function only to show what it does internally. From now on, we'll use it like this instead:

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

Transforming with map

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

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

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

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

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

Summarizing with reduce

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

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

sum.

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

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

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

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

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

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

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

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

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

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

Composability

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

```
let biggest = null;
for (let script of SCRIPTS) {
  if (biggest == null ||
      characterCount(biggest) < characterCount(script)) {
    biggest = script;
  }
}
console.log(biggest);
// → {name: "Han", …}
```

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

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

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

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

So the dead scripts in Unicode are, on average, older than the living ones.

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

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

```
let total = 0, count = 0;
for (let script of SCRIPTS) {
  if (script.living) {
    total += script.year;
    count += 1;
  }
}
console.log(Math.round(total / count));
// → 1165
```

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

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

Strings and character codes

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

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

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

```
  return null;
}

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

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

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

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

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

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

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

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

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

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

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

Recognizing text

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

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

console.log(countBy([1, 2, 3, 4, 5], n => n > 2));
```

```
// → [{name: false, count: 2}, {name: true, count: 3}]
```

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

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

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

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

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

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

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

The function first counts the characters by name, using `characterScript` to assign them a name and falling back to the string `"none"` for characters that aren't part of any script. The `filter` call drops the entry for `"none"` from the resulting array since we aren't interested in those characters.

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

Summary

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

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

Exercises

Flattening

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

Your own loop

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

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

Everything

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

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

Dominant writing direction

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

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

"An abstract data type is realized by writing a special kind of program […] which defines the type in terms of the operations which can be performed on it."

—Barbara Liskov, Programming with Abstract Data Types

CHAPTER 6

The Secret Life of Objects

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

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

Encapsulation

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

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

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

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

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

Even though the language doesn't have this distinction built in, JavaScript programmers *are* successfully using this idea. Typically, the available interface is described in documentation or comments. It is also common to put an

underscore (_) character at the start of property names to indicate that those properties are private.

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

Methods

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

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

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

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

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

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

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

```
speak.call(hungryRabbit, "Burp!");
```

```
// → The hungry rabbit says 'Burp!'
```

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

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

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

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

PROTOTYPES

Watch closely.

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

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

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

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

```
console.log(Object.getPrototypeOf({}) ==
            Object.prototype);
```

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

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

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

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

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

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

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

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

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

The "proto" rabbit acts as a container for the properties that are shared by all rabbits. An individual rabbit object, like the killer rabbit, contains properties that apply only to itself—in this case its type—and derives shared properties

from its prototype.

Classes

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

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

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

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

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

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

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

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

Constructors (all functions, in fact) automatically get a property named

`prototype`, which by default holds a plain, empty object that derives from `Object.prototype`. You can overwrite it with a new object if you want. Or you can add properties to the existing object, as the example does.

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

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

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

Class notation

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

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

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

The `class` keyword starts a class declaration, which allows us to define a constructor and a set of methods all in a single place. Any number of methods may be written inside the declaration's braces. The one named `constructor`

is treated specially. It provides the actual constructor function, which will be bound to the name `Rabbit`. The others are packaged into that constructor's prototype. Thus, the earlier class declaration is equivalent to the constructor definition from the previous section. It just looks nicer.

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

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

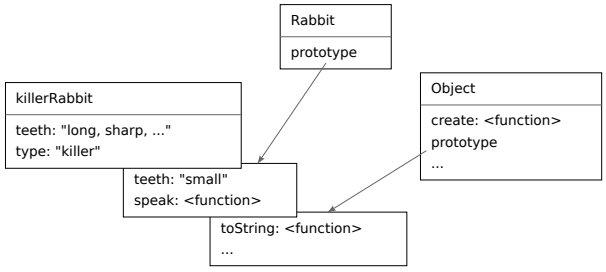
```
let object = new class { getWord() { return "hello"; } };
console.log(object.getWord());
// → hello
```

Overriding derived properties

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

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

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



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

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

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

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

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

Maps

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

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

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

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

Here, the object's property names are the people's names, and the property values are their ages. But we certainly didn't list anybody named toString in our map. Yet, because plain objects derive from `Object.prototype`, it looks like the property is there.

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

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

Object property names must be strings. If you need a map whose keys can't easily be converted to strings—such as objects—you cannot use an object as your map.

