have to make an object that derives from the proper prototype, but you *also* have to make sure it, itself, has the properties that instances of this class are supposed to have. This is what a *constructor* function does.

```
function makeRabbit(type) {
  let rabbit = Object.create(protoRabbit);
  rabbit.type = type;
  return rabbit;
}
```

JavaScript provides a way to make defining this type of function easier. If you put the keyword `new` in front of a function call, the function is treated as a constructor. This means that an object with the right prototype is automatically created, bound to `this` in the function, and returned at the end of the function.

The prototype object used when constructing objects is found by taking the `prototype` property of the constructor function.

```
function Rabbit(type) {
  this.type = type;
}
```

```
Rabbit.prototype.speak = function(line) {
  console.log(`The ${this.type} rabbit says '${line}'`);
};

let weirdRabbit = new Rabbit("weird");
```

Constructors (all functions, in fact) automatically get a property named `prototype`, which by default holds a plain, empty object that derives from `Object.prototype`. You can overwrite it with a new object if you want. Or you can add properties to the existing object, as the example does.

By convention, the names of constructors are capitalized so that they can easily be distinguished from other functions.

It is important to understand the distinction between the way a prototype is associated with a constructor (through its `prototype` property) and the way objects *have* a prototype (which can be found with `Object.getPrototypeOf`). The actual prototype of a constructor is `Function.prototype` since constructors are functions. Its `prototype` *property* holds the prototype used for instances created through it.

```
console.log(Object.getPrototypeOf(Rabbit) ==
            Function.prototype);
// → true
console.log(Object.getPrototypeOf(weirdRabbit) ==
            Rabbit.prototype);
// → true
```

Class notation

So JavaScript classes are constructor functions with a prototype property. That is how they work, and until 2015, that was how you had to write them. These days, we have a less awkward notation.

```
class Rabbit {
  constructor(type) {
    this.type = type;
  }
```

```
  speak(line) {
    console.log(`The ${this.type} rabbit says '${line}'`);
  }
}

let killerRabbit = new Rabbit("killer");
let blackRabbit = new Rabbit("black");
```

The `class` keyword starts a class declaration, which allows us to define a constructor and a set of methods all in a single place. Any number of methods may be written inside the declaration's braces. The one named `constructor` is treated specially. It provides the actual constructor function, which will be bound to the name `Rabbit`. The others are packaged into that constructor's prototype. Thus, the earlier class declaration is equivalent to the constructor definition from the previous section. It just looks nicer.

Class declarations currently allow only *methods*—properties that hold functions—to be added to the prototype. This can be somewhat inconvenient when you want to save a non-function value in there. The next version of the language will probably improve this. For now, you can create such properties by directly manipulating the prototype after you've defined the class.

Like `function`, `class` can be used both in statements and in expressions. When used as an expression, it doesn't define a binding but just produces the constructor as a value. You are allowed to omit the class name in a class expression.

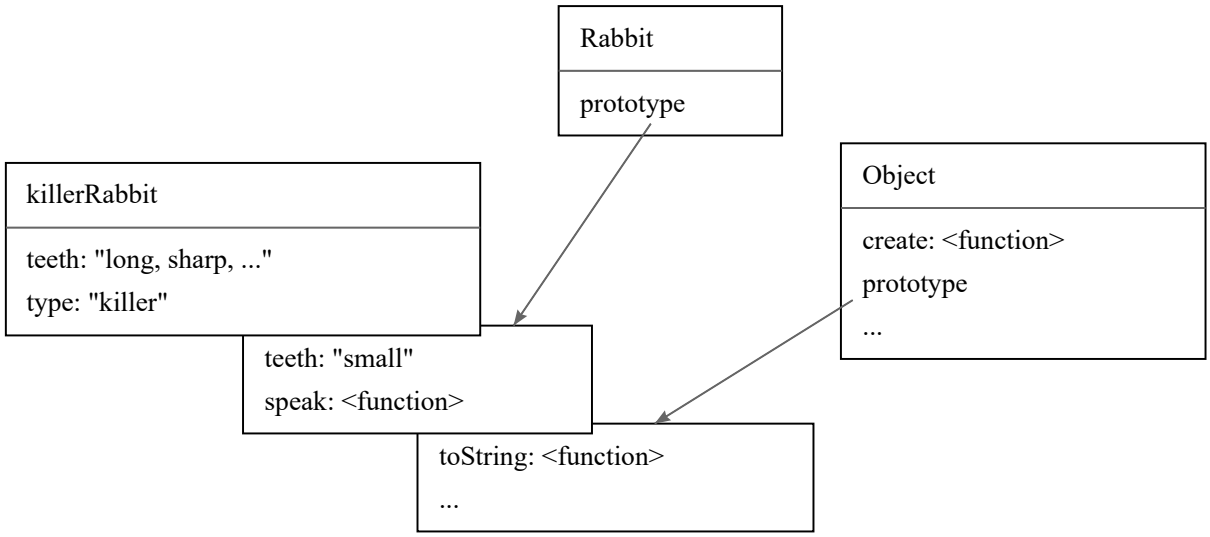
```
let object = new class { getWord() { return "hello"; } };
console.log(object.getWord());
// → hello
```

Overriding derived properties

When you add a property to an object, whether it is present in the prototype or not, the property is added to the object *itself*. If there was already a property with the same name in the prototype, this property will no longer affect the object, as it is now hidden behind the object's own property.

```
Rabbit.prototype.teeth = "small";
console.log(killerRabbit.teeth);
// → small
killerRabbit.teeth = "long, sharp, and bloody";
console.log(killerRabbit.teeth);
// → long, sharp, and bloody
console.log(blackRabbit.teeth);
// → small
console.log(Rabbit.prototype.teeth);
// → small
```

The following diagram sketches the situation after this code has run. The `Rabbit` and `Object` prototypes lie behind `killerRabbit` as a kind of backdrop, where properties that are not found in the object itself can be looked up.



