

ELOQUENT JAVASCRIPT

FOURTH EDITION

Marijn Haverbeke



ELOQUENT JAVASCRIPT

4TH EDITION

Written by Marijn Haverbeke.

Licensed under a [Creative Commons attribution-noncommercial license](#). All code in this book may also be considered licensed under an [MIT license](#).

Illustrations by various artists: Cover by Péchane Sumi-e. Chapter illustrations by Madalina Tantareanu. Pixel art in Chapters 7 and 16 by Antonio Perdomo Pastor. Regular expression diagrams in Chapter 9 generated with [regexper.com](#) by Jeff Avallone. Game concept for Chapter 16 by [Thomas Palef](#).

A paper version of Eloquent JavaScript, including a bonus chapter, is being brought out by [No Starch Press](#). They also sell a more polished EPUB version that includes the bonus chapter.

TABLE OF CONTENTS

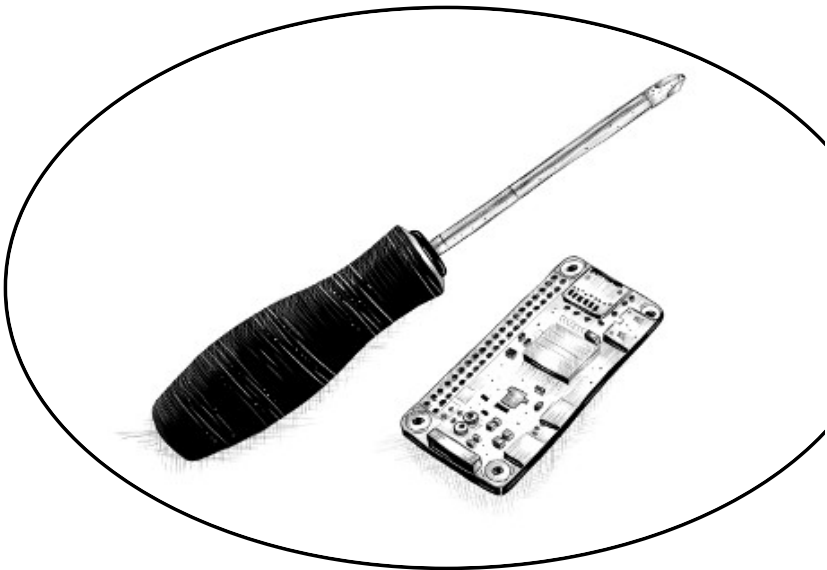
1. [Introduction](#)
2. (Part 1: Language)Values, Types, and Operators
3. Program Structure
4. Functions
5. [Data Structures: Objects and Arrays](#)
6. Higher-order Functions
7. The Secret Life of Objects
8. Project: A Robot
9. Bugs and Errors
10. [Regular Expressions](#)
11. Modules
12. Asynchronous Programming
13. Project: A Programming Language
14. (Part 2: Browser)JavaScript and the Browser
15. [The Document Object Model](#)
16. Handling Events
17. Project: A Platform Game
18. [Drawing on Canvas](#)
19. HTTP and Forms
20. Project: A Pixel Art Editor
21. (Part 3: Node)Node.js
22. Project: Skill-Sharing Website
23. Hints to the exercises

1. [Introduction](#)
2. [Data Structures: Objects and Arrays](#)
3. [Regular Expressions](#)
4. [The Document Object Model](#)
5. [Drawing on Canvas](#)

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 often is no appropriate 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. On top of that, it makes for a wonderful game of puzzle solving and 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. For routine computer use, these have largely been replaced with visual interfaces, which are easier to learn but offer less freedom. But if you know where to look, the languages are still there. One of them, *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 can easily 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 further learning easier.

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

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, and, at the same time, it controls the actions performed on this memory. Analogies that try to compare programs to familiar objects tend to fall short. A superficially fitting one is to compare a program to 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 as a program grows, so does its complexity. The skill of programming is the skill of building programs that don't confuse the programmer. The best programs are those that manage to do something interesting while still being easy to understand.

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 at least once so that you understand them. A sense of what a good program looks like is developed with 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

```

This is a program to add the numbers from 1 to 10 together and print the result: $1 + 2 + \dots + 10 = 55$. It could run on a simple hypothetical machine. To program early computers, it was necessary to set large arrays of switches in the right position or punch holes in strips of cardboard and feed them to the computer. You can 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 0, 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 most confusing 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 test only whether a number is zero and make a decision based on that. It therefore 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 decides 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 importantly, 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 “the 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—that is, 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 could make sure they actually provided the same language. This is called the ECMAScript standard, after the Ecma International organization that conducted 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 room for techniques that are impossible in more rigid languages and makes for a pleasant, informal style of programming. After learning the language properly and working with it for a while, I have come to actually *like* JavaScript.

There have been several versions of JavaScript. ECMAScript version 3 was the widely supported version during 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 version 4 was abandoned in 2008. A much less ambitious version 5, which made only some uncontroversial improvements, came out in 2009. 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 JavaScript is evolving means that browsers have to constantly keep up. 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 2024 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.

The easiest way to run the example code in the book—and to experiment with it—is to look it up in the online version of the book at <https://eloquentjavascript.net>. There, you can click any code example to edit and run it and to see the output it produces. To work on the exercises, go to <https://eloquentjavascript.net/code>, which provides starting code for each coding exercise and allows you to look at the solutions.

Running the programs defined in this book outside of the book's website requires some care. 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. There are five *project chapters* in the book that describe larger example programs to give you a taste of actual programming.

The language part of the book starts with four chapters that introduce the basic structure of the JavaScript language. They discuss 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 that builds a crude delivery robot, 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, where we implement a programming language, concludes the first part of the book.

The second part of the book, 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: building a platform game and a pixel paint program.

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 these 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!

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 from which data structures are built. 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. After learning the basics of data structures in this chapter, 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 introduced earlier in the book.

The online coding sandbox for the book (<https://eloquentjavascript.net/code>) provides a way to run code in the context of a particular chapter. If you decide to work through the examples in another environment, be sure to first download the full code for this chapter from the sandbox page.

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 of waking 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.

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. However, avoiding oak trees did not solve 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.

DATASETS

To work with a chunk of digital data, we 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. We'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

```

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 number of items to skip, counting from the start of the array.

PROPERTIES

We’ve seen a few expressions like `myString.length` (to get the length of a string) and `Math.max` (the maximum function) in past chapters. These expressions 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]` takes the value of the variable named `x` and uses that, converted to a string, as the property name.

If you know that the property in which you are interested 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—starting with a letter or underscore, and containing only letters, numbers, and underscores. 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.

Just like strings, arrays have a `length` property that tells us how many elements the array has.

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. You’ll find out how this works 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. 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. Stacks 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, you write 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 as shown in this example helps with readability. Properties whose names aren't valid binding names or valid numbers must 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 begin 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. The one case where this does come up is when you want to return an object from a shorthand arrow function—you can't write `n => {prop: n}` since the braces will be interpreted as a function body. Instead, you have to put a set of parentheses around the object to make it clear that it is an expression.

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 can think of objects as octopuses with any number of tentacles, each of which has a name written 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. Give the function an object and it will return 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 visualize arrays as long, flat octopuses with all their tentacles in a neat row, labeled with numbers.

Jacques 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 soon, but 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 at which the binding points. 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 that 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

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: `events`. 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.

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:

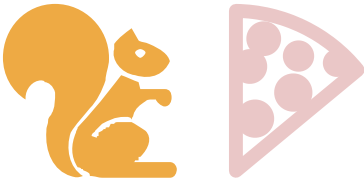
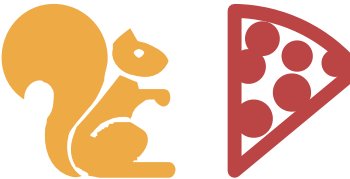
```
addEntry(["work", "touched tree", "pizza", "running",  
  "television"], false);  
addEntry(["work", "ice cream", "cauliflower", "lasagna",  
  "touched tree", "brushed teeth"], false);  
addEntry(["weekend", "cycling", "break", "peanuts",  
  "beer"], true);
```

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 1 indicates that the two are perfectly related—if you know one, you also know the other. Negative 1 also means that the variables are perfectly related but 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 number of times that combination occurred in our measurements.

 No squirrel, no pizza 76	 No squirrel, pizza 9
 Squirrel, no pizza 4	 Squirrel, pizza 1

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

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

(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 , refers to the sum of all measurements where the first variable is true, which is 5 in the example table. Likewise, n_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 $1 \times 76 - 4 \times 9 = 40$, and the part below it (the divisor) would be the square root of $5 \times 85 \times 10 \times 80$, or $\sqrt{340,000}$. 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 keeps his journal for three months. The resulting dataset is available in the [coding sandbox](#) for this chapter (<https://eloquentjavascript.net/code#4>), 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 uses the word `of` after its 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 dataset. 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 adding any event names that aren't already in it to the `events` array, the function collects every type of event.

Using that function, 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. The transformations *do* 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 clearly stronger than the others. Eating peanuts has a strong positive effect on the chance of turning into a squirrel, whereas brushing 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 never even have noticed his affliction.

Knowing this, Jacques stops eating peanuts altogether and finds that his transformations stop.

But it takes only a few months for him to notice that something is missing from this entirely human way of living. Without his feral adventures, Jacques hardly feels alive at all. He decides he'd rather be a full-time wild animal. After building a beautiful little tree house in the forest and equipping it with a peanut butter dispenser and a ten-year supply of peanut butter, he changes form one last time, and lives the short and energetic life of a squirrel.

FURTHER ARRAYOLOGY

Before finishing the chapter, I want to introduce you to a few more object-related concepts. I'll start with 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);  
}
```

This 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 and the end index is 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 append 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 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 we 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 complication 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"]
```

This works even in curly brace objects, where it adds all properties from another object. If a property is added multiple times, the last value to be added wins:

```
let coordinates = {x: 10, y: 0};
console.log({...coordinates, y: 5, z: 1});
// → {x: 10, y: 5, z: 1}
```

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 you’ll want to name something `max` in one of your programs. Since JavaScript’s built-in `max` function is tucked safely inside the `Math` object, you 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 you’re not familiar with sines and cosines, don’t worry. I’ll explain them when they are used in [Chapter 14](#).

The previous example used `Math.random`. This is a function that returns a new pseudorandom number between 0 (inclusive) and 1 (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 return 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 reason 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 that 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.

OPTIONAL PROPERTY ACCESS

When you aren't sure whether a given value produces an object, but still want to read a property from it when it does, you can use a variant of the dot notation: `object?.property`.

```
function city(object) {
  return object.address?.city;
}
console.log(city({address: {city: "Toronto"}}));
// → Toronto
console.log(city({name: "Vera"}));
// → undefined
```

The expression `a?.b` means the same as `a.b` when `a` isn't `null` or `undefined`. When it is, it evaluates to `undefined`. This can be convenient when, as in the example, you aren't sure that a given property exists or when a variable might hold an `undefined` value.

A similar notation can be used with square bracket access, and even with function calls, by putting `?.` in front of the parentheses or brackets:

```
console.log("string".notAMethod?.());  
// → undefined  
console.log({}.arrayProp?.[0]);  
// → undefined
```

JSON

Because properties 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. 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 number that represents the address 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 with 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 provide ways to group several values into a single value. 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, with 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 of which they are a property.

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

should 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 this also works with negative step values so that `range(5, 2, -1)` produces `[5, 4, 3, 2]`.

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`, should take an array as its argument and produce a *new* array that has the same elements in the inverse order. The second, `reverseArrayInPlace`, should do what the `reverse` method does: *modify* the array given as its argument by reversing its elements. Neither may use the standard `reverse` method.

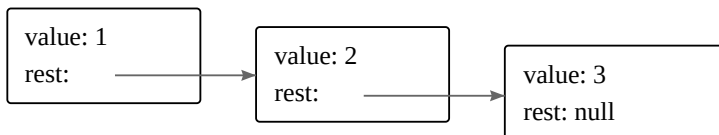
Thinking back to the notes about side effects and pure functions in the previous chapter, which variant do you expect to be useful in more situations? Which one runs faster?

A LIST

As generic blobs of values, objects can be used to build all sorts of data structures. A common data structure is the *list* (not to be confused with arrays). 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, as shown in the following diagram:



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. Add the helper functions `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`.

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`.

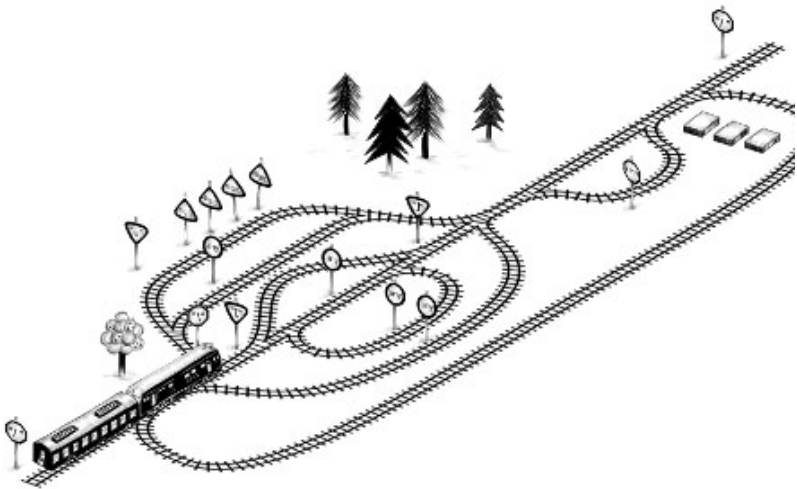
To find out whether values should be compared directly (using the `===` operator for that) or have their properties compared, you can use the `typeof` operator. If it produces `"object"` for both values, you should do a deep comparison. But you have to take one silly exception into account: because of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

REGULAR EXPRESSIONS

“Some people, when confronted with a problem, think ‘I know, I’ll use regular expressions.’ Now they have two problems.”

—Jamie Zawinski



Programming tools and techniques survive and spread in a chaotic, evolutionary way. It’s not always the best or most brilliant ones that win, but rather the ones that function well enough within the right niche or that happen to be integrated with another successful piece of technology.

In this chapter, I will discuss one such tool, *regular expressions*. Regular expressions are a way to describe patterns in string data. They form a small, separate language that is part of JavaScript and many other languages and systems.

Regular expressions are both terribly awkward and extremely useful. Their syntax is cryptic and the programming interface JavaScript provides for them is clumsy. But they are a

powerful tool for inspecting and processing strings. Properly understanding regular expressions will make you a more effective programmer.

CREATING A REGULAR EXPRESSION

A regular expression is a type of object. It can be either constructed with the `RegExp` constructor or written as a literal value by enclosing a pattern in forward slash (`/`) characters.

```
let re1 = new RegExp("abc");
let re2 = /abc/;
```

Both of those regular expression objects represent the same pattern: an *a* character followed by a *b* followed by a *c*.

When using the `RegExp` constructor, the pattern is written as a normal string, so the usual rules apply for backslashes.