Fortunately, JavaScript comes with a class called `Map` that is written for this exact purpose. It stores a mapping and allows any type of keys.

```
let ages = new Map();
ages.set("Boris", 39);
ages.set("Liang", 22);
ages.set("Júlia", 62);

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

The methods `set`, `get`, and `has` are part of the interface of the `Map` object. Writing a data structure that can quickly update and search a large set of values isn't easy, but we don't have to worry about that. Someone else did it for us, and we can go through this simple interface to use their work.

If you do have a plain object that you need to treat as a map for some reason, it is useful to know that `Object.keys` returns only an object's *own* keys, not those in the prototype. As an alternative to the `in` operator, you can use the `hasOwnProperty` method, which ignores the object's prototype.

```
console.log({x: 1}.hasOwnProperty("x"));
// → true
console.log({x: 1}.hasOwnProperty("toString"));
// → false
```

Polymorphism

When you call the `String` function (which converts a value to a string) on an object, it will call the `toString` method on that object to try to create a meaningful string from it. I mentioned that some of the standard prototypes define their own version of `toString` so they can create a string that contains more useful information than `"[object Object]"`. You can also do that yourself.

```
Rabbit.prototype.toString = function() {
  return `a ${this.type} rabbit`;
};

console.log(String(blackRabbit));
// → a black rabbit
```

This is a simple instance of a powerful idea. When a piece of code is written to work with objects that have a certain interface—in this case, a `toString` method—any kind of object that happens to support this interface can be plugged into the code, and it will just work.

This technique is called *polymorphism*. Polymorphic code can work with values of different shapes, as long as they support the interface it expects.

I mentioned in Chapter 4 that a `for/of` loop can loop over several kinds of data structures. This is another case of polymorphism—such loops expect the data structure to expose a specific interface, which arrays and strings do. And

we can also add this interface to your own objects! But before we can do that, we need to know what symbols are.

Symbols

It is possible for multiple interfaces to use the same property name for different things. For example, I could define an interface in which the `toString` method is supposed to convert the object into a piece of yarn. It would not be possible for an object to conform to both that interface and the standard use of `toString`.

That would be a bad idea, and this problem isn't that common. Most JavaScript programmers simply don't think about it. But the language designers, whose *job* it is to think about this stuff, have provided us with a solution anyway.

When I claimed that property names are strings, that wasn't entirely accurate. They usually are, but they can also be *symbols*. Symbols are values created with the `Symbol` function. Unlike strings, newly created symbols are unique—you cannot create the same symbol twice.

```
let sym = Symbol("name");
console.log(sym == Symbol("name"));
// → false
Rabbit.prototype[sym] = 55;
console.log(blackRabbit[sym]);
// → 55
```

The string you pass to `Symbol` is included when you convert it to a string and can make it easier to recognize a symbol when, for example, showing it in the console. But it has no meaning beyond that—multiple symbols may have the same name.

Being both unique and usable as property names makes symbols suitable for defining interfaces that can peacefully live alongside other properties, no matter what their names are.

```
const toStringSymbol = Symbol("toString");
Array.prototype[toStringSymbol] = function() {
  return `${this.length} cm of blue yarn`;
};

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

```
// → 2 cm of blue yarn
```

It is possible to include symbol properties in object expressions and classes by using square brackets around the property name. That causes the property name to be evaluated, much like the square bracket property access notation, which allows us to refer to a binding that holds the symbol.

```
let stringObject = {
  [toStringSymbol]() { return "a jute rope"; }
};
console.log(stringObject[toStringSymbol]());
// → a jute rope
```

The iterator interface

The object given to a `for`/`of` loop is expected to be *iterable*. This means it has a method named with the `Symbol.iterator` symbol (a symbol value defined by the language, stored as a property of the `Symbol` function).

When called, that method should return an object that provides a second interface, *iterator*. This is the actual thing that iterates. It has a `next` method that returns the next result. That result should be an object with a `value` property that provides the next value, if there is one, and a `done` property, which should be true when there are no more results and false otherwise.

Note that the `next`, `value`, and `done` property names are plain strings, not symbols. Only `Symbol.iterator`, which is likely to be added to a *lot* of different objects, is an actual symbol.

We can directly use this interface ourselves.

```
let okIterator = "OK"[Symbol.iterator]();
console.log(okIterator.next());
// → {value: "O", done: false}
console.log(okIterator.next());
// → {value: "K", done: false}
console.log(okIterator.next());
// → {value: undefined, done: true}
```

Let's implement an iterable data structure. We'll build a *matrix* class, acting as a two-dimensional array.

```
class Matrix {
  constructor(width, height, element = (x, y) => undefined) {
    this.width = width;
    this.height = height;
    this.content = [];

    for (let y = 0; y < height; y++) {
      for (let x = 0; x < width; x++) {
        this.content[y * width + x] = element(x, y);
      }
    }
  }

  get(x, y) {
    return this.content[y * this.width + x];
  }
  set(x, y, value) {
    this.content[y * this.width + x] = value;
  }
}
```

The class stores its content in a single array of *width* × *height* elements. The elements are stored row by row, so, for example, the third element in the fifth row is (using zero-based indexing) stored at position 4 × *width* + 2.