Overriding properties that exist in a prototype can be a useful thing to do. As the rabbit teeth example shows, overriding can be used to express exceptional properties in instances of a more generic class of objects, while letting the nonexceptional objects take a standard value from their prototype.

Overriding is also used to give the standard function and array prototypes a different `toString` method than the basic object prototype.

```
console.log(Array.prototype.toString ==
            Object.prototype.toString);
// → false
```

```
console.log([1, 2].toString());
// → 1,2
```

Calling `toString` on an array gives a result similar to calling `.join(",")` on it—it puts commas between the values in the array. Directly calling `Object.prototype.toString` with an array produces a different string. That function doesn't know about arrays, so it simply puts the word *object* and the name of the type between square brackets.

```
console.log(Object.prototype.toString.call([1, 2]));
// → [object Array]
```

Maps

We saw the word *map* used in the previous chapter for an operation that transforms a data structure by applying a function to its elements. Confusing as it is, in programming the same word is also used for a related but rather different thing.

A *map* (noun) is a data structure that associates values (the keys) with other values. For example, you might want to map names to ages. It is possible to use objects for this.

```
let ages = {
  Boris: 39,
  Liang: 22,
  Júlia: 62
};

console.log(`Júlia is ${ages["Júlia"]}`);
// → Júlia is 62
console.log("Is Jack's age known?", "Jack" in ages);
// → Is Jack's age known? false
console.log("Is toString's age known?", "toString" in ages);
// → Is toString's age known? true
```

Here, the object's property names are the people's names, and the property values are their ages. But we certainly didn't list anybody named toString in

our map. Yet, because plain objects derive from `Object.prototype`, it looks like the property is there.

As such, using plain objects as maps is dangerous. There are several possible ways to avoid this problem. First, it is possible to create objects with *no* prototype. If you pass `null` to `Object.create`, the resulting object will not derive from `Object.prototype` and can safely be used as a map.

```
console.log("toString" in Object.create(null));
// → false
```

Object property names must be strings. If you need a map whose keys can't easily be converted to strings—such as objects—you cannot use an object as your map.

Fortunately, JavaScript comes with a class called `Map` that is written for this exact purpose. It stores a mapping and allows any type of keys.

```
let ages = new Map();
ages.set("Boris", 39);
ages.set("Liang", 22);
ages.set("Júlia", 62);

console.log(`Júlia is ${ages.get("Júlia")}`);
// → Júlia is 62
console.log("Is Jack's age known?", ages.has("Jack"));
// → Is Jack's age known? false
console.log(ages.has("toString"));
// → false
```

The methods `set`, `get`, and `has` are part of the interface of the `Map` object. Writing a data structure that can quickly update and search a large set of values isn't easy, but we don't have to worry about that. Someone else did it for us, and we can go through this simple interface to use their work.

If you do have a plain object that you need to treat as a map for some reason, it is useful to know that `Object.keys` returns only an object's *own* keys, not

those in the prototype. As an alternative to the `in` operator, you can use the `hasOwnProperty` method, which ignores the object's prototype.

```
console.log({x: 1}.hasOwnProperty("x"));
// → true
console.log({x: 1}.hasOwnProperty("toString"));
// → false
```

Polymorphism

When you call the `String` function (which converts a value to a string) on an object, it will call the `toString` method on that object to try to create a meaningful string from it. I mentioned that some of the standard prototypes define their own version of `toString` so they can create a string that contains more useful information than `"[object Object]"`. You can also do that yourself.

```
Rabbit.prototype.toString = function() {
  return `a ${this.type} rabbit`;
};

console.log(String(blackRabbit));
// → a black rabbit
```

This is a simple instance of a powerful idea. When a piece of code is written to work with objects that have a certain interface—in this case, a `toString` method—any kind of object that happens to support this interface can be plugged into the code, and it will just work.

This technique is called *polymorphism*. Polymorphic code can work with values of different shapes, as long as they support the interface it expects.

I mentioned in Chapter 4 that a `for/of` loop can loop over several kinds of data structures. This is another case of polymorphism—such loops expect the data structure to expose a specific interface, which arrays and strings do. And we can also add this interface to your own objects! But before we can do that, we need to know what symbols are.

Symbols

It is possible for multiple interfaces to use the same property name for different things. For example, I could define an interface in which the `toString` method is supposed to convert the object into a piece of yarn. It would not be possible for an object to conform to both that interface and the standard use of `toString`.

That would be a bad idea, and this problem isn't that common. Most JavaScript programmers simply don't think about it. But the language designers, whose *job* it is to think about this stuff, have provided us with a solution anyway.