The second notation, where the pattern appears between slash characters, treats backslashes somewhat differently. First, since a forward slash ends the pattern, we need to put a backslash before any forward slash that we want to be *part* of the pattern. In addition, backslashes that aren't part of special character codes (like `\n`) will be *preserved*, rather than ignored as they are in strings, and change the meaning of the pattern. Some characters, such as question marks and plus signs, have special meanings in regular expressions and must be preceded by a backslash if they are meant to represent the character itself.

```
let aPlus = /A\+/;
```

TESTING FOR MATCHES

Regular expression objects have a number of methods. The simplest one is `test`. If you pass it a string, it will return a Boolean telling you whether the string contains a match of the pattern in the expression.

```
console.log(/abc/.test("abcde"));
// → true
console.log(/abc/.test("abxde"));
// → false
```

A regular expression consisting of only nonspecial characters simply represents that sequence of characters. If *abc* occurs anywhere in the string we are testing against (not just at the start), `test` will return `true`.

SETS OF CHARACTERS

Finding out whether a string contains *abc* could just as well be done with a call to `indexOf`. Regular expressions are useful because they allow us to describe more complicated patterns.

Say we want to match any number. In a regular expression, putting a set of characters between square brackets makes that part of the expression match any of the characters between the brackets.

Both of the following expressions match all strings that contain a digit:

```
console.log(/[0123456789]/.test("in 1992"));  
// → true  
console.log(/[0-9]/.test("in 1992"));  
// → true
```

Within square brackets, a hyphen (-) between two characters can be used to indicate a range of characters, where the ordering is determined by the character's Unicode number. Characters 0 to 9 sit right next to each other in this ordering (codes 48 to 57), so `[0-9]` covers all of them and matches any digit.

A number of common character groups have their own built-in shortcuts. Digits are one of them: `\d` means the same thing as `[0-9]`.

<code>\d</code>	Any digit character
<code>\w</code>	An alphanumeric character ("word character")
<code>\s</code>	Any whitespace character (space, tab, newline, and similar)
<code>\D</code>	A character that is <i>not</i> a digit
<code>\W</code>	A nonalphanumeric character
<code>\S</code>	A nonwhitespace character
<code>.</code>	Any character except for newline

You could match a date and time format like 01-30-2003 15:20 with the following expression:

```
let dateTime = /\d\d-\d\d-\d\d\d\d \d\d:\d\d/;  
console.log(dateTime.test("01-30-2003 15:20"));  
// → true  
console.log(dateTime.test("30-jan-2003 15:20"));  
// → false
```

That regular expression looks completely awful, doesn't it? Half of it is backslashes, producing a background noise that makes it hard to spot the actual pattern expressed. We'll see a slightly improved version of this expression [later](#).

These backslash codes can also be used inside square brackets. For example, `[\d.]` means any digit or a period character. The period itself, between square brackets, loses its special meaning. The same goes for other special characters, such as the plus sign (+).

To *invert* a set of characters—that is, to express that you want to match any character *except* the ones in the set—you can write a caret (^) character after the opening bracket.

```
let nonBinary = /^[^01]/;
console.log(nonBinary.test("1100100010100110"));
// → false
console.log(nonBinary.test("01111010112101001"));
// → true
```

INTERNATIONAL CHARACTERS

Because of JavaScript's initial simplistic implementation and the fact that this simplistic approach was later set in stone as standard behavior, JavaScript's regular expressions are rather dumb about characters that do not appear in the English language. For example, as far as JavaScript's regular expressions are concerned, a “word character” is only one of the 26 characters in the Latin alphabet (uppercase or lowercase), decimal digits, and, for some reason, the underscore character. Things like *é* or *β*, which most definitely are word characters, will not match `\w` (and *will* match uppercase `\W`, the nonword category).

By a strange historical accident, `\s` (whitespace) does not have this problem and matches all characters that the Unicode standard considers whitespace, including things like the nonbreaking space and the Mongolian vowel separator.

It is possible to use `\p` in a regular expression to match all characters to which the Unicode standard assigns a given property. This allows us to match things like letters in a more cosmopolitan way. However, again due to compatibility with the original language standards, those are recognized only when you put a `u` character (for Unicode) after the regular expression.

<code>\p{L}</code>	Any letter
<code>\p{N}</code>	Any numeric character
<code>\p{P}</code>	Any punctuation character
<code>\P{L}</code>	Any nonletter (uppercase P inverts)
<code>\p{Script=Hangul}</code>	Any character from the given script (see Chapter 5)

Using `\w` for text processing that may need to handle non-English text (or even English text with borrowed words like “*cliché*”) is a liability, since it won’t treat characters like “*é*” as letters. Though they tend to be a bit more verbose, `\p` property groups are more robust.

```
console.log(/\p{L}/u.test("α"));
// → true
console.log(/\p{L}/u.test("!"));
// → false
console.log(/\p{Script=Greek}/u.test("α"));
// → true
console.log(/\p{Script=Arabic}/u.test("α"));
// → false
```

On the other hand, if you are matching numbers in order to do something with them, you often do want `\d` for digits, since converting arbitrary numeric characters into a JavaScript number is not something that a function like `Number` can do for you.

REPEATING PARTS OF A PATTERN

We now know how to match a single digit. What if we want to match a whole number—a sequence of one or more digits?

When you put a plus sign (+) after something in a regular expression, it indicates that the element may be repeated more than once. Thus, `/\d+/` matches one or more digit characters.

```
console.log(/\d+/.test("'123'"));
// → true
console.log(/\d+/.test("' '"));
// → false
console.log(/\d*/.test("'123'"));
// → true
console.log(/\d*/.test("' '"));
// → true
```

The star (*) has a similar meaning but also allows the pattern to match zero times. Something with a star after it never prevents a pattern from matching—it’ll just match zero instances if it can’t find any suitable text to match.

A question mark (?) makes a part of a pattern *optional*, meaning it may occur zero times or one time. In the following example, the *u* character is allowed to occur, but the pattern also matches when it is missing:

```
let neighbor = /neighbou?r/;
console.log(neighbor.test("neighbour"));
// → true
console.log(neighbor.test("neighbor"));
// → true
```

To indicate that a pattern should occur a precise number of times, use braces. Putting {4} after an element, for example, requires it to occur exactly four times. It is also possible to specify a range this way: {2,4} means the element must occur at least twice and at most four times.

Here is another version of the date and time pattern that allows both single- and double-digit days, months, and hours. It is also slightly easier to decipher.

```
let dateTime = /\d{1,2}-\d{1,2}-\d{4} \d{1,2}:\d{2}/;
console.log(dateTime.test("1-30-2003 8:45"));
// → true
```

You can also specify open-ended ranges when using braces by omitting the number after the comma. For example, {5,} means five or more times.

GROUPING SUBEXPRESSIONS

To use an operator like * or + on more than one element at a time, you must use parentheses. A part of a regular expression that is enclosed in parentheses counts as a single element as far as the operators following it are concerned.

```
let cartoonCrying = /boo+(hoo+)/i;
console.log(cartoonCrying.test("Boohooooohooohoo"));
// → true
```

The first and second + characters apply only to the second o in boo and hoo, respectively. The third + applies to the whole group (hoo+), matching one or more sequences like that.

The i at the end of the expression in the example makes this regular expression case insensitive, allowing it to match the uppercase B in the input string, even though the pattern is itself all lowercase.

MATCHES AND GROUPS

The `test` method is the absolute simplest way to match a regular expression. It tells you only whether it matched and nothing else. Regular expressions also have an `exec` (execute) method that will return `null` if no match was found and return an object with information about the match otherwise.

```
let match = /\d+/.exec("one two 100");
console.log(match);
// → ["100"]
console.log(match.index);
// → 8
```

An object returned from `exec` has an `index` property that tells us *where* in the string the successful match begins. Other than that, the object looks like (and in fact is) an array of strings, whose first element is the string that was matched. In the previous example, this is the sequence of digits that we were looking for.

String values have a `match` method that behaves similarly.

```
console.log("one two 100".match(/\d+/));
// → ["100"]
```

When the regular expression contains subexpressions grouped with parentheses, the text that matched those groups will also show up in the array. The whole match is always the first element. The next element is the part matched by the first group (the one whose opening parenthesis comes first in the expression), then the second group, and so on.

```
let quotedText = /'([^']*)*'/;
console.log(quotedText.exec("she said 'hello'"));
// → ["'hello'", "hello"]
```

When a group does not end up being matched at all (for example, when followed by a question mark), its position in the output array will hold `undefined`. When a group is matched multiple times (for example, when followed by a `+`), only the last match ends up in the array.

```
console.log(/bad(ly)?/.exec("bad"));
// → ["bad", undefined]
console.log(/(\d)+/.exec("123"));
// → ["123", "3"]
```

If you want to use parentheses purely for grouping, without having them show up in the array of matches, you can put `?:` after the opening parenthesis.

```
console.log(/(?:na)+/.exec("banana"));  
// → ["nana"]
```

Groups can be useful for extracting parts of a string. If we don't just want to verify whether a string contains a date but also extract it and construct an object that represents it, we can wrap parentheses around the digit patterns and directly pick the date out of the result of `exec`.

But first we'll take a brief detour to discuss the built-in way to represent date and time values in JavaScript.

THE DATE CLASS

JavaScript has a standard `Date` class for representing dates, or rather, points in time. If you simply create a date object using `new`, you get the current date and time.

```
console.log(new Date());  
// → Fri Feb 02 2024 18:03:06 GMT+0100 (CET)
```

You can also create an object for a specific time.

```
console.log(new Date(2009, 11, 9));  
// → Wed Dec 09 2009 00:00:00 GMT+0100 (CET)  
console.log(new Date(2009, 11, 9, 12, 59, 59, 999));  
// → Wed Dec 09 2009 12:59:59 GMT+0100 (CET)
```

JavaScript uses a convention where month numbers start at zero (so December is 11), yet day numbers start at one. This is confusing and silly. Be careful.

The last four arguments (hours, minutes, seconds, and milliseconds) are optional and taken to be zero when not given.

Timestamps are stored as the number of milliseconds since the start of 1970, in the UTC time zone. This follows a convention set by “Unix time”, which was invented around that time. You can use negative numbers for times before 1970. The `getTime` method on a date object returns this number. It is big, as you can imagine.

```

console.log(new Date(2013, 11, 19).getTime());
// → 1387407600000
console.log(new Date(1387407600000));
// → Thu Dec 19 2013 00:00:00 GMT+0100 (CET)

```

If you give the `Date` constructor a single argument, that argument is treated as such a millisecond count. You can get the current millisecond count by creating a new `Date` object and calling `getTime` on it or by calling the `Date.now` function.

`Date` objects provide methods such as `getFullYear`, `getMonth`, `getDate`, `getHours`, `getMinutes`, and `getSeconds` to extract their components. Besides `getFullYear` there's also `getYear`, which gives you the year minus 1900 (such as 98 or 125) and is mostly useless.

Putting parentheses around the parts of the expression that we are interested in, we can now create a date object from a string.

```

function getDate(string) {
  let [, month, day, year] =
    /(\d{1,2})-(\d{1,2})-(\d{4})/.exec(string);
  return new Date(year, month - 1, day);
}
console.log(getDate("1-30-2003"));
// → Thu Jan 30 2003 00:00:00 GMT+0100 (CET)

```

The underscore (`_`) binding is ignored and used only to skip the full match element in the array returned by `exec`.

BOUNDARIES AND LOOK-AHEAD

Unfortunately, `getDate` will also happily extract a date from the string `"100-1-30000"`. A match may happen anywhere in the string, so in this case, it'll just start at the second character and end at the second-to-last character.

If we want to enforce that the match must span the whole string, we can add the markers `^` and `$`. The caret matches the start of the input string, whereas the dollar sign matches the end. Thus `/^d+$` matches a string consisting entirely of one or more digits, `/^!/?` matches any string that starts with an exclamation mark, and `/x^/?` does not match any string (there cannot be an `x` before the start of the string).

There is also a `\b` marker that matches *word boundaries*, positions that have a word character on one side, and a nonword character on the other. Unfortunately, these use the same simplistic concept of word characters as `\w` and are therefore not very reliable.

Note that these boundary markers don't match any actual characters. They just enforce that a given condition holds at the place where it appears in the pattern.

Look-ahead tests do something similar. They provide a pattern and will make the match fail if the input doesn't match that pattern, but don't actually move the match position forward. They are written between `(?=` and `)`.

```
console.log(/a(?=e)/.exec("braeburn"));
// → ["a"]
console.log(/a(?! )/.exec("a b"));
// → null
```

The `e` in the first example is necessary to match, but is not part of the matched string. The `(?! )` notation expresses a *negative* look-ahead. This matches only if the pattern in the parentheses *doesn't* match, causing the second example to match only `a` characters that don't have a space after them.

CHOICE PATTERNS

Say we want to know whether a piece of text contains not only a number but a number followed by one of the words *pig*, *cow*, or *chicken*, or any of their plural forms.

We could write three regular expressions and test them in turn, but there is a nicer way. The pipe character (`|`) denotes a choice between the pattern to its left and the pattern to its right. We can use it in expressions like this:

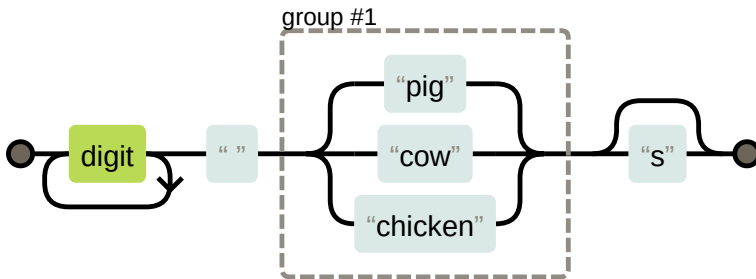
```
let animalCount = /\d+ (pig|cow|chicken)s?/;
console.log(animalCount.test("15 pigs"));
// → true
console.log(animalCount.test("15 pugs"));
// → false
```

Parentheses can be used to limit the part of the pattern to which the pipe operator applies, and you can put multiple such operators next to each other to express a choice between more than two alternatives.

THE MECHANICS OF MATCHING

Conceptually, when you use `exec` or `test`, the regular expression engine looks for a match in your string by trying to match the expression first from the start of the string, then from the second character, and so on until it finds a match or reaches the end of the string. It'll either return the first match that can be found or fail to find any match at all.

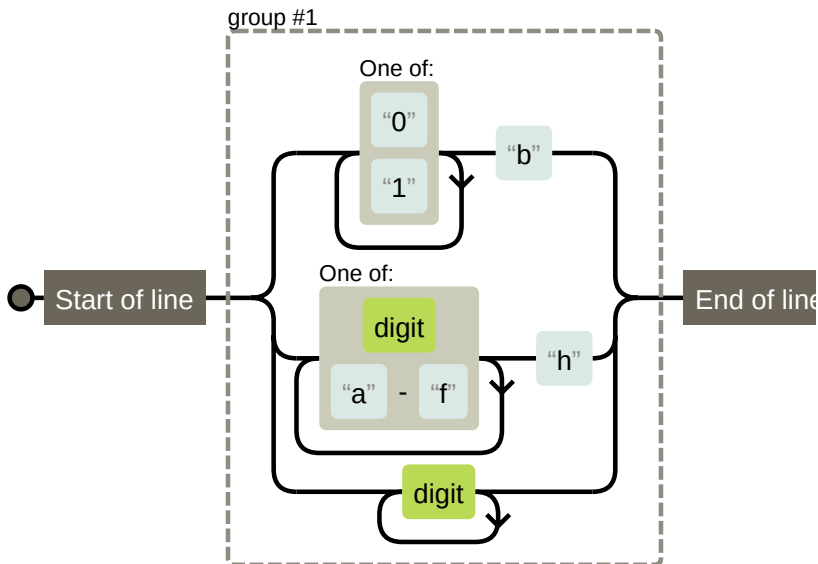
To do the actual matching, the engine treats a regular expression something like a flow diagram. This is the diagram for the livestock expression in the previous example:



If we can find a path from the left side of the diagram to the right side, our expression matches. We keep a current position in the string, and every time we move through a box, we verify that the part of the string after our current position matches that box.