The constructor function takes a width, a height, and an optional `element` function that will be used to fill in the initial values. There are `get` and `set` methods to retrieve and update elements in the matrix.

When looping over a matrix, you are usually interested in the position of the elements as well as the elements themselves, so we'll have our iterator produce objects with `x`, `y`, and `value` properties.

```
class MatrixIterator {
  constructor(matrix) {
    this.x = 0;
    this.y = 0;
    this.matrix = matrix;
  }

  next() {
    if (this.y == this.matrix.height) return {done: true};

    let value = {x: this.x,
                 y: this.y,
                 value: this.matrix.get(this.x, this.y)};
```

```
      this.x++;
      if (this.x == this.matrix.width) {
        this.x = 0;
        this.y++;
      }
      return {value, done: false};
    }
  }
```

The class tracks the progress of iterating over a matrix in its `x` and `y` properties. The `next` method starts by checking whether the bottom of the matrix has been reached. If it hasn't, it *first* creates the object holding the current value and *then* updates its position, moving to the next row if necessary.

Let's set up the `Matrix` class to be iterable. Throughout this book, I'll occasionally use after-the-fact prototype manipulation to add methods to classes so that the individual pieces of code remain small and self-contained. In a regular program, where there is no need to split the code into small pieces, you'd declare these methods directly in the class instead.

```
Matrix.prototype[Symbol.iterator] = function() {
  return new MatrixIterator(this);
};
```

We can now loop over a matrix with `for`/`of`.

```
let matrix = new Matrix(2, 2, (x, y) => `value ${x},${y}`);
for (let {x, y, value} of matrix) {
  console.log(x, y, value);
}
// → 0 0 value 0,0
// → 1 0 value 1,0
// → 0 1 value 0,1
// → 1 1 value 1,1
```

Getters, setters, and statics

Interfaces often consist mostly of methods, but it is also okay to include properties that hold non-function values. For example, `Map` objects have a `size` property that tells you how many keys are stored in them.

It is not even necessary for such an object to compute and store such a

property directly in the instance. Even properties that are accessed directly may hide a method call. Such methods are called *getters*, and they are defined by writing `get` in front of the method name in an object expression or class declaration.

```
let varyingSize = {
  get size() {
    return Math.floor(Math.random() * 100);
  }
};

console.log(varyingSize.size);
// → 73
console.log(varyingSize.size);
// → 49
```

Whenever someone reads from this object's `size` property, the associated method is called. You can do a similar thing when a property is written to, using a *setter*.

```
class Temperature {
  constructor(celsius) {
    this.celsius = celsius;
  }
  get fahrenheit() {
    return this.celsius * 1.8 + 32;
  }
  set fahrenheit(value) {
    this.celsius = (value - 32) / 1.8;
  }

  static fromFahrenheit(value) {
    return new Temperature((value - 32) / 1.8);
  }
}

let temp = new Temperature(22);
console.log(temp.fahrenheit);
// → 71.6
temp.fahrenheit = 86;
console.log(temp.celsius);
// → 30
```

The `Temperature` class allows you to read and write the temperature in either degrees Celsius or degrees Fahrenheit, but internally it stores only Celsius and automatically converts to and from Celsius in the `fahrenheit` getter and setter.

Sometimes you want to attach some properties directly to your constructor function, rather than to the prototype. Such methods won't have access to a class instance but can, for example, be used to provide additional ways to create instances.

Inside a class declaration, methods that have `static` written before their name are stored on the constructor. So the `Temperature` class allows you to write `Temperature.fromFahrenheit(100)` to create a temperature using degrees Fahrenheit.

Inheritance

Some matrices are known to be *symmetric*. If you mirror a symmetric matrix around its top-left-to-bottom-right diagonal, it stays the same. In other words, the value stored at x,y is always the same as that at y,x.

Imagine we need a data structure like `Matrix` but one that enforces the fact that the matrix is and remains symmetrical. We could write it from scratch, but that would involve repeating some code very similar to what we already wrote.

JavaScript's prototype system makes it possible to create a *new* class, much like the old class, but with new definitions for some of its properties. The prototype for the new class derives from the old prototype but adds a new definition for, say, the `set` method.