When I claimed that property names are strings, that wasn't entirely accurate. They usually are, but they can also be *symbols*. Symbols are values created with the `Symbol` function. Unlike strings, newly created symbols are unique—you cannot create the same symbol twice.

```
let sym = Symbol("name");
console.log(sym == Symbol("name"));
// → false
Rabbit.prototype[sym] = 55;
console.log(blackRabbit[sym]);
// → 55
```

The string you pass to `Symbol` is included when you convert it to a string and can make it easier to recognize a symbol when, for example, showing it in the console. But it has no meaning beyond that—multiple symbols may have the same name.

Being both unique and usable as property names makes symbols suitable for defining interfaces that can peacefully live alongside other properties, no matter what their names are.

```
const toStringSymbol = Symbol("toString");
Array.prototype[toStringSymbol] = function() {
  return `${this.length} cm of blue yarn`;
};
```

```
console.log([1, 2].toString());
// → 1,2
console.log([1, 2][toStringSymbol]());
// → 2 cm of blue yarn
```

It is possible to include symbol properties in object expressions and classes by using square brackets around the property name. That causes the property name to be evaluated, much like the square bracket property access notation, which allows us to refer to a binding that holds the symbol.

```
let stringObject = {
  [toStringSymbol]() { return "a jute rope"; }
};
console.log(stringObject[toStringSymbol]());
// → a jute rope
```

The iterator interface

The object given to a `for/of` loop is expected to be *iterable*. This means it has a method named with the `Symbol.iterator` symbol (a symbol value defined by the language, stored as a property of the `Symbol` function).

When called, that method should return an object that provides a second interface, *iterator*. This is the actual thing that iterates. It has a `next` method that returns the next result. That result should be an object with a `value` property that provides the next value, if there is one, and a `done` property, which should be true when there are no more results and false otherwise.

Note that the `next`, `value`, and `done` property names are plain strings, not symbols. Only `Symbol.iterator`, which is likely to be added to a *lot* of different objects, is an actual symbol.

We can directly use this interface ourselves.

```
let okIterator = "OK"[Symbol.iterator]();
console.log(okIterator.next());
// → {value: "O", done: false}
```

```
console.log(okIterator.next());
// → {value: "K", done: false}
console.log(okIterator.next());
// → {value: undefined, done: true}
```

Let's implement an iterable data structure. We'll build a *matrix* class, acting as a two-dimensional array.

```
class Matrix {
  constructor(width, height, element = (x, y) => undefined) {
    this.width = width;
    this.height = height;
    this.content = [];

    for (let y = 0; y < height; y++) {
      for (let x = 0; x < width; x++) {
        this.content[y * width + x] = element(x, y);
      }
    }
  }

  get(x, y) {
    return this.content[y * this.width + x];
  }
  set(x, y, value) {
    this.content[y * this.width + x] = value;
  }
}
```

The class stores its content in a single array of *width* × *height* elements. The elements are stored row by row, so, for example, the third element in the fifth row is (using zero-based indexing) stored at position 4 × *width* + 2.

The constructor function takes a width, a height, and an optional `element` function that will be used to fill in the initial values. There are `get` and `set` methods to retrieve and update elements in the matrix.

When looping over a matrix, you are usually interested in the position of the elements as well as the elements themselves, so we'll have our iterator produce

objects with `x`, `y`, and `value` properties.

```
class MatrixIterator {
  constructor(matrix) {
    this.x = 0;
    this.y = 0;
    this.matrix = matrix;
  }

  next() {
    if (this.y == this.matrix.height) return {done: true};

    let value = {x: this.x,
                 y: this.y,
                 value: this.matrix.get(this.x, this.y)};
    this.x++;
    if (this.x == this.matrix.width) {
      this.x = 0;
      this.y++;
    }
    return {value, done: false};
  }
}
```

The class tracks the progress of iterating over a matrix in its `x` and `y` properties. The `next` method starts by checking whether the bottom of the matrix has been reached. If it hasn't, it *first* creates the object holding the current value and *then* updates its position, moving to the next row if necessary.

Let's set up the `Matrix` class to be iterable. Throughout this book, I'll occasionally use after-the-fact prototype manipulation to add methods to classes so that the individual pieces of code remain small and self-contained. In a regular program, where there is no need to split the code into small pieces, you'd declare these methods directly in the class instead.

```
Matrix.prototype[Symbol.iterator] = function() {
  return new MatrixIterator(this);
};
```

We can now loop over a matrix with `for/of`.

```
let matrix = new Matrix(2, 2, (x, y) => `value ${x},${y}`);
for (let {x, y, value} of matrix) {
  console.log(x, y, value);
}
// → 0 0 value 0,0
// → 1 0 value 1,0
// → 0 1 value 0,1
// → 1 1 value 1,1
```

Getters, setters, and statics

Interfaces often consist mostly of methods, but it is also okay to include properties that hold non-function values. For example, `Map` objects have a `size` property that tells you how many keys are stored in them.