BACKTRACKING

The regular expression `/^[01]+b|[\da-f]+h|\d+)$/` matches either a binary number followed by a `b`, a hexadecimal number (that is, base 16, with the letters `a` to `f` standing for the digits 10 to 15) followed by an `h`, or a regular decimal number with no suffix character. This is the corresponding diagram:



When matching this expression, the top (binary) branch will often be entered even though the input does not actually contain a binary number. When matching the string "103", for example, it becomes clear only at the 3 that we are in the wrong branch. The string *does* match the expression, just not the branch we are currently in.

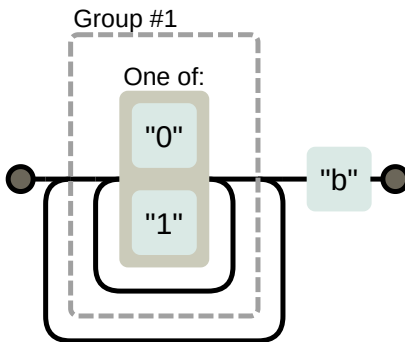
So the matcher *backtracks*. When entering a branch, it remembers its current position (in this case, at the start of the string, just past the first boundary box in the diagram) so that it can go back and try another branch if the current one does not work out. For the string "103", after encountering the 3 character, the matcher starts trying the branch for hexadecimal numbers, which fails again because there is no h after the number. It then tries the decimal number branch. This one fits, and a match is reported after all.

The matcher stops as soon as it finds a full match. This means that if multiple branches could potentially match a string, only the first one (ordered by where the branches appear in the regular expression) is used.

Backtracking also happens for repetition operators like + and *. If you match `/^.*x/` against "abcxe", the `.*` part will first try to consume the whole string. The engine will then realize that it needs an x to match the pattern. Since there is no x past the end of the string, the star operator tries to match one character less. But the matcher doesn't find an x after

`abcx` either, so it backtracks again, matching the star operator to just `abc`. Now it finds an `x` where it needs it and reports a successful match from positions 0 to 4.

It is possible to write regular expressions that will do a *lot* of backtracking. This problem occurs when a pattern can match a piece of input in many different ways. For example, if we get confused while writing a binary-number regular expression, we might accidentally write something like `/([01]+)+b/`.



If that tries to match some long series of zeros and ones with no trailing `b` character, the matcher first goes through the inner loop until it runs out of digits. Then it notices there is no `b`, so it backtracks one position, goes through the outer loop once, and gives up again, trying to backtrack out of the inner loop once more. It will continue to try every possible route through these two loops. This means the amount of work *doubles* with each additional character. For even just a few dozen characters, the resulting match will take practically forever.

THE REPLACE METHOD

String values have a `replace` method that can be used to replace part of the string with another string.

```
console.log("papa".replace("p", "m"));  
// → mapa
```

The first argument can also be a regular expression, in which case the first match of the regular expression is replaced. When a `g` option (for *global*) is added after the regular expression, *all* matches in the string will be replaced, not just the first.

```

console.log("Borobudur".replace(/ou/, "a"));
// → Barobudur
console.log("Borobudur".replace(/ou/g, "a"));
// → Barabadar

```

The real power of using regular expressions with `replace` comes from the fact that we can refer to matched groups in the replacement string. For example, say we have a big string containing the names of people, one name per line, in the format `Lastname, Firstname`. If we want to swap these names and remove the comma to get a `Firstname Lastname` format, we can use the following code:

```

console.log(
  "Liskov, Barbara\nMcCarthy, John\nMilner, Robin"
  .replace(/(\p{L}+), (\p{L}+)/gu, "$2 $1"));
// → Barbara Liskov
//   John McCarthy
//   Robin Milner

```

The `$1` and `$2` in the replacement string refer to the parenthesized groups in the pattern. `$1` is replaced by the text that matched against the first group, `$2` by the second, and so on, up to `$9`. The whole match can be referred to with `&`.

It is possible to pass a function—rather than a string—as the second argument to `replace`. For each replacement, the function will be called with the matched groups (as well as the whole match) as arguments, and its return value will be inserted into the new string.

Here's an example:

```

let stock = "1 lemon, 2 cabbages, and 101 eggs";
function minusOne(match, amount, unit) {
  amount = Number(amount) - 1;
  if (amount == 1) { // only one left, remove the 's'
    unit = unit.slice(0, unit.length - 1);
  } else if (amount == 0) {
    amount = "no";
  }
  return amount + " " + unit;
}
console.log(stock.replace(/(\d+) (\p{L}+)/gu, minusOne));
// → no lemon, 1 cabbage, and 100 eggs

```

This code takes a string, finds all occurrences of a number followed by an alphanumeric word, and returns a string that has one less of every such quantity.

The `(\d+)` group ends up as the `amount` argument to the function, and the `(\p{L}+)` group gets bound to `unit`. The function converts `amount` to a number—which always works, since it matched `\d+` earlier—and makes some adjustments in case there is only one or zero left.

GREED

We can use `replace` to write a function that removes all comments from a piece of JavaScript code. Here is a first attempt:

```
function stripComments(code) {
  return code.replace(/\/\/*.*|\/\/*[^\/*]*\/\/*/g, "");
}
console.log(stripComments("1 + /* 2 */3"));
// → 1 + 3
console.log(stripComments("x = 10; // ten!"));
// → x = 10;
console.log(stripComments("1 /* a */+/* b */ 1"));
// → 1 1
```

The part before the `|` operator matches two slash characters followed by any number of non-newline characters. The part for multiline comments is more involved. We use `[^]` (any character that is not in the empty set of characters) as a way to match any character. We cannot just use a period here because block comments can continue on a new line, and the period character does not match newline characters.

But the output for the last line appears to have gone wrong. Why?

The `[^]*` part of the expression, as I described in the section on backtracking, will first match as much as it can. If that causes the next part of the pattern to fail, the matcher moves back one character and tries again from there. In the example, the matcher first tries to match the whole rest of the string and then moves back from there. It will find an occurrence of `*/` after going back four characters and match that. This is not what we wanted—the intention was to match a single comment, not to go all the way to the end of the code and find the end of the last block comment.

Because of this behavior, we say the repetition operators (`+`, `*`, `?`, and `{}`) are *greedy*, meaning they match as much as they can and backtrack from there. If you put a question mark after them (`+`, `*`, `?`, `{}`), they become nongreedy and start by matching as little as possible, matching more only when the remaining pattern does not fit the smaller match.

And that is exactly what we want in this case. By having the star match the smallest stretch of characters that brings us to a `*/`, we consume one block comment and nothing more.

```
function stripComments(code) {
  return code.replace(/\|\/.*|\/\^[^]*?\/\//g, "");
}
console.log(stripComments("1 /* a */+/* b */ 1"));
// → 1 + 1
```

A lot of bugs in regular expression programs can be traced to unintentionally using a greedy operator where a nongreedy one would work better. When using a repetition operator, prefer the nongreedy variant.

DYNAMICALLY CREATING REGEXP OBJECTS

In some cases you may not know the exact pattern you need to match against when you are writing your code. Say you want to test for the user’s name in a piece of text. You can build up a string and use the `RegExp` constructor on that.

```
let name = "harry";
let regexp = new RegExp("(^|\\s)" + name + "($|\\s)", "gi");
console.log(regexp.test("Harry is a dodgy character."));
// → true
```

When creating the `\\s` part of the string, we have to use two backslashes because we are writing them in a normal string, not a slash-enclosed regular expression. The second argument to the `RegExp` constructor contains the options for the regular expression—in this case, `"gi"` for global and case insensitive.

But what if the name is `"deahl[]rd"` because our user is a nerdy teenager? That would result in a nonsensical regular expression that won’t actually match the user’s name.

To work around this, we can add backslashes before any character that has a special meaning.

```
let name = "deahl[]rd";
let escaped = name.replace(/[\[\].+*?(){}|^$]/g, "\\$&");
let regexp = new RegExp("(^|\\s)" + escaped + "($|\\s)",
  "gi");
let text = "This deahl[]rd guy is super annoying.";
console.log(regexp.test(text));
// → true
```

THE SEARCH METHOD

While the `indexOf` method on strings cannot be called with a regular expression, there is another method, `search`, that does expect a regular expression. Like `indexOf`, it returns the

first index on which the expression was found, or -1 when it wasn't found.

```
console.log("  word".search(/\S/));  
// → 2  
console.log("    ".search(/\S/));  
// → -1
```

Unfortunately, there is no way to indicate that the match should start at a given offset (like we can with the second argument to `indexOf`), which would often be useful.

THE LASTINDEX PROPERTY

The `exec` method similarly does not provide a convenient way to start searching from a given position in the string. But it does provide an *inconvenient* way.

Regular expression objects have properties. One such property is `source`, which contains the string that expression was created from. Another property is `lastIndex`, which controls, in some limited circumstances, where the next match will start.

Those circumstances are that the regular expression must have the global (`g`) or sticky (`y`) option enabled, and the match must happen through the `exec` method. Again, a less confusing solution would have been to just allow an extra argument to be passed to `exec`, but confusion is an essential feature of JavaScript's regular expression interface.

```
let pattern = /y/g;  
pattern.lastIndex = 3;  
let match = pattern.exec("xyzyzzy");  
console.log(match.index);  
// → 4  
console.log(pattern.lastIndex);  
// → 5
```

If the match was successful, the call to `exec` automatically updates the `lastIndex` property to point after the match. If no match was found, `lastIndex` is set back to 0, which is also the value it has in a newly constructed regular expression object.

The difference between the global and the sticky options is that when sticky is enabled, the match will succeed only if it starts directly at `lastIndex`, whereas with global, it will search ahead for a position where a match can start.

```
let global = /abc/g;
console.log(global.exec("xyz abc"));
// → ["abc"]
let sticky = /abc/y;
console.log(sticky.exec("xyz abc"));
// → null
```

When using a shared regular expression value for multiple `exec` calls, these automatic updates to the `lastIndex` property can cause problems. Your regular expression might be accidentally starting at an index left over from a previous call.

```
let digit = /\d/g;
console.log(digit.exec("here it is: 1"));
// → ["1"]
console.log(digit.exec("and now: 1"));
// → null
```

Another interesting effect of the `global` option is that it changes the way the `match` method on strings works. When called with a global expression, instead of returning an array similar to that returned by `exec`, `match` will find *all* matches of the pattern in the string and return an array containing the matched strings.

```
console.log("Banana".match(/an/g));
// → ["an", "an"]
```

So be cautious with global regular expressions. The cases where they are necessary—calls to `replace` and places where you want to explicitly use `lastIndex`—are typically the situations where you want to use them.

A common thing to do is to find all the matches of a regular expression in a string. We can do this by using the `matchAll` method.

```
let input = "A string with 3 numbers in it... 42 and 88.";
let matches = input.matchAll(/\d+/g);
for (let match of matches) {
  console.log("Found", match[0], "at", match.index);
}
// → Found 3 at 14
//   Found 42 at 33
//   Found 88 at 40
```

This method returns an array of match arrays. The regular expression given to `matchAll` *must* have `g` enabled.

PARSING AN INI FILE

To conclude the chapter, we'll look at a problem that calls for regular expressions. Imagine we are writing a program to automatically collect information about our enemies from the internet. (We will not actually write that program here, just the part that reads the configuration file. Sorry.) The configuration file looks like this:

```
searchengine=https://duckduckgo.com/?q=$1
spitefulness=9.7

; comments are preceded by a semicolon...
; each section concerns an individual enemy
[larry]
fullname=Larry Doe
type=kindergarten bully
website=http://www.geocities.com/CapeCanaveral/11451

[davaeorn]
fullname=Davaeorn
type=evil wizard
outputdir=/home/marijn/enemies/davaeorn
```

The exact rules for this format—which is a widely used file format, usually called an *INI* file—are as follows:

- Blank lines and lines starting with semicolons are ignored.
- Lines wrapped in `[` and `]` start a new section.
- Lines containing an alphanumeric identifier followed by an `=` character add a setting to the current section.
- Anything else is invalid.

Our task is to convert a string like this into an object whose properties hold strings for settings written before the first section header and subobjects for sections, with those subobjects holding the section's settings.

Since the format has to be processed line by line, splitting up the file into separate lines is a good start. We saw the `split` method in [Chapter 4](#). Some operating systems, however, use not just a newline character to separate lines but a carriage return character followed by a newline (`"\r\n"`). Given that the `split` method also allows a regular expression as its argument, we can use a regular expression like `/\r?\n/` to split in a way that allows both `"\n"` and `"\r\n"` between lines.

```

function parseINI(string) {
  // Start with an object to hold the top-level fields
  let result = {};
  let section = result;
  for (let line of string.split(/\r?\n/)) {
    let match;
    if (match = line.match(/^(\w+)=(.*)$/)) {
      section[match[1]] = match[2];
    } else if (match = line.match(/^\[([.*])\]$/)) {
      section = result[match[1]] = {};
    } else if (!/^\s*(;|$/).test(line)) {
      throw new Error("Line '" + line + "' is not valid.");
    }
  };
  return result;
}

console.log(parseINI(`
name=Vasilis
[address]
city=Tessaloniki`));
// → {name: "Vasilis", address: {city: "Tessaloniki"}}

```

The code goes over the file's lines and builds up an object. Properties at the top are stored directly into that object, whereas properties found in sections are stored in a separate section object. The `section` binding points at the object for the current section.

There are two kinds of significant lines—section headers or property lines. When a line is a regular property, it is stored in the current section. When it is a section header, a new section object is created, and `section` is set to point at it.

Note the recurring use of `^` and `$` to make sure the expression matches the whole line, not just part of it. Leaving these out results in code that mostly works but behaves strangely for some input, which can be a difficult bug to track down.

The pattern `if (match = string.match(...))` makes use of the fact that the value of an assignment expression (`=`) is the assigned value. You often aren't sure that your call to `match` will succeed, so you can access the resulting object only inside an `if` statement that tests for this. To not break the pleasant chain of `else if` forms, we assign the result of the match to a binding and immediately use that assignment as the test for the `if` statement.

If a line is not a section header or a property, the function checks whether it is a comment or an empty line using the expression `/^\s*(;|$/)` to match lines that either contain only whitespace, or whitespace followed by a semicolon (making the rest of the line a comment). When a line doesn't match any of the expected forms, the function throws an exception.

CODE UNITS AND CHARACTERS

Another design mistake that's been standardized in JavaScript regular expressions is that by default, operators like `.` or `?` work on code units (as discussed in Chapter 5), not actual characters. This means characters that are composed of two code units behave strangely.

```
console.log(/🍎{3}/.test("🍎🍎🍎"));  
// → false  
console.log(/<.>/.test("<🌹>"));  
// → false  
console.log(/<.>/u.test("<🌹>"));  
// → true
```

The problem is that the 🍎 in the first line is treated as two code units, and `{3}` is applied only to the second unit. Similarly, the dot matches a single code unit, not the two that make up the rose emoji.

