Lesson 5

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

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

The chapter will work through a more or less realistic programming example, introducing concepts as they apply to the problem at hand. The example code will often build on functions and bindings that were introduced earlier in the text.

The weresquirrel

Every now and then, usually between eight and ten in the evening, Jacques finds himself transforming into a small furry rodent with a bushy tail.

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

That takes care of the cat and tree problems. But Jacques would prefer to get rid of his condition entirely. The irregular occurrences of the transformation make him suspect that they might be triggered by

something. For a while, he believed that it happened only on days when he had been near oak trees. But avoiding oak trees did not stop the problem.

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

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

Data sets

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

We could get creative with strings—after all, strings can have any length, so we can put a lot of data into them—and use "2 3 5 7 11" as our representation. But this is awkward. You'd have to somehow extract the digits and convert them back to numbers to access them.

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

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

The notation for getting at the elements inside an array also uses square brackets. A pair of square brackets immediately after an expression, with another expression inside of them, will look up the element in the

left-hand expression that corresponds to the *index* given by the expression in the brackets.

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

Properties

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

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

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

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

So if you know that the property you are interested in is called *color*, you say value.color. If you want to extract the property named by the value

held in the binding i, you say value[i]. Property names are strings. They can be any string, but the dot notation only works with names that look like valid binding names. So if you want to access a property named *2* or *John Doe*, you must use square brackets: value[2] or value["John Doe"]. The elements in an array are stored as the array's properties, using numbers as property names. Because you can't use the dot notation with numbers, and usually want to use a binding that holds the index anyway, you have to use the bracket notation to get at them.

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

Methods

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

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

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

Interestingly, even though the call to toUpperCase does not pass any arguments, the function somehow has access to the string "Doh", the value whose property we called. How this works is described in Chapter 6. Properties that contain functions are generally called *methods* of the value they belong to. As in, "toUpperCase is a method of a string".

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

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

The push method adds values to the end of an array, and the pop method does the opposite, removing the last value in the array and returning it. These somewhat silly names are the traditional terms for operations on a *stack*. A stack, in programming, is a data structure that allows you to push values into it and pop them out again in the opposite order, so that the thing that was added last is removed first. These are common in programming—you might remember the function call stack from arrays, 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 curly braces as an expression.

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

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

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

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

Reading a property that doesn't exist will give you the value undefined. It is possible to assign a value to a property expression with the = operator. This will replace the property's value if it already existed or create a new property on the object if it didn't.

To briefly return to our tentacle model of bindings—property bindings are similar. They *grasp* values, but other bindings and properties might be holding onto those same values. You may think of objects as octopuses with any number of tentacles, each of which has a name tattooed on it. The delete operator cuts off a tentacle from such an octopus. It is a unary operator that, when applied to an object property, will remove the named property from the object. This is not a common thing to do, but it is possible.

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

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

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

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

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

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

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

We will represent Jacques' journal as an array of objects.

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

Mutability

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

We saw that object values can be modified. The types of values discussed in earlier chapters, such as numbers, strings, and Booleans, are all *immutable*—it is impossible to change values of those types. You can

combine them and derive new values from them, but when you take a specific string value, that value will always remain the same. The text inside it cannot be changed. If you have a string that contains "cat", it is not possible for other code to change a character in your string to make it spell "rat".

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

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

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

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

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

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

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

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

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

The lycanthrope's log

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

```
let journal = [];

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

Note that the object added to the journal looks a little odd. Instead of declaring properties like events: events, it just gives a property name. This is a short-hand that means the same thing—if a property name in

curly brace notation isn't followed by a value, its value is taken from the binding with the same name.

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

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

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

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

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

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

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

$$\phi = \frac{n_{11}n_{00} - n_{10}n_{01}}{\sqrt{n_{1\bullet}n_{0\bullet}n_{\bullet1}n_{\bullet0}}}$$

(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—just this one formula for now. And even with this one, all we do is turn it into JavaScript.)

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

The value $n_{1\bullet}$ refers to the sum of all measurements where the first variable is true, which is 5 in the example table. Likewise, $n_{\bullet0}$ 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 $\sqrt{340000}$. This comes out to $\phi \approx 0.069$, which is tiny. Eating pizza does not appear to have influence on the transformations.

Computing correlation

We can represent a two-by-two table in JavaScript with a four-element array (`[76, 9, 4, 1]`). We could also use other representations, such as an array containing two two-element arrays (`[[76, 9], [4, 1]]`) or an object with property names like `"11"` and `"01"`, but the flat array is simple and makes the expressions that access the table pleasantly short. We'll

interpret the indices to the array as two-bit binary numbers, where the leftmost (most significant) digit refers to the squirrel variable and the rightmost (least significant) digit refers to the event variable. For example, the binary number 10 refers to the case where Jacques did turn into a squirrel, but the event (say, "pizza") didn't occur. This happened four times. And since binary 10 is 2 in decimal notation, we will store this number at index 2 of the array.

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

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

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

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

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

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

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

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

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

The body of the loop in tableFor figures out which box in the table each journal entry falls into by checking whether the entry contains the specific event it's interested in and whether the event happens alongside a squirrel incident. The loop then adds one to the correct box in the table. We now have the tools we need to compute individual correlations. The only step remaining is to find a correlation for every type of event that was recorded and see whether anything stands out.

Array loops

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

```
for (let i = 0; i < JOURNAL.length; i++) {
  let entry = JOURNAL[i];
  // Do something with entry
}
```

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

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

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

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

The final analysis

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

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

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

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

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

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

```javascript
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
```

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

Interesting. Let's try something.

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

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

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

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

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

Further arrayology

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

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

```javascript
let todoList = [];
function remember(task) {
  todoList.push(task);
}
function getTask() {
  return todoList.shift();
}
function rememberUrgently(task) {
  todoList.unshift(task);
}
```

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

To search for a specific value, arrays provide an indexOf method. It 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.

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

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

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

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

When the end index is not given, slice will take all of the elements after the start index. You can also omit the start index to copy the entire array. The concat method can be used to glue arrays together to create a new array, similar to what the + operator does for strings.

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

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

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

Strings and their properties

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

```
let kim = "Kim";
kim.age = 88;
```

```
console.log(kim.age);
// → undefined
```

Values of type string, number, and Boolean are not objects, and though the language doesn't complain if you try to set new properties on them, it doesn't actually store those properties. As mentioned before, 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 together again with join.

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

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

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

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

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

Rest parameters

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

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

```
function max(...numbers) {
  let result = -Infinity;
  for (let number of numbers) {
    if (number > result) result = number;
  }
```

```
  return result;
}
console.log(max(4, 1, 9, -2));
// → 9
```

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

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

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

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

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

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

The Math object

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

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

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

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

Back to the Math object. If you need to do trigonometry, Math can help. It contains cos (cosine), sin (sine), and tan (tangent), as well as their inverse functions, acos, asin, and atan, respectively. The number π (pi)—or at least the closest approximation that fits in a JavaScript number—is available as Math.PI. There is an old programming tradition of writing the names of constant values in all caps.

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

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

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

```javascript
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 zero and below 10. Since Math.floor rounds down, this expression will produce, with equal chance, any number from 0 through 9.

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

Destructuring

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

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

```
      (table[0] + table[1]) *
      (table[1] + table[3]) *
      (table[0] + table[2]));
}
```

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

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

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

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

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

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

JSON

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

(among other things) a binary number that represents the position of the inner array.

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

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

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

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

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

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

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

```
console.log(JSON.parse(string).events);
// → ["weekend"]
```