It is not even necessary for such an object to compute and store such a property directly in the instance. Even properties that are accessed directly may hide a method call. Such methods are called *getters*, and they are defined by writing `get` in front of the method name in an object expression or class declaration.

```
let varyingSize = {
  get size() {
    return Math.floor(Math.random() * 100);
  }
};

console.log(varyingSize.size);
// → 73
console.log(varyingSize.size);
// → 49
```

Whenever someone reads from this object's `size` property, the associated method is called. You can do a similar thing when a property is written to, using a *setter*.

```
class Temperature {
  constructor(celsius) {
```

```
    this.celsius = celsius;
  }
  get fahrenheit() {
    return this.celsius * 1.8 + 32;
  }
  set fahrenheit(value) {
    this.celsius = (value - 32) / 1.8;
  }

  static fromFahrenheit(value) {
    return new Temperature((value - 32) / 1.8);
  }
}

let temp = new Temperature(22);
console.log(temp.fahrenheit);
// → 71.6
temp.fahrenheit = 86;
console.log(temp.celsius);
// → 30
```

The `Temperature` class allows you to read and write the temperature in either degrees Celsius or degrees Fahrenheit, but internally it stores only Celsius and automatically converts to and from Celsius in the `fahrenheit` getter and setter.

Sometimes you want to attach some properties directly to your constructor function, rather than to the prototype. Such methods won't have access to a class instance but can, for example, be used to provide additional ways to create instances.

Inside a class declaration, methods that have `static` written before their name are stored on the constructor. So the `Temperature` class allows you to write `Temperature.fromFahrenheit(100)` to create a temperature using degrees Fahrenheit.

Inheritance

Some matrices are known to be *symmetric*. If you mirror a symmetric matrix around its top-left-to-bottom-right diagonal, it stays the same. In other words, the value stored at *x*,*y* is always the same as that at *y*,*x*.

Imagine we need a data structure like `Matrix` but one that enforces the fact that the matrix is and remains symmetrical. We could write it from scratch, but that would involve repeating some code very similar to what we already wrote.

JavaScript's prototype system makes it possible to create a *new* class, much like the old class, but with new definitions for some of its properties. The prototype for the new class derives from the old prototype but adds a new definition for, say, the `set` method.

In object-oriented programming terms, this is called *inheritance*. The new class inherits properties and behavior from the old class.

```
class SymmetricMatrix extends Matrix {
  constructor(size, element = (x, y) => undefined) {
    super(size, size, (x, y) => {
      if (x < y) return element(y, x);
      else return element(x, y);
    });
  }

  set(x, y, value) {
    super.set(x, y, value);
    if (x != y) {
      super.set(y, x, value);
    }
  }
}

let matrix = new SymmetricMatrix(5, (x, y) => `${x},${y}`);
console.log(matrix.get(2, 3));
// → 3,2
```

The use of the word `extends` indicates that this class shouldn't be directly based on the default `Object` prototype but on some other class. This is called

the *superclass*. The derived class is the *subclass*.

To initialize a `SymmetricMatrix` instance, the constructor calls its superclass's constructor through the `super` keyword. This is necessary because if this new object is to behave (roughly) like a `Matrix`, it is going to need the instance properties that matrices have. To ensure the matrix is symmetrical, the constructor wraps the `element` function to swap the coordinates for values below the diagonal.

The `set` method again uses `super` but this time not to call the constructor but to call a specific method from the superclass's set of methods. We are redefining `set` but do want to use the original behavior. Because `this.set` refers to the *new* `set` method, calling that wouldn't work. Inside class methods, `super` provides a way to call methods as they were defined in the superclass.

Inheritance allows us to build slightly different data types from existing data types with relatively little work. It is a fundamental part of the object-oriented tradition, alongside encapsulation and polymorphism. But while the latter two are now generally regarded as wonderful ideas, inheritance is more controversial.

Whereas encapsulation and polymorphism can be used to *separate* pieces of code from each other, reducing the tangledness of the overall program, inheritance fundamentally ties classes together, creating *more* tangle. When inheriting from a class, you usually have to know more about how it works than when simply using it. Inheritance can be a useful tool, and I use it now and then in my own programs, but it shouldn't be the first tool you reach for, and you probably shouldn't actively go looking for opportunities to construct class hierarchies (family trees of classes).

The instanceof operator

It is occasionally useful to know whether an object was derived from a specific class. For this, JavaScript provides a binary operator called `instanceof`.

```
console.log(
  new SymmetricMatrix(2) instanceof SymmetricMatrix);
// → true
console.log(new SymmetricMatrix(2) instanceof Matrix);
// → true
console.log(new Matrix(2, 2) instanceof SymmetricMatrix);
// → false
console.log([1] instanceof Array);
// → true
```

The operator will see through inherited types, so a `SymmetricMatrix` is an instance of `Matrix`. The operator can also be applied to standard constructors like `Array`. Almost every object is an instance of `Object`.

Summary

So objects do more than just hold their own properties. They have prototypes, which are other objects. They'll act as if they have properties they don't have as long as their prototype has that property. Simple objects have `Object.prototype` as their prototype.

Constructors, which are functions whose names usually start with a capital letter, can be used with the `new` operator to create new objects. The new object's prototype will be the object found in the `prototype` property of the constructor. You can make good use of this by putting the properties that all values of a given type share into their prototype. There's a `class` notation that provides a clear way to define a constructor and its prototype.

You can define getters and setters to secretly call methods every time an object's property is accessed. Static methods are methods stored in a class's constructor, rather than its prototype.

The `instanceof` operator can, given an object and a constructor, tell you whether that object is an instance of that constructor.

One useful thing to do with objects is to specify an interface for them and tell everybody that they are supposed to talk to your object only through that

interface. The rest of the details that make up your object are now *encapsulated*, hidden behind the interface.

More than one type may implement the same interface. Code written to use an interface automatically knows how to work with any number of different objects that provide the interface. This is called *polymorphism*.