You must add the `u` (Unicode) option to your regular expression to make it treat such characters properly.

```
console.log(/🍎{3}/u.test("🍎🍎🍎"));  
// → true
```

SUMMARY

Regular expressions are objects that represent patterns in strings. They use their own language to express these patterns.

/abc/	A sequence of characters
/[abc]/	Any character from a set of characters
/[^abc]/	Any character <i>not</i> in a set of characters
/[0-9]/	Any character in a range of characters
/x+/	One or more occurrences of the pattern <code>x</code>
/x+?/	One or more occurrences, nongreedy
/x*/	Zero or more occurrences
/x?/	Zero or one occurrence
/x{2,4}/	Two to four occurrences
/(abc)/	A group
/a b c/	Any one of several patterns
/\d/	Any digit character
/\w/	An alphanumeric character (“word character”)
/\s/	Any whitespace character
/./	Any character except newlines
/\p{L}/u	Any letter character
/^/	Start of input
/\$/	End of input
/(?=a)/	A look-ahead test

A regular expression has a method `test` to test whether a given string matches it. It also has a method `exec` that, when a match is found, returns an array containing all matched groups. Such an array has an `index` property that indicates where the match started.

Strings have a `match` method to match them against a regular expression and a `search` method to search for one, returning only the starting position of the match. Their `replace` method can replace matches of a pattern with a replacement string or function.

Regular expressions can have options, which are written after the closing slash. The `i` option makes the match case insensitive. The `g` option makes the expression *global*, which, among other things, causes the `replace` method to replace all instances instead of just the first. The `y` option makes an expression sticky, which means that it will not search ahead and skip part of the string when looking for a match. The `u` option turns on Unicode mode, which enables `\p` syntax and fixes a number of problems around the handling of characters that take up two code units.

Regular expressions are a sharp tool with an awkward handle. They simplify some tasks tremendously but can quickly become unmanageable when applied to complex problems.

Part of knowing how to use them is resisting the urge to try to shoehorn things into them that they cannot cleanly express.

EXERCISES

It is almost unavoidable that, in the course of working on these exercises, you will get confused and frustrated by some regular expression's inexplicable behavior. Sometimes it helps to enter your expression into an online tool like debuggex.com to see whether its visualization corresponds to what you intended and to experiment with the way it responds to various input strings.

REGEXP GOLF

Code golf is a term used for the game of trying to express a particular program in as few characters as possible. Similarly, *regexp golf* is the practice of writing as tiny a regular expression as possible to match a given pattern and *only* that pattern.

For each of the following items, write a regular expression to test whether the given pattern occurs in a string. The regular expression should match only strings containing the pattern. When your expression works, see whether you can make it any smaller.

1. *car* and *cat*
2. *pop* and *prop*
3. *ferret*, *ferry*, and *ferrari*
4. Any word ending in *ious*
5. A whitespace character followed by a period, comma, colon, or semicolon
6. A word longer than six letters
7. A word without the letter *e* (or *E*)

Refer to the table in the [chapter summary](#) for help. Test each solution with a few test strings.

QUOTING STYLE

Imagine you have written a story and used single quotation marks throughout to mark pieces of dialogue. Now you want to replace all the dialogue quotes with double quotes, while keeping the single quotes used in contractions like *aren't*.

Think of a pattern that distinguishes these two kinds of quote usage and craft a call to the `replace` method that does the proper replacement.

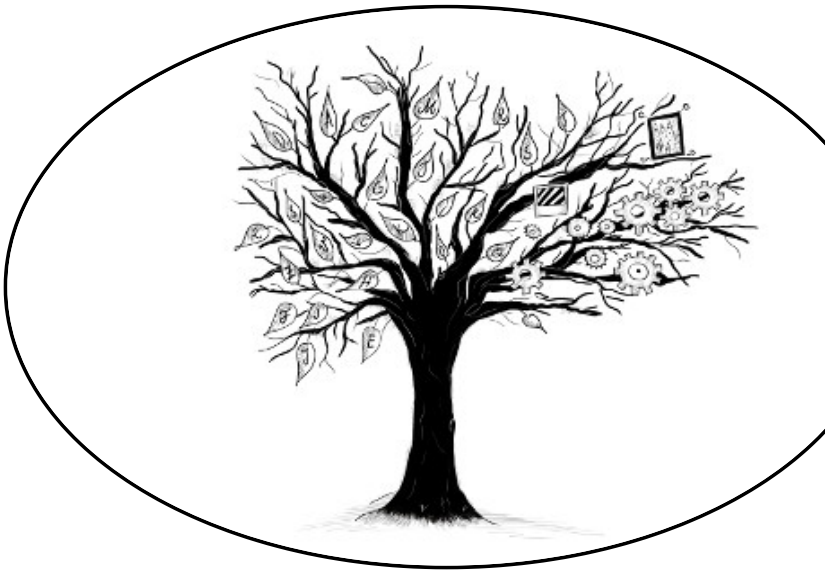
NUMBERS AGAIN

Write an expression that matches only JavaScript-style numbers. It must support an optional minus *or* plus sign in front of the number, the decimal dot, and exponent notation— $5e-3$ or $1E10$ —again with an optional sign in front of the exponent. Also note that it is not necessary for there to be digits in front of or after the dot, but the number cannot be a dot alone. That is, $.5$ and $5.$ are valid JavaScript numbers, but a lone dot isn't.

THE DOCUMENT OBJECT MODEL

“Too bad! Same old story! Once you’ve finished building your house you notice you’ve accidentally learned something that you really should have known—before you started.”

—Friedrich Nietzsche, *Beyond Good and Evil*



When you open a web page, your browser retrieves the page’s HTML text and parses it, much like our parser from Chapter 12 parsed programs. The browser builds up a model of the document’s structure and uses this model to draw the page on the screen.

This representation of the document is one of the toys that a JavaScript program has available in its sandbox. It is a data structure that you can read or modify. It acts as a *live* data structure: when it’s modified, the page on the screen is updated to reflect the changes.

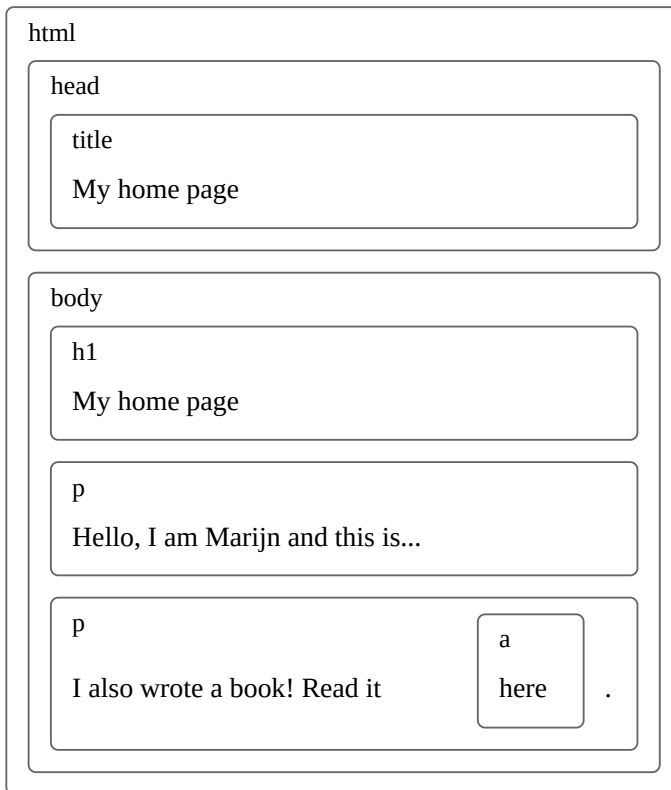
DOCUMENT STRUCTURE

You can imagine an HTML document as a nested set of boxes. Tags such as `<body>` and `</body>` enclose other tags, which in turn contain other tags or text. Here’s the example

document from the previous chapter:

```
<!doctype html>
<html>
  <head>
    <title>My home page</title>
  </head>
  <body>
    <h1>My home page</h1>
    <p>Hello, I am Marijn and this is my home page.</p>
    <p>I also wrote a book! Read it
      <a href="http://eloquentjavascript.net">here</a>.</p>
  </body>
</html>
```

This page has the following structure:



The data structure the browser uses to represent the document follows this shape. For each box, there is an object, which we can interact with to find out things such as what HTML tag it represents and which boxes and text it contains. This representation is called the *Document Object Model*, or *DOM* for short.

The global binding `document` gives us access to these objects. Its `documentElement` property refers to the object representing the `<html>` tag. Since every HTML document has a head and a body, it also has `head` and `body` properties pointing at those elements.

TREES

Think back to the syntax trees from Chapter 12 for a moment. Their structures are strikingly similar to the structure of a browser's document. Each *node* may refer to other nodes, *children*, which in turn may have their own children. This shape is typical of nested structures, where elements can contain subelements that are similar to themselves.

We call a data structure a *tree* when it has a branching structure, no cycles (a node may not contain itself, directly or indirectly), and a single, well-defined *root*. In the case of the DOM, `document.documentElement` serves as the root.

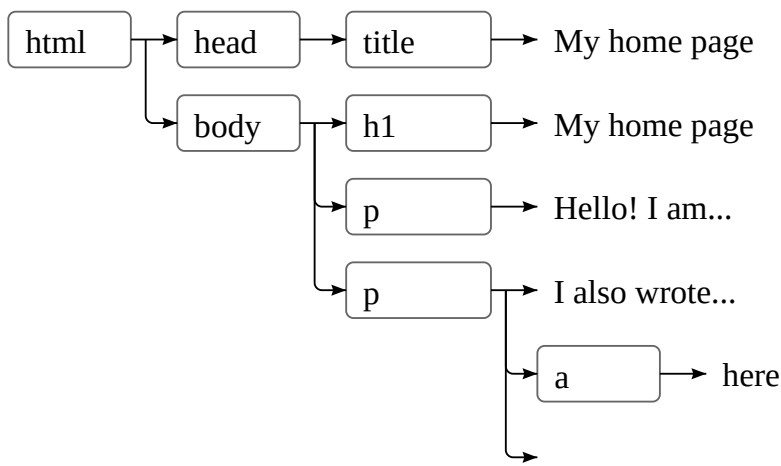
Trees come up a lot in computer science. In addition to representing recursive structures such as HTML documents or programs, they are often used to maintain sorted sets of data because elements can usually be found or inserted more efficiently in a tree than in a flat array.

A typical tree has different kinds of nodes. The syntax tree for the Egg language had identifiers, values, and application nodes. Application nodes may have children, whereas identifiers and values are *leaves*, or nodes without children.

The same goes for the DOM. Nodes for *elements*, which represent HTML tags, determine the structure of the document. These can have child nodes. An example of such a node is `document.body`. Some of these children can be leaf nodes, such as pieces of text or comment nodes.

Each DOM node object has a `nodeType` property, which contains a code (number) that identifies the type of node. Elements have code 1, which is also defined as the constant property `Node.ELEMENT_NODE`. Text nodes, representing a section of text in the document, get code 3 (`Node.TEXT_NODE`). Comments have code 8 (`Node.COMMENT_NODE`).

Another way to visualize our document tree is as follows:



The leaves are text nodes, and the arrows indicate parent-child relationships between nodes.

THE STANDARD

Using cryptic numeric codes to represent node types is not a very JavaScript-like thing to do. Later in this chapter, we'll see that other parts of the DOM interface also feel cumbersome and alien. This is because the DOM interface wasn't designed for JavaScript alone. Rather, it tries to be a language-neutral interface that can be used in other systems as well—not just for HTML but also for XML, which is a generic data format with an HTML-like syntax.

This is unfortunate. Standards are often useful. But in this case, the advantage (cross-language consistency) isn't all that compelling. Having an interface that is properly integrated with the language you're using will save you more time than having a familiar interface across languages.

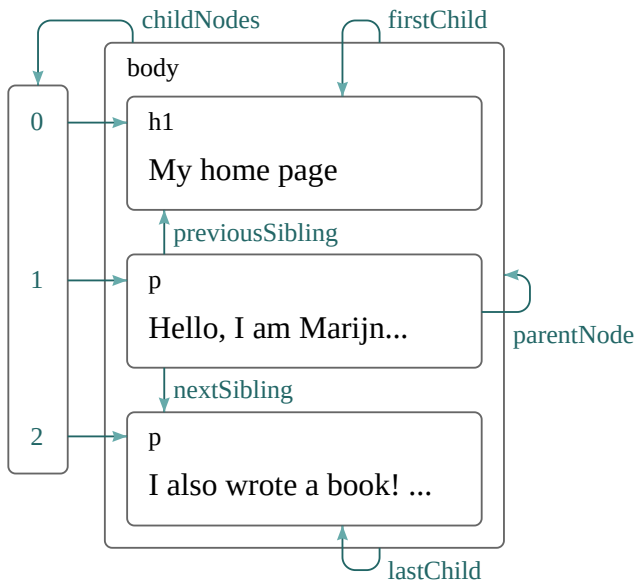
As an example of this poor integration, consider the `childNodes` property that element nodes in the DOM have. This property holds an array-like object with a `length` property and properties labeled by numbers to access the child nodes. But it is an instance of the `NodeList` type, not a real array, so it does not have methods such as `slice` and `map`.

Then there are issues that are simply caused by poor design. For example, there is no way to create a new node and immediately add children or attributes to it. Instead, you have to first create it and then add the children and attributes one by one, using side effects. Code that interacts heavily with the DOM tends to get long, repetitive, and ugly.

But these flaws aren't fatal. Since JavaScript allows us to create our own abstractions, it is possible to design improved ways to express the operations we are performing. Many libraries intended for browser programming come with such tools.

MOVING THROUGH THE TREE

DOM nodes contain a wealth of links to other nearby nodes. The following diagram illustrates these:



Although the diagram shows only one link of each type, every node has a `parentNode` property that points to the node it is part of, if any. Likewise, every element node (node type 1) has a `childNodes` property that points to an array-like object holding its children.

In theory, you could move anywhere in the tree using just these parent and child links. But JavaScript also gives you access to a number of additional convenience links. The `firstChild` and `lastChild` properties point to the first and last child elements or have the value `null` for nodes without children. Similarly, `previousSibling` and `nextSibling` point to adjacent nodes, which are nodes with the same parent that appear immediately before or after the node itself. For a first child, `previousSibling` will be `null`, and for a last child, `nextSibling` will be `null`.

There's also the `children` property, which is like `childNodes` but contains only element (type 1) children, not other types of child nodes. This can be useful when you aren't interested in text nodes.

When dealing with a nested data structure like this one, recursive functions are often useful. The following function scans a document for text nodes containing a given string and returns `true` when it has found one:

```
function talksAbout(node, string) {
  if (node.nodeType == Node.ELEMENT_NODE) {
    for (let child of node.childNodes) {
      if (talksAbout(child, string)) {
        return true;
      }
    }
  }
  return false;
} else if (node.nodeType == Node.TEXT_NODE) {
  return node.nodeValue.indexOf(string) > -1;
}
}

console.log(talksAbout(document.body, "book"));
// → true
```

The `nodeValue` property of a text node holds the string of text that it represents.