In object-oriented programming terms, this is called *inheritance*. The new class inherits properties and behavior from the old class.

```
class SymmetricMatrix extends Matrix {
  constructor(size, element = (x, y) => undefined) {
    super(size, size, (x, y) => {
      if (x < y) return element(y, x);
      else return element(x, y);
    });
  }

  set(x, y, value) {
    super.set(x, y, value);
    if (x != y) {
      super.set(y, x, value);
    }
  }
```

```
}

let matrix = new SymmetricMatrix(5, (x, y) => `${x},${y}`);
console.log(matrix.get(2, 3));
// → 3,2
```

The use of the word `extends` indicates that this class shouldn't be directly based on the default `Object` prototype but on some other class. This is called the *superclass*. The derived class is the *subclass*.

To initialize a `SymmetricMatrix` instance, the constructor calls its superclass's constructor through the `super` keyword. This is necessary because if this new object is to behave (roughly) like a `Matrix`, it is going to need the instance properties that matrices have. To ensure the matrix is symmetrical, the constructor wraps the `element` function to swap the coordinates for values below the diagonal.

The `set` method again uses `super` but this time not to call the constructor but to call a specific method from the superclass's set of methods. We are redefining `set` but do want to use the original behavior. Because `this.set` refers to the *new* `set` method, calling that wouldn't work. Inside class methods, `super` provides a way to call methods as they were defined in the superclass.

Inheritance allows us to build slightly different data types from existing data types with relatively little work. It is a fundamental part of the object-oriented tradition, alongside encapsulation and polymorphism. But while the latter two are now generally regarded as wonderful ideas, inheritance is more controversial.

Whereas encapsulation and polymorphism can be used to *separate* pieces of code from each other, reducing the tangledness of the overall program, inheritance fundamentally ties classes together, creating *more* tangle. When inheriting from a class, you usually have to know more about how it works than when simply using it. Inheritance can be a useful tool, and I use it now and then in my own programs, but it shouldn't be the first tool you reach for, and you probably shouldn't actively go looking for opportunities to construct class hierarchies (family trees of classes).

The instanceof operator

It is occasionally useful to know whether an object was derived from a specific class. For this, JavaScript provides a binary operator called `instanceof`.

```
console.log(
  new SymmetricMatrix(2) instanceof SymmetricMatrix);
```

```
// → true
console.log(new SymmetricMatrix(2) instanceof Matrix);
// → true
console.log(new Matrix(2, 2) instanceof SymmetricMatrix);
// → false
console.log([1] instanceof Array);
// → true
```

The operator will see through inherited types, so a `SymmetricMatrix` is an instance of `Matrix`. The operator can also be applied to standard constructors like `Array`. Almost every object is an instance of `Object`.

Summary

So objects do more than just hold their own properties. They have prototypes, which are other objects. They'll act as if they have properties they don't have as long as their prototype has that property. Simple objects have `Object.prototype` as their prototype.

Constructors, which are functions whose names usually start with a capital letter, can be used with the `new` operator to create new objects. The new object's prototype will be the object found in the `prototype` property of the constructor. You can make good use of this by putting the properties that all values of a given type share into their prototype. There's a `class` notation that provides a clear way to define a constructor and its prototype.

You can define getters and setters to secretly call methods every time an object's property is accessed. Static methods are methods stored in a class's constructor, rather than its prototype.

The `instanceof` operator can, given an object and a constructor, tell you whether that object is an instance of that constructor.

One useful thing to do with objects is to specify an interface for them and tell everybody that they are supposed to talk to your object only through that interface. The rest of the details that make up your object are now *encapsulated*, hidden behind the interface.

More than one type may implement the same interface. Code written to use an interface automatically knows how to work with any number of different objects that provide the interface. This is called *polymorphism*.

When implementing multiple classes that differ in only some details, it can be helpful to write the new classes as *subclasses* of an existing class, *inheriting* part of its behavior.

EXERCISES

A VECTOR TYPE

Write a class `Vec` that represents a vector in two-dimensional space. It takes `x` and `y` parameters (numbers), which it should save to properties of the same name.

Give the `Vec` prototype two methods, `plus` and `minus`, that take another vector as a parameter and return a new vector that has the sum or difference of the two vectors' (`this` and the parameter) x and y values.

Add a getter property `length` to the prototype that computes the length of the vector—that is, the distance of the point (x, y) from the origin (0, 0).

GROUPS

The standard JavaScript environment provides another data structure called `Set`. Like an instance of `Map`, a set holds a collection of values. Unlike `Map`, it does not associate other values with those—it just tracks which values are part of the set. A value can be part of a set only once—adding it again doesn't have any effect.

Write a class called `Group` (since `Set` is already taken). Like `Set`, it has `add`, `delete`, and `has` methods. Its constructor creates an empty group, `add` adds a value to the group (but only if it isn't already a member), `delete` removes its argument from the group (if it was a member), and `has` returns a Boolean value indicating whether its argument is a member of the group.