When implementing multiple classes that differ in only some details, it can be helpful to write the new classes as *subclasses* of an existing class, *inheriting* part of its behavior.

Exercises

A vector type

Write a class `Vec` that represents a vector in two-dimensional space. It takes `x` and `y` parameters (numbers), which it should save to properties of the same name.

Give the `Vec` prototype two methods, `plus` and `minus`, that take another vector as a parameter and return a new vector that has the sum or difference of the two vectors' (`this` and the parameter) *x* and *y* values.

Add a getter property `length` to the prototype that computes the length of the vector—that is, the distance of the point (*x*, *y*) from the origin (0, 0).

```
// Your code here.

console.log(new Vec(1, 2).plus(new Vec(2, 3)));
// → Vec{x: 3, y: 5}
console.log(new Vec(1, 2).minus(new Vec(2, 3)));
// → Vec{x: -1, y: -1}
console.log(new Vec(3, 4).length);
// → 5
```

» Display hints...

Groups

The standard JavaScript environment provides another data structure called `Set`. Like an instance of `Map`, a set holds a collection of values. Unlike `Map`, it does not associate other values with those—it just tracks which values are part of the set. A value can be part of a set only once—adding it again doesn't have any effect.

Write a class called `Group` (since `Set` is already taken). Like `Set`, it has `add`, `delete`, and `has` methods. Its constructor creates an empty group, `add` adds a value to the group (but only if it isn't already a member), `delete` removes its argument from the group (if it was a member), and `has` returns a Boolean value indicating whether its argument is a member of the group.

Use the `===` operator, or something equivalent such as `indexOf`, to determine whether two values are the same.

Give the class a static `from` method that takes an iterable object as argument and creates a group that contains all the values produced by iterating over it.

```
class Group {
  // Your code here.
}

let group = Group.from([10, 20]);
console.log(group.has(10));
// → true
console.log(group.has(30));
// → false
group.add(10);
group.delete(10);
console.log(group.has(10));
// → false
```

» Display hints...

ITERABLE GROUPS

Make the `Group` class from the previous exercise iterable. Refer to the section about the iterator interface earlier in the chapter if you aren't clear on the exact

form of the interface anymore.

If you used an array to represent the group's members, don't just return the iterator created by calling the `Symbol.iterator` method on the array. That would work, but it defeats the purpose of this exercise.

It is okay if your iterator behaves strangely when the group is modified during iteration.

```
// Your code here (and the code from the previous exercise)

for (let value of Group.from(["a", "b", "c"])) {
  console.log(value);
}
// → a
// → b
// → c
```

» Display hints...

Borrowing a method

Earlier in the chapter I mentioned that an object's `hasOwnProperty` can be used as a more robust alternative to the `in` operator when you want to ignore the prototype's properties. But what if your map needs to include the word `"hasOwnProperty"`? You won't be able to call that method anymore because the object's own property hides the method value.

Can you think of a way to call `hasOwnProperty` on an object that has its own property by that name?

```
let map = {one: true, two: true, hasOwnProperty: true};

// Fix this call
console.log(map.hasOwnProperty("one"));
// → true
```

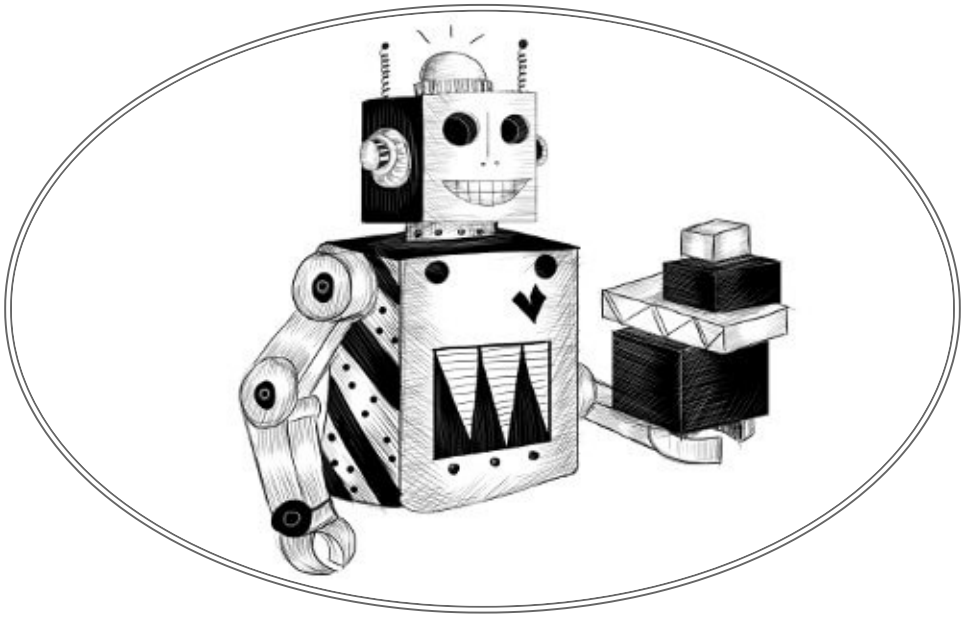
» Display hints...

CHAPTER 7

◀ ◆ ▶

PROJECT: A ROBOT

> "[...] the question of whether Machines Can Think [...] is about as relevant as the question of whether Submarines Can Swim."
>
> — Edsger Dijkstra, *The Threats to Computing Science*



In "project" chapters, I'll stop pummeling you with new theory for a brief moment, and instead we'll work through a program together. Theory is necessary to learn to program, but reading and understanding actual programs is just as important.