FINDING ELEMENTS

Navigating these links among parents, children, and siblings is often useful. But if we want to find a specific node in the document, reaching it by starting at `document.body` and following a fixed path of properties is a bad idea. Doing so bakes assumptions into our program about the precise structure of the document—a structure you might want to change later. Another complicating factor is that text nodes are created even for the whitespace between nodes. The example document's `<body>` tag has not just three children (`<h1>` and two `<p>` elements), but seven: those three, plus the spaces before, after, and between them.

If we want to get the `href` attribute of the link in that document, we don't want to say something like "Get the second child of the sixth child of the document body". It'd be better if we could say "Get the first link in the document". And we can.

```
let link = document.body.getElementsByTagName("a")[0];
console.log(link.href);
```

All element nodes have a `getElementsByTagName` method, which collects all elements with the given tag name that are descendants (direct or indirect children) of that node and returns them as an array-like object.

To find a specific *single* node, you can give it an `id` attribute and use `document.getElementById` instead.

```
<p>My ostrich Gertrude:</p>
<p></p>

<script>
  let ostrich = document.getElementById("gertrude");
  console.log(ostrich.src);
</script>
```

A third, similar method is `getElementsByTagName`, which, like `getElementsByTagName`, searches through the contents of an element node and retrieves all elements that have the given string in their `class` attribute.

CHANGING THE DOCUMENT

Almost everything about the DOM data structure can be changed. The shape of the document tree can be modified by changing parent-child relationships. Nodes have a `remove` method to remove them from their current parent node. To add a child node to an element node, we can use `appendChild`, which puts it at the end of the list of children, or `insertBefore`, which inserts the node given as the first argument before the node given as the second argument.

```
<p>One</p>
<p>Two</p>
<p>Three</p>

<script>
  let paragraphs = document.body.getElementsByTagName("p");
  document.body.insertBefore(paragraphs[2], paragraphs[0]);
</script>
```

A node can exist in the document in only one place. Thus, inserting paragraph *Three* in front of paragraph *One* will first remove it from the end of the document and then insert it at the front, resulting in *Three/One/Two*. All operations that insert a node somewhere will, as a side effect, cause it to be removed from its current position (if it has one).

The `replaceChild` method is used to replace a child node with another one. It takes as arguments two nodes: a new node and the node to be replaced. The replaced node must be a child of the element the method is called on. Note that both `replaceChild` and `insertBefore` expect the *new* node as their first argument.

CREATING NODES

Say we want to write a script that replaces all images (`<img>` tags) in the document with the text held in their `alt` attributes, which specifies an alternative textual representation of the image. This involves not only removing the images but also adding a new text node to replace them.

```
<p>The  in the  
  .</p>  
  
<p><button onclick="replaceImages()">Replace</button></p>  
  
<script>  
  function replaceImages() {  
    let images = document.getElementsByTagName("img");  
    for (let i = images.length - 1; i >= 0; i--) {  
      let image = images[i];  
      if (image.alt) {  
        let text = document.createTextNode(image.alt);  
        image.parentNode.replaceChild(text, image);  
      }  
    }  
  }  
</script>
```

Given a string, `createTextNode` gives us a text node that we can insert into the document to make it show up on the screen.

The loop that goes over the images starts at the end of the list. This is necessary because the node list returned by a method like `getElementsByTagName` (or a property like `childNodes`) is *live*. That is, it is updated as the document changes. If we started from the front, removing the first image would cause the list to lose its first element so that the second time the loop repeats, where `i` is 1, it would stop because the length of the collection is now also 1.

If you want a *solid* collection of nodes, as opposed to a live one, you can convert the collection to a real array by calling `Array.from`.

```
let arrayish = {0: "one", 1: "two", length: 2};
let array = Array.from(arrayish);
console.log(array.map(s => s.toUpperCase()));
// → ["ONE", "TWO"]
```

To create element nodes, you can use the `document.createElement` method. This method takes a tag name and returns a new empty node of the given type.

The following example defines a utility `elt`, which creates an element node and treats the rest of its arguments as children to that node. This function is then used to add an attribution to a quote.

```
<blockquote id="quote">
  No book can ever be finished. While working on it we learn
  just enough to find it immature the moment we turn away
  from it.
</blockquote>

<script>
  function elt(type, ...children) {
    let node = document.createElement(type);
    for (let child of children) {
      if (typeof child !== "string") node.appendChild(child);
      else node.appendChild(document.createTextNode(child));
    }
    return node;
  }

  document.getElementById("quote").appendChild(
    elt("footer", "-",
      elt("strong", "Karl Popper"),
      ", preface to the second edition of ",
      elt("em", "The Open Society and Its Enemies"),
      ", 1950"));
</script>
```

This is what the resulting document looks like:

No book can ever be finished. While working on it we learn
just enough to find it immature the moment we turn away
from it.
—**Karl Popper**, preface to the second edition of *The Open
Society and Its Enemies*, 1950

ATTRIBUTES

Some element attributes, such as `href` for links, can be accessed through a property of the same name on the element's DOM object. This is the case for most commonly used standard attributes.

HTML allows you to set any attribute you want on nodes. This can be useful because it allows you to store extra information in a document. To read or change custom attributes, which aren't available as regular object properties, you have to use the `getAttribute` and `setAttribute` methods.

```
<p data-classified="secret">The launch code is 00000000.</p>
<p data-classified="unclassified">I have two feet.</p>

<script>
  let paras = document.getElementsByTagName("p");
  for (let para of Array.from(paras)) {
    if (para.getAttribute("data-classified") == "secret") {
      para.remove();
    }
  }
</script>
```

It is recommended to prefix the names of such made-up attributes with `data-` to ensure they do not conflict with any other attributes.

There is a commonly used attribute, `class`, which is a keyword in the JavaScript language. For historical reasons—some old JavaScript implementations could not handle property names that matched keywords—the property used to access this attribute is called `className`. You can also access it under its real name, `"class"`, with the `getAttribute` and `setAttribute` methods.

LAYOUT

You may have noticed that different types of elements are laid out differently. Some, such as paragraphs (`<p>`) or headings (`<h1>`), take up the whole width of the document and are rendered on separate lines. These are called *block* elements. Others, such as links (`<a>`) or the `<strong>` element, are rendered on the same line with their surrounding text. Such elements are called *inline* elements.

For any given document, browsers are able to compute a layout, which gives each element a size and position based on its type and content. This layout is then used to actually draw the document.

The size and position of an element can be accessed from JavaScript. The `offsetWidth` and `offsetHeight` properties give you the space the element takes up in *pixels*. A pixel is the basic unit of measurement in the browser. It traditionally corresponds to the smallest dot that the screen can draw, but on modern displays, which can draw very small dots, that may no longer be the case, and a browser pixel may span multiple display dots.

Similarly, `clientWidth` and `clientHeight` give you the size of the space *inside* the element, ignoring border width.

```
<p style="border: 3px solid red">
  I'm boxed in
</p>

<script>
  let para = document.body.getElementsByTagName("p")[0];
  console.log("clientHeight:", para.clientHeight);
  // → 19
  console.log("offsetHeight:", para.offsetHeight);
  // → 25
</script>
```

Giving a paragraph a border causes a rectangle to be drawn around it.



I'm boxed in

The most effective way to find the precise position of an element on the screen is the `getBoundingClientRect` method. It returns an object with `top`, `bottom`, `left`, and `right` properties, indicating the pixel positions of the sides of the element relative to the upper left of the screen. If you want pixel positions relative to the whole document, you must add the current scroll position, which you can find in the `pageXOffset` and `pageYOffset` bindings.

Laying out a document can be quite a lot of work. In the interest of speed, browser engines do not immediately re-layout a document every time you change it but wait as long as they can before doing so. When a JavaScript program that changed the document finishes running, the browser will have to compute a new layout to draw the changed document to the screen. When a program *asks* for the position or size of something by reading properties such as `offsetHeight` or calling `getBoundingClientRect`, providing that information also requires computing a layout.

A program that repeatedly alternates between reading DOM layout information and changing the DOM forces a lot of layout computations to happen and will consequently run very

slowly. The following code is an example of this. It contains two different programs that build up a line of *X* characters 2,000 pixels wide and measures the time each one takes.

```
<p><span id="one"></span></p>
<p><span id="two"></span></p>

<script>
  function time(name, action) {
    let start = Date.now(); // Current time in milliseconds
    action();
    console.log(name, "took", Date.now() - start, "ms");
  }

  time("naive", () => {
    let target = document.getElementById("one");
    while (target.offsetWidth < 2000) {
      target.appendChild(document.createTextNode("X"));
    }
  });
  // → naive took 32 ms

  time("clever", function() {
    let target = document.getElementById("two");
    target.appendChild(document.createTextNode("XXXXX"));
    let total = Math.ceil(2000 / (target.offsetWidth / 5));
    target.firstChild.nodeValue = "X".repeat(total);
  });
  // → clever took 1 ms
</script>
```

STYLING

We have seen that different HTML elements are drawn differently. Some are displayed as blocks, others inline. Some add styling—`<strong>` makes its content bold, and `<a>` makes it blue and underlines it.

The way an `<img>` tag shows an image or an `<a>` tag causes a link to be followed when it is clicked is strongly tied to the element type. But we can change the styling associated with an element, such as the text color or underline. Here is an example that uses the `style` property:

```
<p><a href=".">Normal link</a></p>
<p><a href="." style="color: green">Green link</a></p>
```

The second link will be green instead of the default link color:

Normal link

Green link

A style attribute may contain one or more *declarations*, which are a property (such as `color`) followed by a colon and a value (such as `green`). When there is more than one declaration, they must be separated by semicolons, as in `"color: red; border: none"`.

A lot of aspects of the document can be influenced by styling. For example, the `display` property controls whether an element is displayed as a block or an inline element.

```
This text is displayed <strong>inline</strong>,  
<strong style="display: block">as a block</strong>, and  
<strong style="display: none">not at all</strong>.
```

The `block` tag will end up on its own line, since block elements are not displayed inline with the text around them. The last tag is not displayed at all—`display: none` prevents an element from showing up on the screen. This is a way to hide elements. It is often preferable to removing them from the document entirely because it makes it easy to reveal them again later.

**This text is displayed inline,
as a block
, and .**

JavaScript code can directly manipulate the style of an element through the element's `style` property. This property holds an object that has properties for all possible style properties. The values of these properties are strings, which we can write to in order to change a particular aspect of the element's style.

```

<p id="para" style="color: purple">
  Nice text
</p>

<script>
  let para = document.getElementById("para");
  console.log(para.style.color);
  para.style.color = "magenta";
</script>

```

Some style property names contain hyphens, such as `font-family`. Because such property names are awkward to work with in JavaScript (you'd have to say `style["font-family"]`), the property names in the `style` object for such properties have their hyphens removed and the letters after them capitalized (`style.FontFamily`).

CASCADING STYLES

The styling system for HTML is called *CSS*, for *Cascading Style Sheets*. A *style sheet* is a set of rules for how to style elements in a document. It can be given inside a `<style>` tag.

```

<style>
  strong {
    font-style: italic;
    color: gray;
  }
</style>
<p>Now <strong>strong text</strong> is italic and gray.</p>

```

The *cascading* in the name refers to the fact that multiple such rules are combined to produce the final style for an element. In the example, the default styling for `<strong>` tags, which gives them `font-weight: bold`, is overlaid by the rule in the `<style>` tag, which adds `font-style` and `color`.

When multiple rules define a value for the same property, the most recently read rule gets a higher precedence and wins. For example, if the rule in the `<style>` tag included `font-weight: normal`, contradicting the default `font-weight` rule, the text would be normal, *not* bold. Styles in a `style` attribute applied directly to the node have the highest precedence and always win.

It is possible to target things other than tag names in CSS rules. A rule for `.abc` applies to all elements with `"abc"` in their `class` attribute. A rule for `#xyz` applies to the element with an `id` attribute of `"xyz"` (which should be unique within the document).

```

.subtle {
  color: gray;
  font-size: 80%;
}
#header {
  background: blue;
  color: white;
}
/* p elements with id main and with classes a and b */
p#main.a.b {
  margin-bottom: 20px;
}

```

The precedence rule favoring the most recently defined rule applies only when the rules have the same *specificity*. A rule’s specificity is a measure of how precisely it describes matching elements, determined by the number and kind (tag, class, or ID) of element aspects it requires. For example, a rule that targets `p.a` is more specific than rules that target `p` or just `.a` and would thus take precedence over them.

The notation `p > a {...}` applies the given styles to all `<a>` tags that are direct children of `<p>` tags. Similarly, `p a {...}` applies to all `<a>` tags inside `<p>` tags, whether they are direct or indirect children.

QUERY SELECTORS

We won’t be using style sheets very much in this book. Understanding them is helpful when programming in the browser, but they are complicated enough to warrant a separate book. The main reason I introduced *selector* syntax—the notation used in style sheets to determine which elements a set of styles apply to—is that we can use this same mini-language as an effective way to find DOM elements.

The `querySelectorAll` method, which is defined both on the `document` object and on element nodes, takes a selector string and returns a `NodeList` containing all the elements that it matches.

```

<p>And if you go chasing
  <span class="animal">rabbits</span></p>
<p>And you know you're going to fall</p>
<p>Tell 'em a <span class="character">hookah smoking
  <span class="animal">caterpillar</span></span></p>
<p>Has given you the call</p>

<script>
  function count(selector) {
    return document.querySelectorAll(selector).length;
  }
  console.log(count("p"));           // All <p> elements
  // → 4
  console.log(count(".animal"));     // Class animal
  // → 2
  console.log(count("p .animal"));   // Animal inside of <p>
  // → 2
  console.log(count("p > .animal")); // Direct child of <p>
  // → 1
</script>

```

Unlike methods such as `getElementsByTagName`, the object returned by `querySelectorAll` is *not* live. It won't change when you change the document. It is still not a real array, though, so you need to call `Array.from` if you want to treat it like one.

The `querySelector` method (without the `All` part) works in a similar way. This one is useful if you want a specific single element. It will return only the first matching element, or `null` when no element matches.

POSITIONING AND ANIMATING

The `position` style property influences layout in a powerful way. It has a default value of `static`, meaning the element sits in its normal place in the document. When it is set to `relative`, the element still takes up space in the document, but now the `top` and `left` style properties can be used to move it relative to that normal place. When `position` is set to `absolute`, the element is removed from the normal document flow—that is, it no longer takes up space and may overlap with other elements. Its `top` and `left` properties can be used to absolutely position it relative to the upper-left corner of the nearest enclosing element whose `position` property isn't `static`, or relative to the document if no such enclosing element exists.

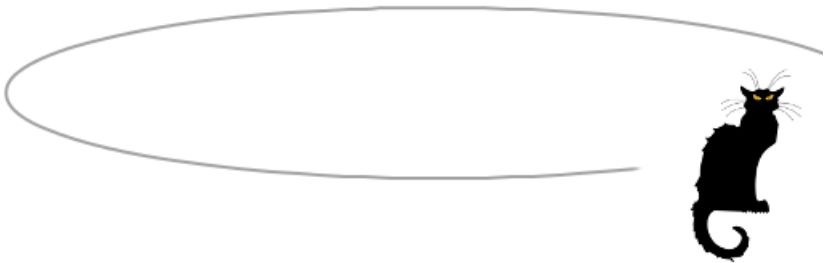
We can use this to create an animation. The following document displays a picture of a cat that moves around in an ellipse:

```

<p style="text-align: center">
  
</p>
<script>
  let cat = document.querySelector("img");
  let angle = Math.PI / 2;
  function animate(time, lastTime) {
    if (lastTime != null) {
      angle += (time - lastTime) * 0.001;
    }
    cat.style.top = (Math.sin(angle) * 20) + "px";
    cat.style.left = (Math.cos(angle) * 200) + "px";
    requestAnimationFrame(newTime => animate(newTime, time));
  }
  requestAnimationFrame(animate);
</script>

```

The gray arrow shows the path along which the image moves.



Our picture is centered on the page and given a position of relative. We'll repeatedly update that picture's top and left styles to move it.

The script uses `requestAnimationFrame` to schedule the `animate` function to run whenever the browser is ready to repaint the screen. The `animate` function itself again calls `requestAnimationFrame` to schedule the next update. When the browser window (or tab) is active, this will cause updates to happen at a rate of about 60 per second, which tends to produce a good-looking animation.

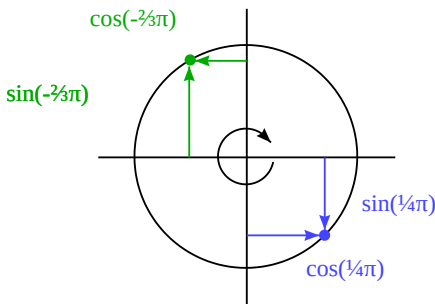
If we just updated the DOM in a loop, the page would freeze, and nothing would show up on the screen. Browsers do not update their display while a JavaScript program is running, nor do they allow any interaction with the page. This is why we need `requestAnimationFrame`—it lets the browser know that we are done for now, and it can go ahead and do the things that browsers do, such as updating the screen and responding to user actions.

The animation function is passed the current time as an argument. To ensure that the motion of the cat per millisecond is stable, it bases the speed at which the angle changes on the difference between the current time and the last time the function ran. If it just moved the angle by a fixed amount per step, the motion would stutter when, for example, another heavy task running on the same computer prevented the function from running for a fraction of a second.

Moving in circles is done using the trigonometry functions `Math.cos` and `Math.sin`. For those who aren't familiar with these, I'll briefly introduce them, since we will occasionally use them in this book.

`Math.cos` and `Math.sin` are useful for finding points that lie on a circle around point (0, 0) with a radius of 1. Both functions interpret their argument as the position on this circle, with 0 denoting the point on the far right of the circle, going clockwise until 2π (about 6.28) has taken us around the whole circle. `Math.cos` tells you the x-coordinate of the point that corresponds to the given position, and `Math.sin` yields the y-coordinate. Positions (or angles) greater than 2π or less than 0 are valid—the rotation repeats so that $a+2\pi$ refers to the same angle as a .

This unit for measuring angles is called radians—a full circle is 2π radians, similar to how it is 360 degrees when measuring in degrees. The constant π is available as `Math.PI` in JavaScript.



The cat animation code keeps a counter, `angle`, for the current angle of the animation and increments it every time the `animate` function is called. It can then use this angle to compute the current position of the image element. The `top` style is computed with `Math.sin` and multiplied by 20, which is the vertical radius of our ellipse. The `left` style is based on `Math.cos` and multiplied by 200 so that the ellipse is much wider than it is high.

Note that styles usually need *units*. In this case, we have to append "px" to the number to tell the browser that we are counting in pixels (as opposed to centimeters, "ems", or other units). This is easy to forget. Using numbers without units will result in your style being ignored—unless the number is 0, which always means the same thing, regardless of its unit.

SUMMARY

JavaScript programs may inspect and interfere with the document that the browser is displaying through a data structure called the DOM. This data structure represents the browser's model of the document, and a JavaScript program can modify it to change the visible document.

The DOM is organized like a tree, where elements are arranged hierarchically according to the structure of the document. The objects representing elements have properties such as `parentNode` and `childNodes`, which can be used to navigate through this tree.

The way a document is displayed can be influenced by *styling*, both by attaching styles to nodes directly and by defining rules that match certain nodes. There are many different style properties, such as `color` or `display`. JavaScript code can manipulate an element's style directly through its `style` property.

EXERCISES

BUILD A TABLE

An HTML table is built with the following tag structure:

```
<table>
  <tr>
    <th>name</th>
    <th>height</th>
    <th>place</th>
  </tr>
  <tr>
    <td>Kilimanjaro</td>
    <td>5895</td>
    <td>Tanzania</td>
  </tr>
</table>
```

For each row, the `<table>` tag contains a `<tr>` tag. Inside of these `<tr>` tags, we can put cell elements: either heading cells (`<th>`) or regular cells (`<td>`).

Given a dataset of mountains, an array of objects with `name`, `height`, and `place` properties, generate the DOM structure for a table that enumerates the objects. It has one column per key and one row per object, plus a header row with `<th>` elements at the top, listing the column names.

Write this so that the columns are automatically derived from the objects, by taking the property names of the first object in the data.

Show the resulting table in the document by appending it to the element that has an `id` attribute of `"mountains"`.

Once you have this working, right-align cells that contain number values by setting their `style.textAlign` property to `"right"`.

ELEMENTS BY TAG NAME

The `document.getElementsByTagName` method returns all child elements with a given tag name. Implement your own version of this as a function that takes a node and a string (the tag name) as arguments and returns an array containing all descendant element nodes with the given tag name. Your function should go through the document itself. It may not use a method like `querySelectorAll` to do the work.

To find the tag name of an element, use its `nodeName` property. But note that this will return the tag name in all uppercase. Use the `toLowerCase` or `toUpperCase` string methods to compensate for this.

THE CAT'S HAT

Extend the cat animation defined [earlier](#) so that both the cat and his hat (``) orbit at opposite sides of the ellipse.

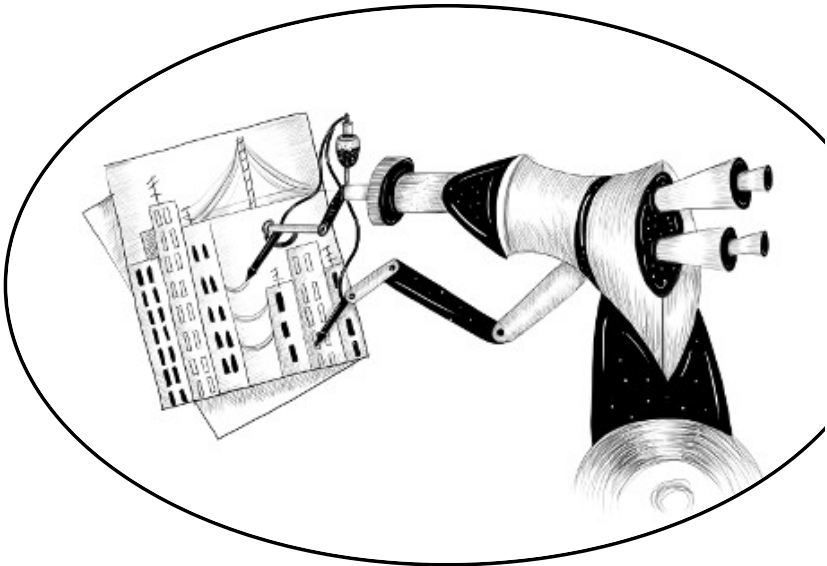
Or make the hat circle around the cat. Or alter the animation in some other interesting way.

To make positioning multiple objects easier, you'll probably want to switch to absolute positioning. This means that `top` and `left` are counted relative to the upper left of the document. To avoid using negative coordinates, which would cause the image to move outside of the visible page, you can add a fixed number of pixels to the position values.

DRAWING ON CANVAS

“Drawing is deception.”

—M.C. Escher, *cited by Bruno Ernst in The Magic Mirror of M.C. Escher*



Browsers give us several ways to display graphics. The simplest way is to use styles to position and color regular DOM elements. This can get us quite far, as the game in the previous chapter showed. By adding partially transparent background images to the nodes, we can make them look exactly the way we want. It is even possible to rotate or skew nodes with the `transform` style.

But we’d be using the DOM for something that it wasn’t originally designed for. Some tasks, such as drawing a line between arbitrary points, are extremely awkward to do with regular HTML elements.

There are two alternatives. The first is DOM based but utilizes *Scalable Vector Graphics* (SVG) rather than HTML. Think of SVG as a document-markup dialect that focuses on

shapes rather than text. You can embed an SVG document directly in an HTML document or include it with an `<img>` tag.

The second alternative is called a *canvas*. A canvas is a single DOM element that encapsulates a picture. It provides a programming interface for drawing shapes onto the space taken up by the node. The main difference between a canvas and an SVG picture is that in SVG the original description of the shapes is preserved so that they can be moved or resized at any time. A canvas, on the other hand, converts the shapes to pixels (colored dots on a raster) as soon as they are drawn and does not remember what these pixels represent. The only way to move a shape on a canvas is to clear the canvas (or the part of the canvas around the shape) and redraw it with the shape in a new position.

SVG

This book won't go into SVG in detail, but I'll briefly explain how it works. At the [end of the chapter](#), I'll come back to the trade-offs that you must consider when deciding which drawing mechanism is appropriate for a given application.

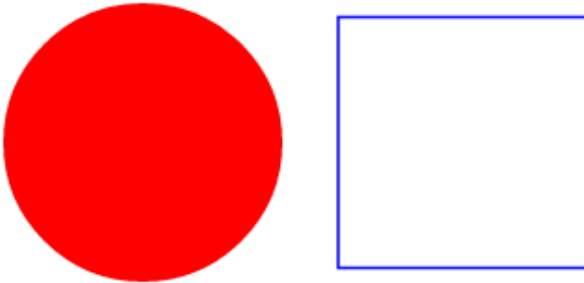
This is an HTML document with a simple SVG picture in it:

```
<p>Normal HTML here.</p>
<svg xmlns="http://www.w3.org/2000/svg">
  <circle r="50" cx="50" cy="50" fill="red"/>
  <rect x="120" y="5" width="90" height="90"
    stroke="blue" fill="none"/>
</svg>
```

The `xmlns` attribute changes an element (and its children) to a different *XML namespace*. This namespace, identified by a URL, specifies the dialect that we are currently speaking. The `<circle>` and `<rect>` tags, which do not exist in HTML, do have a meaning in SVG—they draw shapes using the style and position specified by their attributes.

The document is displayed like this:

Normal HTML here.



These tags create DOM elements, just like HTML tags, that scripts can interact with. For example, this changes the `<circle>` element to be colored cyan instead:

```
let circle = document.querySelector("circle");
circle.setAttribute("fill", "cyan");
```

THE CANVAS ELEMENT

Canvas graphics can be drawn onto a `<canvas>` element. You can give such an element `width` and `height` attributes to determine its size in pixels.

A new canvas is empty, meaning it is entirely transparent and thus shows up as empty space in the document.

The `<canvas>` tag is intended to allow different styles of drawing. To get access to an actual drawing interface, we first need to create a *context*, an object whose methods provide the drawing interface. There are currently three widely supported drawing styles: "2d" for two-dimensional graphics, "webgl" for three-dimensional graphics through the OpenGL interface, and "webgpu", a more modern and flexible alternative to WebGL.

This book won't discuss WebGL or WebGPU—we'll stick to two dimensions. But if you are interested in three-dimensional graphics, I do encourage you to look into WebGPU. It provides a direct interface to graphics hardware and allows you to render even complicated scenes efficiently, using JavaScript.

You create a context with the `getContext` method on the `<canvas>` DOM element.

```
<p>Before canvas.</p>
<canvas width="120" height="60"></canvas>
<p>After canvas.</p>
<script>
  let canvas = document.querySelector("canvas");
  let context = canvas.getContext("2d");
  context.fillStyle = "red";
  context.fillRect(10, 10, 100, 50);
</script>
```

After creating the context object, the example draws a red rectangle that is 100 pixels wide and 50 pixels high, with its upper-left corner at coordinates (10, 10).

Before canvas.



After canvas.

Just like in HTML (and SVG), the coordinate system that the canvas uses puts (0, 0) at the upper-left corner, and the positive y-axis goes down from there. This means (10, 10) is 10 pixels below and to the right of the upper-left corner.

LINES AND SURFACES

In the canvas interface, a shape can be *filled*, meaning its area is given a certain color or pattern, or it can be *stroked*, which means a line is drawn along its edge. SVG uses the same terminology.

The `fillRect` method fills a rectangle. It takes first the x- and y-coordinates of the rectangle's upper-left corner, then its width, and then its height. A similar method called

`strokeRect` draws the outline of a rectangle.

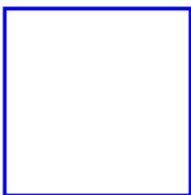
Neither method takes any further parameters. The color of the fill, thickness of the stroke, and so on, are not determined by an argument to the method, as you might reasonably expect, but rather by properties of the context object.

The `fillStyle` property controls the way shapes are filled. It can be set to a string that specifies a color, using the color notation used by CSS.

The `strokeStyle` property works similarly but determines the color used for a stroked line. The width of that line is determined by the `lineWidth` property, which may contain any positive number.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.strokeStyle = "blue";
  cx.strokeRect(5, 5, 50, 50);
  cx.lineWidth = 5;
  cx.strokeRect(135, 5, 50, 50);
</script>
```

This code draws two blue squares, using a thicker line for the second one.



When no `width` or `height` attribute is specified, as in the example, a canvas element gets a default width of 300 pixels and height of 150 pixels.

PATHS

A path is a sequence of lines. The 2D canvas interface takes a peculiar approach to describing such a path. It is done entirely through side effects. Paths are not values that can be stored and passed around. Instead, if you want to do something with a path, you make a sequence of method calls to describe its shape.

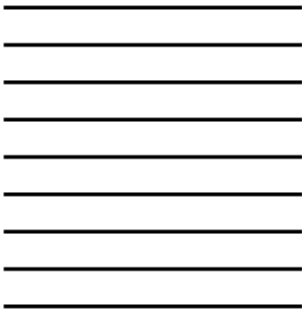
```

<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  for (let y = 10; y < 100; y += 10) {
    cx.moveTo(10, y);
    cx.lineTo(90, y);
  }
  cx.stroke();
</script>

```

This example creates a path with a number of horizontal line segments and then strokes it using the `stroke` method. Each segment created with `lineTo` starts at the path's *current* position. That position is usually the end of the last segment, unless `moveTo` was called. In that case, the next segment would start at the position passed to `moveTo`.

The path described by the previous program looks like this:



When filling a path (using the `fill` method), each shape is filled separately. A path can contain multiple shapes—each `moveTo` motion starts a new one. But the path needs to be *closed* (meaning its start and end are in the same position) before it can be filled. If the path is not already closed, a line is added from its end to its start, and the shape enclosed by the completed path is filled.