Our project in this chapter is to build an automaton, a little program that performs a task in a virtual world. Our automaton will be a mail-delivery robot picking up and dropping off parcels.

MEADOWFIELD

The village of Meadowfield isn't very big. It consists of 11 places with 14 roads between them. It can be described with this array of roads:

edit & run code by clicking it

```
const roads = [
  "Alice's House-Bob's House",   "Alice's House-Cabin",
  "Alice's House-Post Office",   "Bob's House-Town Hall",
  "Daria's House-Ernie's House", "Daria's House-Town Hall",
  "Ernie's House-Grete's House", "Grete's House-Farm",
  "Grete's House-Shop",          "Marketplace-Farm",
  "Marketplace-Post Office",     "Marketplace-Shop",
  "Marketplace-Town Hall",       "Shop-Town Hall"
];
```

The network of roads in the village forms a *graph*. A graph is a collection of points (places in the village) with lines between them (roads). This graph will be the world that our robot moves through.

The array of strings isn't very easy to work with. What we're interested in is the destinations that we can reach from a given place. Let's convert the list of roads to a data structure that, for each place, tells us what can be reached from there.

```
function buildGraph(edges) {
  let graph = Object.create(null);
  function addEdge(from, to) {
    if (graph[from] == null) {
      graph[from] = [to];
    } else {
      graph[from].push(to);
    }
```

```
    }
    for (let [from, to] of edges.map(r => r.split("-"))) {
      addEdge(from, to);
      addEdge(to, from);
    }
    return graph;
  }

  const roadGraph = buildGraph(roads);
```

Given an array of edges, `buildGraph` creates a map object that, for each node, stores an array of connected nodes.

It uses the `split` method to go from the road strings, which have the form `"Start-End"`, to two-element arrays containing the start and end as separate strings.

THE TASK

Our robot will be moving around the village. There are parcels in various places, each addressed to some other place. The robot picks up parcels when it comes to them and delivers them when it arrives at their destinations.

The automaton must decide, at each point, where to go next. It has finished its task when all parcels have been delivered.

To be able to simulate this process, we must define a virtual world that can describe it. This model tells us where the robot is and where the parcels are. When the robot has decided to move somewhere, we need to update the model to reflect the new situation.

If you're thinking in terms of object-oriented programming, your first impulse might be to start defining objects for the various elements in the world: a class for the robot, one for a parcel, maybe one for places. These could then hold properties that describe their current state, such as the pile of parcels at a location, which we could change when updating the world.

This is wrong.

At least, it usually is. The fact that something sounds like an object does not automatically mean that it should be an object in your program. Reflexively writing classes for every concept in your application tends to leave you with a collection of interconnected objects that each have their own internal, changing state. Such programs are often hard to understand and thus easy to break.

Instead, let's condense the village's state down to the minimal set of values that define it. There's the robot's current location and the collection of undelivered parcels, each of which has a current location and a destination address. That's it.

And while we're at it, let's make it so that we don't *change* this state when the robot moves but rather compute a *new* state for the situation after the move.

```
class VillageState {
  constructor(place, parcels) {
    this.place = place;
    this.parcels = parcels;
  }

  move(destination) {
    if (!roadGraph[this.place].includes(destination)) {
      return this;
    } else {
      let parcels = this.parcels.map(p => {
        if (p.place != this.place) return p;
        return {place: destination, address: p.address};
      }).filter(p => p.place != p.address);
      return new VillageState(destination, parcels);
    }
  }
}
```

The `move` method is where the action happens. It first checks whether there is a road going from the current place to the destination, and if not, it returns the old state since this is not a valid move.

Then it creates a new state with the destination as the robot's new place. But it also needs to create a new set of parcels—parcels that the robot is carrying (that are at the robot's current place) need to be moved along to the new place. And parcels that are addressed to the new place need to be delivered—that is, they need to be removed from the set of undelivered parcels. The call to `map` takes care of the moving, and the call to `filter` does the delivering.

Parcel objects aren't changed when they are moved but re-created. The `move` method gives us a new village state but leaves the old one entirely intact.

```
let first = new VillageState(
  "Post Office",
  [{place: "Post Office", address: "Alice's House"}]
);
let next = first.move("Alice's House");

console.log(next.place);
// → Alice's House
console.log(next.parcels);
// → []
console.log(first.place);
// → Post Office
```

The move causes the parcel to be delivered, and this is reflected in the next state. But the initial state still describes the situation where the robot is at the post office and the parcel is undelivered.

Persistent data

Data structures that don't change are called *immutable* or *persistent*. They behave a lot like strings and numbers in that they are who they are and stay that way, rather than containing different things at different times.

In JavaScript, just about everything *can* be changed, so working with values that are supposed to be persistent requires some restraint. There is a function called `Object.freeze` that changes an object so that writing to its properties is ignored. You could use that to make sure your objects aren't changed, if you

want to be careful. Freezing does require the computer to do some extra work, and having updates ignored is just about as likely to confuse someone as having them do the wrong thing. So I usually prefer to just tell people that a given object shouldn't be messed with and hope they remember it.

```
let object = Object.freeze({value: 5});
object.value = 10;
console.log(object.value);
// → 5
```

Why am I going out of my way to not change objects when the language is obviously expecting me to?