```

<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  cx.moveTo(50, 10);
  cx.lineTo(10, 70);
  cx.lineTo(90, 70);
  cx.fill();
</script>

```

This example draws a filled triangle. Note that only two of the triangle's sides are explicitly drawn. The third, from the lower-right corner back to the top, is implied and wouldn't be there if you stroked the path.



You could also use the `closePath` method to explicitly close a path by adding an actual line segment back to the path's start. This segment is drawn when stroking the path.

CURVES

A path may also contain curved lines. These are unfortunately a bit more involved to draw.

The `quadraticCurveTo` method draws a curve to a given point. To determine the curvature of the line, the method is given a control point as well as a destination point. Imagine this control point as *attracting* the line, giving it its curve. The line won't go through the control point, but its direction at the start and end points will be such that a straight line in that direction would point toward the control point. The following example illustrates this:

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  cx.moveTo(10, 90);
  // control=(60, 10) goal=(90, 90)
  cx.quadraticCurveTo(60, 10, 90, 90);
  cx.lineTo(60, 10);
  cx.closePath();
  cx.stroke();
</script>
```

It produces a path that looks like this:

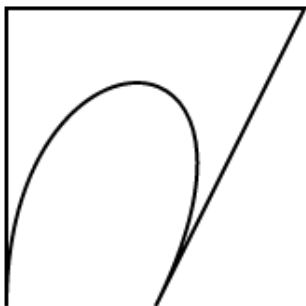


We draw a quadratic curve from the left to the right, with (60, 10) as the control point, and then draw two line segments going through that control point and back to the start of the line. The result somewhat resembles a *Star Trek* insignia. You can see the effect of the control point: the lines leaving the lower corners start off in the direction of the control point and then curve toward their target.

The `bezierCurveTo` method draws a similar kind of curve. Instead of a single control point, this method has two—one for each of the line’s end points. Here is a similar sketch to illustrate the behavior of such a curve:

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  cx.moveTo(10, 90);
  // control1=(10, 10) control2=(90, 10) goal=(50, 90)
  cx.bezierCurveTo(10, 10, 90, 10, 50, 90);
  cx.lineTo(90, 10);
  cx.lineTo(10, 10);
  cx.closePath();
  cx.stroke();
</script>
```

The two control points specify the direction at both ends of the curve. The farther they are away from their corresponding point, the more the curve will “bulge” in that direction.



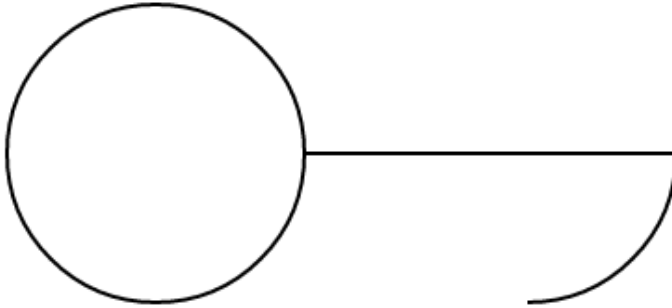
Such curves can be hard to work with—it's not always clear how to find the control points that provide the shape you are looking for. Sometimes you can compute them, and sometimes you'll just have to find a suitable value by trial and error.

The `arc` method is a way to draw a line that curves along the edge of a circle. It takes a pair of coordinates for the arc's center, a radius, and then a start angle and end angle.

Those last two parameters make it possible to draw only part of the circle. The angles are measured in radians, not degrees. This means a full circle has an angle of 2π , or `2 * Math.PI`, which is about 6.28. The angle starts counting at the point to the right of the circle's center and goes clockwise from there. You can use a start of 0 and an end bigger than 2π (say, 7) to draw a full circle.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  // center=(50, 50) radius=40 angle=0 to 7
  cx.arc(50, 50, 40, 0, 7);
  // center=(150, 50) radius=40 angle=0 to ½π
  cx.arc(150, 50, 40, 0, 0.5 * Math.PI);
  cx.stroke();
</script>
```

The resulting picture contains a line from the right of the full circle (first call to `arc`) to the right of the quarter-circle (second call).



Like other path-drawing methods, a line drawn with `arc` is connected to the previous path segment. You can call `moveTo` or start a new path to avoid this.

DRAWING A PIE CHART

Imagine you've just taken a job at EconomiCorp, Inc. Your first assignment is to draw a pie chart of its customer satisfaction survey results.

The `results` binding contains an array of objects that represent the survey responses.

```
const results = [
  {name: "Satisfied", count: 1043, color: "lightblue"},
  {name: "Neutral", count: 563, color: "lightgreen"},
  {name: "Unsatisfied", count: 510, color: "pink"},
  {name: "No comment", count: 175, color: "silver"}
];
```

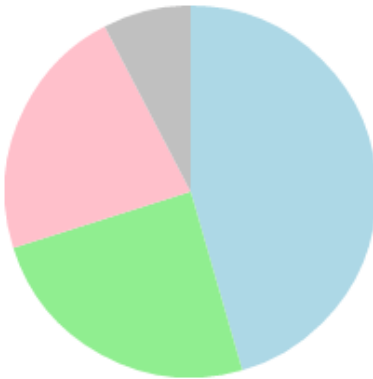
To draw a pie chart, we draw a number of pie slices, each made up of an arc and a pair of lines to the center of that arc. We can compute the angle taken up by each arc by dividing a full circle (2π) by the total number of responses and then multiplying that number (the angle per response) by the number of people who picked a given choice.

```

<canvas width="200" height="200"></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  let total = results
    .reduce((sum, {count}) => sum + count, 0);
  // Start at the top
  let currentAngle = -0.5 * Math.PI;
  for (let result of results) {
    let sliceAngle = (result.count / total) * 2 * Math.PI;
    cx.beginPath();
    // center=100,100, radius=100
    // from current angle, clockwise by slice's angle
    cx.arc(100, 100, 100,
      currentAngle, currentAngle + sliceAngle);
    currentAngle += sliceAngle;
    cx.lineTo(100, 100);
    cx.fillStyle = result.color;
    cx.fill();
  }
</script>

```

This draws the following chart:



But a chart that doesn't tell us what the slices mean isn't very helpful. We need a way to draw text to the canvas.

TEXT

A 2D canvas drawing context provides the methods `fillText` and `strokeText`. The latter can be useful for outlining letters, but usually `fillText` is what you need. It will fill the outline of the given text with the current `fillStyle`.

```

<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.font = "28px Georgia";
  cx.fillStyle = "fuchsia";
  cx.fillText("I can draw text, too!", 10, 50);
</script>

```

You can specify the size, style, and font of the text with the `font` property. This example just gives a font size and family name. It is also possible to add `italic` or `bold` to the start of the string to select a style.

The last two arguments to `fillText` and `strokeText` provide the position at which the font is drawn. By default, they indicate the position of the start of the text’s alphabetic baseline, which is the line that letters “stand” on, not counting hanging parts in letters such as *j* or *p*. You can change the horizontal position by setting the `textAlign` property to `"end"` or `"center"` and the vertical position by setting `textBaseline` to `"top"`, `"middle"`, or `"bottom"`.

We’ll come back to our pie chart, and the problem of labeling the slices, in the [exercises](#) at the end of the chapter.

IMAGES

In computer graphics, a distinction is often made between *vector* graphics and *bitmap* graphics. The first is what we have been doing so far in this chapter—specifying a picture by giving a logical description of shapes. Bitmap graphics, on the other hand, don’t specify actual shapes but rather work with pixel data (rasters of colored dots).

The `drawImage` method allows us to draw pixel data onto a canvas. This pixel data can originate from an `<img>` element or from another canvas. The following example creates a detached `<img>` element and loads an image file into it. But the method cannot immediately start drawing from this picture because the browser may not have loaded it yet. To deal with this, we register a `"load"` event handler and do the drawing after the image has loaded.

```

<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  let img = document.createElement("img");
  img.src = "img/hat.png";
  img.addEventListener("load", () => {
    for (let x = 10; x < 200; x += 30) {
      cx.drawImage(img, x, 10);
    }
  });
</script>

```

By default, `drawImage` will draw the image at its original size. You can also give it two additional arguments to specify the width and height of the drawn image, when those aren't the same as the origin image.

When `drawImage` is given *nine* arguments, it can be used to draw only a fragment of an image. The second through fifth arguments indicate the rectangle (x, y, width, and height) in the source image that should be copied, and the sixth to ninth arguments give the rectangle (on the canvas) into which it should be copied.

This can be used to pack multiple *sprites* (image elements) into a single image file and then draw only the part you need. For example, this picture contains a game character in multiple poses:



By alternating which pose we draw, we can show an animation that looks like a walking character.

To animate a picture on a canvas, the `clearRect` method is useful. It resembles `fillRect`, but instead of coloring the rectangle, it makes it transparent, removing the previously drawn pixels.

We know that each *sprite*, each subpicture, is 24 pixels wide and 30 pixels high. The following code loads the image and then sets up an interval (repeated timer) to draw the next frame:

```

<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  let img = document.createElement("img");
  img.src = "img/player.png";
  let spriteW = 24, spriteH = 30;
  img.addEventListener("load", () => {
    let cycle = 0;
    setInterval(() => {
      cx.clearRect(0, 0, spriteW, spriteH);
      cx.drawImage(img,
                    // source rectangle
                    cycle * spriteW, 0, spriteW, spriteH,
                    // destination rectangle
                    0, 0, spriteW, spriteH);
      cycle = (cycle + 1) % 8;
    }, 120);
  });
</script>

```

The `cycle` binding tracks our position in the animation. For each frame, it is incremented and then clipped back to the 0 to 7 range by using the remainder operator. This binding is then used to compute the x-coordinate that the sprite for the current pose has in the picture.

TRANSFORMATION

What if we want our character to walk to the left instead of to the right? We could draw another set of sprites, of course. But we could also instruct the canvas to draw the picture the other way round.

Calling the `scale` method will cause anything drawn after it to be scaled. This method takes two parameters, one to set a horizontal scale and one to set a vertical scale.

```

<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.scale(3, .5);
  cx.beginPath();
  cx.arc(50, 50, 40, 0, 7);
  cx.lineWidth = 3;
  cx.stroke();
</script>

```

Because of the call to `scale`, the circle is drawn three times as wide and half as high.

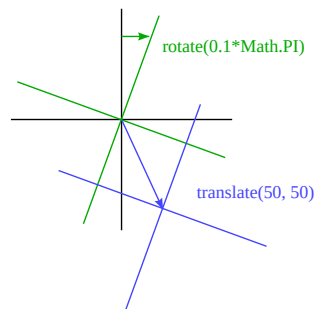
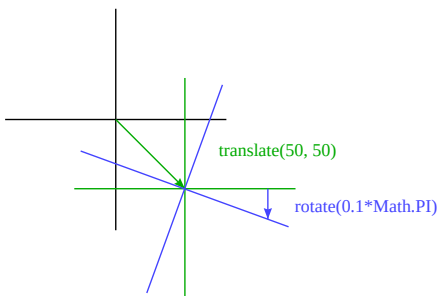


Scaling will cause everything about the drawn image, including the line width, to be stretched out or squeezed together as specified. Scaling by a negative amount will flip the picture around. The flipping happens around point (0, 0), which means it will also flip the direction of the coordinate system. When a horizontal scaling of -1 is applied, a shape drawn at x position 100 will end up at what used to be position -100.

To turn a picture around, we can't simply add `cx.scale(-1, 1)` before the call to `drawImage`. That would move our picture outside of the canvas, where it won't be visible. We could adjust the coordinates given to `drawImage` to compensate for this by drawing the image at x position -50 instead of 0. Another solution, which doesn't require the code doing the drawing to know about the scale change, is to adjust the axis around which the scaling happens.

There are several other methods besides `scale` that influence the coordinate system for a canvas. You can rotate subsequently drawn shapes with the `rotate` method and move them with the `translate` method. The interesting—and confusing—thing is that these transformations *stack*, meaning that each one happens relative to the previous transformations.

If we translate by 10 horizontal pixels twice, everything will be drawn 20 pixels to the right. If we first move the center of the coordinate system to (50, 50) and then rotate by 20 degrees (about 0.1π radians), that rotation will happen *around* point (50, 50).

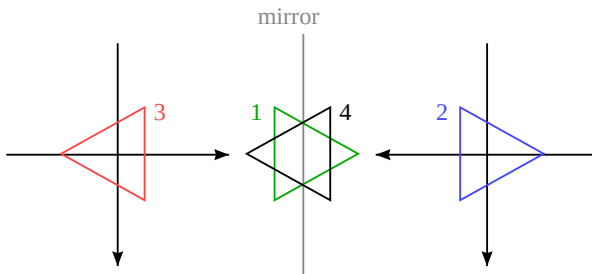


But if we *first* rotate by 20 degrees and *then* translate by (50, 50), the translation will happen in the rotated coordinate system and thus produce a different orientation. The order in which transformations are applied matters.

To flip a picture around the vertical line at a given x position, we can do the following:

```
function flipHorizontally(context, around) {  
    context.translate(around, 0);  
    context.scale(-1, 1);  
    context.translate(-around, 0);  
}
```

We move the y -axis to where we want our mirror to be, apply the mirroring, and finally move the y -axis back to its proper place in the mirrored universe. The following picture explains why this works:



This shows the coordinate systems before and after mirroring across the central line. The triangles are numbered to illustrate each step. If we draw a triangle at a positive x position, it would, by default, be in the place where triangle 1 is. A call to `flipHorizontally` first does a translation to the right, which gets us to triangle 2. It then scales, flipping the triangle over to position 3. This is not where it should be, if it were mirrored in the given line. The second `translate` call fixes this—it “cancels” the initial translation and makes triangle 4 appear exactly where it should.

We can now draw a mirrored character at position (100, 0) by flipping the world around the character’s vertical center.

```

<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  let img = document.createElement("img");
  img.src = "img/player.png";
  let spriteW = 24, spriteH = 30;
  img.addEventListener("load", () => {
    flipHorizontally(cx, 100 + spriteW / 2);
    cx.drawImage(img, 0, 0, spriteW, spriteH,
      100, 0, spriteW, spriteH);
  });
</script>

```

STORING AND CLEARING TRANSFORMATIONS

Transformations stick around. Everything else we draw after drawing that mirrored character would also be mirrored. That might be inconvenient.

It is possible to save the current transformation, do some drawing and transforming, and then restore the old transformation. This is usually the proper thing to do for a function that needs to temporarily transform the coordinate system. First, we save whatever transformation the code that called the function was using. Then the function does its thing, adding more transformations on top of the current transformation. Finally, we revert to the transformation we started with.

The `save` and `restore` methods on the 2D canvas context do this transformation management. They conceptually keep a stack of transformation states. When you call `save`, the current state is pushed onto the stack, and when you call `restore`, the state on top of the stack is taken off and used as the context's current transformation. You can also call `resetTransform` to fully reset the transformation.

The `branch` function in the following example illustrates what you can do with a function that changes the transformation and then calls a function (in this case itself), which continues drawing with the given transformation.