Because it helps me understand my programs. This is about complexity management again. When the objects in my system are fixed, stable things, I can consider operations on them in isolation—moving to Alice's house from a given start state always produces the same new state. When objects change over time, that adds a whole new dimension of complexity to this kind of reasoning.

For a small system like the one we are building in this chapter, we could handle that bit of extra complexity. But the most important limit on what kind of systems we can build is how much we can understand. Anything that makes your code easier to understand makes it possible to build a more ambitious system.

Unfortunately, although understanding a system built on persistent data structures is easier, *designing* one, especially when your programming language isn't helping, can be a little harder. We'll look for opportunities to use persistent data structures in this book, but we'll also be using changeable ones.

Simulation

A delivery robot looks at the world and decides in which direction it wants to move. As such, we could say that a robot is a function that takes a `VillageState` object and returns the name of a nearby place.

Because we want robots to be able to remember things, so that they can make and execute plans, we also pass them their memory and allow them to return a new memory. Thus, the thing a robot returns is an object containing both the direction it wants to move in and a memory value that will be given back to it the next time it is called.

```
function runRobot(state, robot, memory) {
  for (let turn = 0;; turn++) {
    if (state.parcels.length == 0) {
      console.log(`Done in ${turn} turns`);
      break;
    }
    let action = robot(state, memory);
    state = state.move(action.direction);
    memory = action.memory;
    console.log(`Moved to ${action.direction}`);
  }
}
```

Consider what a robot has to do to “solve” a given state. It must pick up all parcels by visiting every location that has a parcel and deliver them by visiting every location that a parcel is addressed to, but only after picking up the parcel.

What is the dumbest strategy that could possibly work? The robot could just walk in a random direction every turn. That means, with great likelihood, it will eventually run into all parcels and then also at some point reach the place where they should be delivered.

Here’s what that could look like:

```
function randomPick(array) {
  let choice = Math.floor(Math.random() * array.length);
  return array[choice];
}

function randomRobot(state) {
  return {direction: randomPick(roadGraph[state.place])};
}
```

Remember that `Math.random()` returns a number between zero and one—but always below one. Multiplying such a number by the length of an array and then applying `Math.floor` to it gives us a random index for the array.

Since this robot does not need to remember anything, it ignores its second argument (remember that JavaScript functions can be called with extra arguments without ill effects) and omits the `memory` property in its returned object.

To put this sophisticated robot to work, we'll first need a way to create a new state with some parcels. A static method (written here by directly adding a property to the constructor) is a good place to put that functionality.

```
VillageState.random = function(parcelCount = 5) {
  let parcels = [];
  for (let i = 0; i < parcelCount; i++) {
    let address = randomPick(Object.keys(roadGraph));
    let place;
    do {
      place = randomPick(Object.keys(roadGraph));
    } while (place == address);
    parcels.push({place, address});
  }
  return new VillageState("Post Office", parcels);
};
```

We don't want any parcels that are sent from the same place that they are addressed to. For this reason, the `do` loop keeps picking new places when it gets one that's equal to the address.

Let's start up a virtual world.

```
runRobot(VillageState.random(), randomRobot);
// → Moved to Marketplace
// → Moved to Town Hall
// → …
// → Done in 63 turns
```

It takes the robot a lot of turns to deliver the parcels because it isn't planning ahead very well. We'll address that soon.

For a more pleasant perspective on the simulation, you can use the `runRobotAnimation` function that's available in this chapter's programming environment. This runs the simulation, but instead of outputting text, it shows you the robot moving around the village map.

```
runRobotAnimation(VillageState.random(), randomRobot);
```

The way `runRobotAnimation` is implemented will remain a mystery for now, but after you've read the later chapters of this book, which discuss JavaScript integration in web browsers, you'll be able to guess how it works.

The mail truck's route

We should be able to do a lot better than the random robot. An easy improvement would be to take a hint from the way real-world mail delivery works. If we find a route that passes all places in the village, the robot could run that route twice, at which point it is guaranteed to be done. Here is one such route (starting from the post office):

```
const mailRoute = [
  "Alice's House", "Cabin", "Alice's House", "Bob's House",
  "Town Hall", "Daria's House", "Ernie's House",
  "Grete's House", "Shop", "Grete's House", "Farm",
  "Marketplace", "Post Office"
];
```

To implement the route-following robot, we'll need to make use of robot memory. The robot keeps the rest of its route in its memory and drops the first element every turn.

```
function routeRobot(state, memory) {
  if (memory.length == 0) {
    memory = mailRoute;
  }
```

```
  return {direction: memory[0], memory: memory.slice(1)};
}
```

This robot is a lot faster already. It'll take a maximum of 26 turns (twice the 13-step route) but usually less.

```
runRobotAnimation(VillageState.random(), routeRobot, []);
```

Pathfinding

Still, I wouldn't really call blindly following a fixed route intelligent behavior. The robot could work more efficiently if it adjusted its behavior to the actual work that needs to be done.

To do that, it has to be able to deliberately move toward a given parcel or toward the location where a parcel has to be delivered. Doing that, even when the goal is more than one move away, will require some kind of route-finding function.

The problem of finding a route through a graph is a typical *search problem*. We can tell whether a given solution (a route) is a valid solution, but we can't directly compute the solution the way we could for 2 + 2. Instead, we have to keep creating potential solutions until we find one that works.

The number of possible routes through a graph is infinite. But when searching for a route from *A* to *B*, we are interested only in the ones that start at *A*. We also don't care about routes that visit the same place twice—those are definitely not the most efficient route anywhere. So that cuts down on the number of routes that the route finder has to consider.

In fact, we are mostly interested in the *shortest* route. So we want to make sure we look at short routes before we look at longer ones. A good approach would be to "grow" routes from the starting point, exploring every reachable place that hasn't been visited yet, until a route reaches the goal. That way, we'll only explore routes that are potentially interesting, and we'll find the shortest route (or one of the shortest routes, if there are more than one) to the goal.

Here is a function that does this:

```
function findRoute(graph, from, to) {
  let work = [{at: from, route: []}];
  for (let i = 0; i < work.length; i++) {
    let {at, route} = work[i];
    for (let place of graph[at]) {
      if (place == to) return route.concat(place);
      if (!work.some(w => w.at == place)) {
        work.push({at: place, route: route.concat(place)});
      }
    }
  }
}
```

The exploring has to be done in the right order—the places that were reached first have to be explored first. We can't immediately explore a place as soon as we reach it because that would mean places reached *from there* would also be explored immediately, and so on, even though there may be other, shorter paths that haven't yet been explored.

Therefore, the function keeps a *work list*. This is an array of places that should be explored next, along with the route that got us there. It starts with just the start position and an empty route.

The search then operates by taking the next item in the list and exploring that, which means all roads going from that place are looked at. If one of them is the goal, a finished route can be returned. Otherwise, if we haven't looked at this place before, a new item is added to the list. If we have looked at it before, since we are looking at short routes first, we've found either a longer route to that place or one precisely as long as the existing one, and we don't need to explore it.

You can visually imagine this as a web of known routes crawling out from the start location, growing evenly on all sides (but never tangling back into itself). As soon as the first thread reaches the goal location, that thread is traced back to the start, giving us our route.

Our code doesn't handle the situation where there are no more work items on the work list because we know that our graph is *connected*, meaning that every location can be reached from all other locations. We'll always be able to find a route between two points, and the search can't fail.

```
function goalOrientedRobot({place, parcels}, route) {
  if (route.length == 0) {
    let parcel = parcels[0];
    if (parcel.place != place) {
      route = findRoute(roadGraph, place, parcel.place);
    } else {
      route = findRoute(roadGraph, place, parcel.address);
    }
  }
  return {direction: route[0], memory: route.slice(1)};
}
```

This robot uses its memory value as a list of directions to move in, just like the route-following robot. Whenever that list is empty, it has to figure out what to do next. It takes the first undelivered parcel in the set and, if that parcel hasn't been picked up yet, plots a route toward it. If the parcel *has* been picked up, it still needs to be delivered, so the robot creates a route toward the delivery address instead.

Let's see how it does.

```
runRobotAnimation(VillageState.random(),
                  goalOrientedRobot, []);
```

This robot usually finishes the task of delivering 5 parcels in about 16 turns. That's slightly better than `routeRobot` but still definitely not optimal.

Exercises

Measuring a robot

It's hard to objectively compare robots by just letting them solve a few scenarios. Maybe one robot just happened to get easier tasks or the kind of

tasks that it is good at, whereas the other didn't.

Write a function `compareRobots` that takes two robots (and their starting memory). It should generate 100 tasks and let each of the robots solve each of these tasks. When done, it should output the average number of steps each robot took per task.

For the sake of fairness, make sure you give each task to both robots, rather than generating different tasks per robot.

```
function compareRobots(robot1, memory1, robot2, memory2) {
  // Your code here
}

compareRobots(routeRobot, [], goalOrientedRobot, []);
```

» Display hints...

Robot efficiency

Can you write a robot that finishes the delivery task faster than `goalOrientedRobot`? If you observe that robot's behavior, what obviously stupid things does it do? How could those be improved?

If you solved the previous exercise, you might want to use your `compareRobots` function to verify whether you improved the robot.

```
// Your code here

runRobotAnimation(VillageState.random(), yourRobot, memory);
```

» Display hints...

Persistent group

Most data structures provided in a standard JavaScript environment aren't very well suited for persistent use. Arrays have `slice` and `concat` methods, which allow us to easily create new arrays without damaging the old one. But

`Set`, for example, has no methods for creating a new set with an item added or removed.

Write a new class `PGroup`, similar to the `Group` class from Chapter 6, which stores a set of values. Like `Group`, it has `add`, `delete`, and `has` methods.

Its `add` method, however, should return a *new* `PGroup` instance with the given member added and leave the old one unchanged. Similarly, `delete` creates a new instance without a given member.

The class should work for values of any type, not just strings. It does *not* have to be efficient when used with large amounts of values.

The constructor shouldn't be part of the class's interface (though you'll definitely want to use it internally). Instead, there is an empty instance, `PGroup.empty`, that can be used as a starting value.

Why do you need only one `PGroup.empty` value, rather than having a function that creates a new, empty map every time?

```
class PGroup {
  // Your code here
}

let a = PGroup.empty.add("a");
let ab = a.add("b");
let b = ab.delete("a");

console.log(b.has("b"));
// → true
console.log(a.has("b"));
// → false
console.log(b.has("a"));
// → false
```

» Display hints...