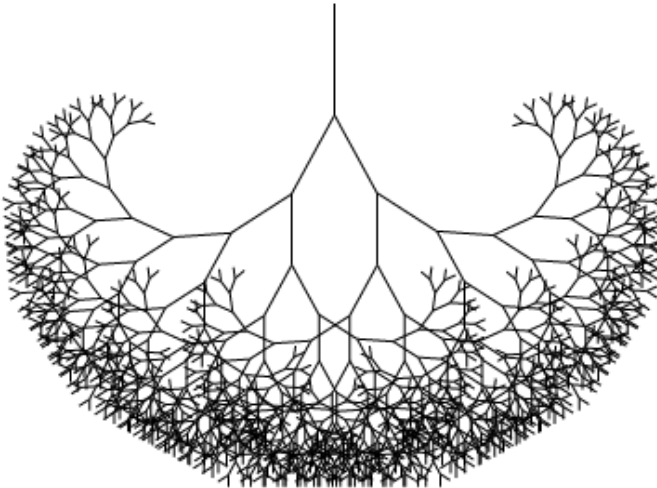
This function draws a treelike shape by drawing a line, moving the center of the coordinate system to the end of the line, and calling itself twice—first rotated to the left and then rotated to the right. Every call reduces the length of the branch drawn, and the recursion stops when the length drops below 8.

```

<canvas width="600" height="300"></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  function branch(length, angle, scale) {
    cx.fillRect(0, 0, 1, length);
    if (length < 8) return;
    cx.save();
    cx.translate(0, length);
    cx.rotate(-angle);
    branch(length * scale, angle, scale);
    cx.rotate(2 * angle);
    branch(length * scale, angle, scale);
    cx.restore();
  }
  cx.translate(300, 0);
  branch(60, 0.5, 0.8);
</script>

```

The result is a simple fractal.



If the calls to `save` and `restore` were not there, the second recursive call to `branch` would end up with the position and rotation created by the first call. It would be connected not to the current branch but rather to the innermost, rightmost branch drawn by the first call. The resulting shape might also be interesting, but it is definitely not a tree.

BACK TO THE GAME

We now know enough about canvas drawing to start working on a canvas-based display system for the game from the previous chapter. The new display will no longer be showing just colored boxes. Instead, we'll use `drawImage` to draw pictures that represent the game's elements.

We define another display object type called `CanvasDisplay`, supporting the same interface as `DOMDisplay` from Chapter 16—namely, the methods `syncState` and `clear`.

This object keeps a little more information than `DOMDisplay`. Rather than using the scroll position of its DOM element, it tracks its own viewport, which tells us which part of the level we are currently looking at. Finally, it keeps a `flipPlayer` property so that even when the player is standing still, it keeps facing the direction in which it last moved.

```
class CanvasDisplay {
  constructor(parent, level) {
    this.canvas = document.createElement("canvas");
    this.canvas.width = Math.min(600, level.width * scale);
    this.canvas.height = Math.min(450, level.height * scale);
    parent.appendChild(this.canvas);
    this.cx = this.canvas.getContext("2d");

    this.flipPlayer = false;

    this.viewport = {
      left: 0,
      top: 0,
      width: this.canvas.width / scale,
      height: this.canvas.height / scale
    };
  }

  clear() {
    this.canvas.remove();
  }
}
```

The `syncState` method first computes a new viewport and then draws the game scene at the appropriate position.

```

CanvasDisplay.prototype.syncState = function(state) {
  this.updateViewport(state);
  this.clearDisplay(state.status);
  this.drawBackground(state.level);
  this.drawActors(state.actors);
};

```

Contrary to `DOMDisplay`, this display style *does* have to redraw the background on every update. Because shapes on a canvas are just pixels, after we draw them there is no good way to move them (or remove them). The only way to update the canvas display is to clear it and redraw the scene. We may also have scrolled, which requires the background to be in a different position.

The `updateViewport` method is similar to `DOMDisplay`'s `scrollPlayerIntoView` method. It checks whether the player is too close to the edge of the screen and moves the viewport when this is the case.

```

CanvasDisplay.prototype.updateViewport = function(state) {
  let view = this.viewport, margin = view.width / 3;
  let player = state.player;
  let center = player.pos.plus(player.size.times(0.5));

  if (center.x < view.left + margin) {
    view.left = Math.max(center.x - margin, 0);
  } else if (center.x > view.left + view.width - margin) {
    view.left = Math.min(center.x + margin - view.width,
      state.level.width - view.width);
  }
  if (center.y < view.top + margin) {
    view.top = Math.max(center.y - margin, 0);
  } else if (center.y > view.top + view.height - margin) {
    view.top = Math.min(center.y + margin - view.height,
      state.level.height - view.height);
  }
};

```

The calls to `Math.max` and `Math.min` ensure that the viewport does not end up showing space outside of the level. `Math.max(x, 0)` makes sure the resulting number is not less than zero. `Math.min` similarly guarantees that a value stays below a given bound.

When clearing the display, we'll use a slightly different color depending on whether the game is won (brighter) or lost (darker).

```

CanvasDisplay.prototype.clearDisplay = function(status) {
  if (status == "won") {
    this.cx.fillStyle = "rgb(68, 191, 255)";
  } else if (status == "lost") {
    this.cx.fillStyle = "rgb(44, 136, 214)";
  } else {
    this.cx.fillStyle = "rgb(52, 166, 251)";
  }
  this.cx.fillRect(0, 0,
    this.canvas.width, this.canvas.height);
};

```

To draw the background, we run through the tiles that are visible in the current viewport, using the same trick used in the `touches` method from the previous chapter.

```

let otherSprites = document.createElement("img");
otherSprites.src = "img/sprites.png";

CanvasDisplay.prototype.drawBackground = function(level) {
  let {left, top, width, height} = this.viewport;
  let xStart = Math.floor(left);
  let xEnd = Math.ceil(left + width);
  let yStart = Math.floor(top);
  let yEnd = Math.ceil(top + height);

  for (let y = yStart; y < yEnd; y++) {
    for (let x = xStart; x < xEnd; x++) {
      let tile = level.rows[y][x];
      if (tile == "empty") continue;
      let screenX = (x - left) * scale;
      let screenY = (y - top) * scale;
      let tileX = tile == "lava" ? scale : 0;
      this.cx.drawImage(otherSprites,
        tileX, 0, scale, scale,
        screenX, screenY, scale, scale);
    }
  }
};

```

Tiles that are not empty are drawn with `drawImage`. The `otherSprites` image contains the pictures used for elements other than the player. It contains, from left to right, the wall tile, the lava tile, and the sprite for a coin.



Background tiles are 20 by 20 pixels, since we'll use the same scale as in `DOMDisplay`. Thus, the offset for lava tiles is 20 (the value of the `scale` binding), and the offset for walls is 0.

We don't bother waiting for the sprite image to load. Calling `drawImage` with an image that hasn't been loaded yet will simply do nothing. Thus, we might fail to draw the game properly for the first few frames while the image is still loading, but that isn't a serious problem. Since we keep updating the screen, the correct scene will appear as soon as the loading finishes.

The walking character shown earlier will be used to represent the player. The code that draws it needs to pick the right sprite and direction based on the player's current motion. The first eight sprites contain a walking animation. When the player is moving along a floor, we cycle through them based on the current time. We want to switch frames every 60 milliseconds, so the time is divided by 60 first. When the player is standing still, we draw the ninth sprite. During jumps, which are recognized by the fact that the vertical speed is not zero, we use the tenth, rightmost sprite.

Because the sprites are slightly wider than the player object—24 instead of 16 pixels to allow some space for feet and arms—the method has to adjust the x-coordinate and width by a given amount (`playerXOverlap`).

```

let playerSprites = document.createElement("img");
playerSprites.src = "img/player.png";
const playerXOverlap = 4;

CanvasDisplay.prototype.drawPlayer = function(player, x, y,
                                              width, height){
    width += playerXOverlap * 2;
    x -= playerXOverlap;
    if (player.speed.x != 0) {
        this.flipPlayer = player.speed.x < 0;
    }

    let tile = 8;
    if (player.speed.y != 0) {
        tile = 9;
    } else if (player.speed.x != 0) {
        tile = Math.floor(Date.now() / 60) % 8;
    }

    this.cx.save();
    if (this.flipPlayer) {
        flipHorizontally(this.cx, x + width / 2);
    }
    let tileX = tile * width;
    this.cx.drawImage(playerSprites, tileX, 0, width, height,
                      x, y, width, height);
    this.cx.restore();
};

```

The `drawPlayer` method is called by `drawActors`, which is responsible for drawing all the actors in the game.

```

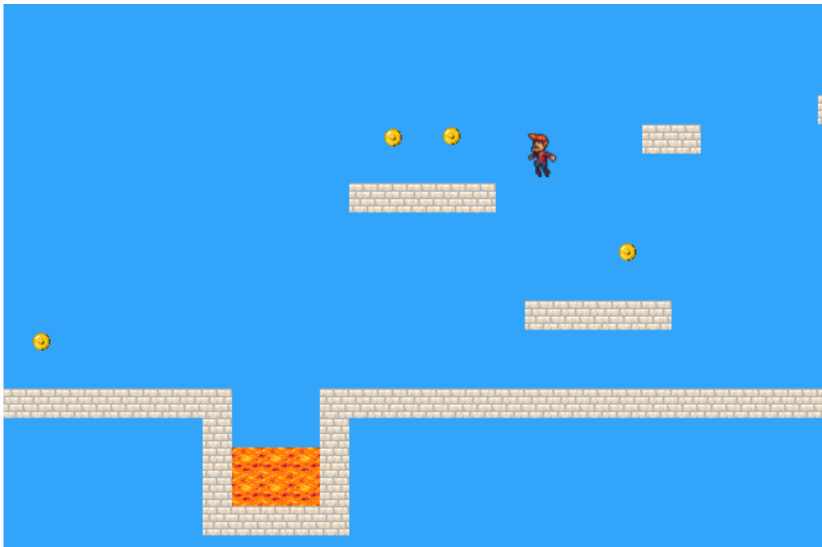
CanvasDisplay.prototype.drawActors = function(actors) {
    for (let actor of actors) {
        let width = actor.size.x * scale;
        let height = actor.size.y * scale;
        let x = (actor.pos.x - this.viewport.left) * scale;
        let y = (actor.pos.y - this.viewport.top) * scale;
        if (actor.type == "player") {
            this.drawPlayer(actor, x, y, width, height);
        } else {
            let tileX = (actor.type == "coin" ? 2 : 1) * scale;
            this.cx.drawImage(otherSprites,
                              tileX, 0, width, height,
                              x, y, width, height);
        }
    }
};

```

When drawing something that is not the player, we look at its type to find the offset of the correct sprite. The lava tile is found at offset 20, and the coin sprite is found at 40 (two times `scale`).

We have to subtract the viewport's position when computing the actor's position, since (0, 0) on our canvas corresponds to the top left of the viewport, not the top left of the level. We could also have used `translate` for this. Either way works.

That concludes the new display system. The resulting game looks something like this:



CHOOSING A GRAPHICS INTERFACE

When you need to generate graphics in the browser, you can choose between plain HTML, SVG, and canvas. There is no single *best* approach that works in all situations. Each option has strengths and weaknesses.

Plain HTML has the advantage of being simple. It also integrates well with text. Both SVG and canvas allow you to draw text, but they won't help you position that text or wrap it when it takes up more than one line. In an HTML-based picture, it is much easier to include blocks of text.

SVG can be used to produce crisp graphics that look good at any zoom level. Unlike HTML, it is designed for drawing and is thus more suitable for that purpose.

Both SVG and HTML build up a data structure (the DOM) that represents your picture. This makes it possible to modify elements after they are drawn. If you need to repeatedly change a small part of a big picture in response to what the user is doing or as part of an animation, doing it in a canvas can be needlessly expensive. The DOM also allows us to register mouse event handlers on every element in the picture (even on shapes drawn with SVG). You can't do that with canvas.

But canvas's pixel-oriented approach can be an advantage when drawing a huge number of tiny elements. The fact that it does not build up a data structure but only repeatedly draws onto the same pixel surface gives canvas a lower cost per shape. There are also effects that are only practical with a canvas element, such as rendering a scene one pixel at a time (for example, using a ray tracer) or postprocessing an image with JavaScript (blurring or distorting it).

In some cases, you may want to combine several of these techniques. For example, you might draw a graph with SVG or canvas but show textual information by positioning an HTML element on top of the picture.

For nondemanding applications, it really doesn't matter much which interface you choose. The display we built for our game in this chapter could have been implemented using any of these three graphics technologies, since it does not need to draw text, handle mouse interaction, or work with an extraordinarily large number of elements.

SUMMARY

In this chapter we discussed techniques for drawing graphics in the browser, focusing on the `<canvas>` element.

A canvas node represents an area in a document that our program may draw on. This drawing is done through a drawing context object, created with the `getContext` method.

The 2D drawing interface allows us to fill and stroke various shapes. The context's `fillStyle` property determines how shapes are filled. The `strokeStyle` and `lineWidth` properties control the way lines are drawn.

Rectangles and pieces of text can be drawn with a single method call. The `fillRect` and `strokeRect` methods draw rectangles, and the `fillText` and `strokeText` methods draw text. To create custom shapes, we must first build up a path.

Calling `beginPath` starts a new path. A number of other methods add lines and curves to the current path. For example, `lineTo` can add a straight line. When a path is finished, it can be

filled with the `fill` method or stroked with the `stroke` method.

Moving pixels from an image or another canvas onto our canvas is done with the `drawImage` method. By default, this method draws the whole source image, but by giving it more parameters, you can copy a specific area of the image. We used this for our game by copying individual poses of the game character out of an image that contained many such poses.

Transformations allow you to draw a shape in multiple orientations. A 2D drawing context has a current transformation that can be changed with the `translate`, `scale`, and `rotate` methods. These will affect all subsequent drawing operations. A transformation state can be saved with the `save` method and restored with the `restore` method.

When showing an animation on a canvas, the `clearRect` method can be used to clear part of the canvas before redrawing it.

EXERCISES

SHAPES

Write a program that draws the following shapes on a canvas:

1. A trapezoid (a rectangle that is wider on one side)
2. A red diamond (a rectangle rotated 45 degrees or $\frac{1}{4}\pi$ radians)
3. A zigzagging line
4. A spiral made up of 100 straight line segments
5. A yellow star



When drawing the last two shapes, you may want to refer to the explanation of `Math.cos` and `Math.sin` in [Chapter 14](#), which describes how to get coordinates on a circle using these functions.

I recommend creating a function for each shape. Pass the position, and optionally other properties such as the size or the number of points, as parameters. The alternative, which is

to hardcode numbers all over your code, tends to make the code needlessly hard to read and modify.

THE PIE CHART

[Earlier](#) in the chapter, we saw an example program that drew a pie chart. Modify this program so that the name of each category is shown next to the slice that represents it. Try to find a pleasing-looking way to automatically position this text that would work for other datasets as well. You may assume that categories are big enough to leave enough room for their labels.

You might need `Math.sin` and `Math.cos` again, which are described in [Chapter 14](#).

A BOUNCING BALL

Use the `requestAnimationFrame` technique that we saw in [Chapter 14](#) and Chapter 16 to draw a box with a bouncing ball in it. The ball moves at a constant speed and bounces off the box's sides when it hits them.

PRECOMPUTED MIRRORING

One unfortunate thing about transformations is that they slow down the drawing of bitmaps. The position and size of each pixel have to be transformed, and though it is possible that browsers will get cleverer about transformation in the future, they currently cause a measurable increase in the time it takes to draw a bitmap.

In a game like ours, where we are drawing only a single transformed sprite, this is a nonissue. But imagine that we need to draw hundreds of characters or thousands of rotating particles from an explosion.

Think of a way to draw an inverted character without loading additional image files and without having to make transformed `drawImage` calls every frame.