Use the `===` operator, or something equivalent such as `indexOf`, to determine whether two values are the same.

Give the class a static `from` method that takes an iterable object as argument and creates a group that contains all the values produced by iterating over it.

ITERABLE GROUPS

Make the `Group` class from the previous exercise iterable. Refer to the section about the iterator interface earlier in the chapter if you aren't clear on the exact form of the interface anymore.

If you used an array to represent the group's members, don't just return the iterator created by calling the `Symbol.iterator` method on the array. That would work, but it defeats the purpose of this exercise.

It is okay if your iterator behaves strangely when the group is modified during iteration.

Borrowing a method

Earlier in the chapter I mentioned that an object's `hasOwnProperty` can be used as a more robust alternative to the `in` operator when you want to ignore the prototype's properties. But what if your map needs to include the word `"hasOwnProperty"`? You won't be able to call that method anymore because the object's own property hides the method value.

Can you think of a way to call `hasOwnProperty` on an object that has its own property by that name?

"[...] the question of whether Machines Can Think [...] is about as relevant as the question of whether Submarines Can Swim."

—Edsger Dijkstra, The Threats to Computing Science

CHAPTER 7

Project: A Robot

In "project" chapters, I'll stop pummeling you with new theory for a brief moment, and instead we'll work through a program together. Theory is necessary to learn to program, but reading and understanding actual programs is just as important.

Our project in this chapter is to build an automaton, a little program that performs a task in a virtual world. Our automaton will be a mail-delivery robot picking up and dropping off parcels.

Meadowfield

The village of Meadowfield isn't very big. It consists of 11 places with 14 roads between them. It can be described with this array of roads:

```
const roads = [
  "Alice's House-Bob's House",   "Alice's House-Cabin",
  "Alice's House-Post Office",   "Bob's House-Town Hall",
  "Daria's House-Ernie's House", "Daria's House-Town Hall",
  "Ernie's House-Grete's House", "Grete's House-Farm",
  "Grete's House-Shop",          "Marketplace-Farm",
  "Marketplace-Post Office",     "Marketplace-Shop",
  "Marketplace-Town Hall",       "Shop-Town Hall"
];
```

The network of roads in the village forms a *graph*. A graph is a collection of points (places in the village) with lines between them (roads). This graph will be the world that our robot moves through.

The array of strings isn't very easy to work with. What we're interested in is the destinations that we can reach from a given place. Let's convert the list of roads to a data structure that, for each place, tells us what can be reached from there.

```
function buildGraph(edges) {
  let graph = Object.create(null);
  function addEdge(from, to) {
    if (graph[from] == null) {
      graph[from] = [to];
    } else {
      graph[from].push(to);
    }
  }
  for (let [from, to] of edges.map(r => r.split("-"))) {
    addEdge(from, to);
    addEdge(to, from);
  }
  return graph;
}

const roadGraph = buildGraph(roads);
```

Given an array of edges, `buildGraph` creates a map object that, for each node,

stores an array of connected nodes.

It uses the `split` method to go from the road strings, which have the form `"Start-End"`, to two-element arrays containing the start and end as separate strings.

The task

Our robot will be moving around the village. There are parcels in various places, each addressed to some other place. The robot picks up parcels when it comes to them and delivers them when it arrives at their destinations.

The automaton must decide, at each point, where to go next. It has finished its task when all parcels have been delivered.

To be able to simulate this process, we must define a virtual world that can describe it. This model tells us where the robot is and where the parcels are. When the robot has decided to move somewhere, we need to update the model to reflect the new situation.

If you're thinking in terms of object-oriented programming, your first impulse might be to start defining objects for the various elements in the world: a class for the robot, one for a parcel, maybe one for places. These could then hold properties that describe their current state, such as the pile of parcels at a location, which we could change when updating the world.

This is wrong.

At least, it usually is. The fact that something sounds like an object does not automatically mean that it should be an object in your program. Reflexively writing classes for every concept in your application tends to leave you with a collection of interconnected objects that each have their own internal, changing state. Such programs are often hard to understand and thus easy to break.

Instead, let's condense the village's state down to the minimal set of values that define it. There's the robot's current location and the collection of undelivered parcels, each of which has a current location and a destination address. That's it.

And while we're at it, let's make it so that we don't *change* this state when the robot moves but rather compute a *new* state for the situation after the move.

```
class VillageState {
  constructor(place, parcels) {
    this.place = place;
    this.parcels = parcels;
  }
```

```
  move(destination) {
    if (!roadGraph[this.place].includes(destination)) {
      return this;
    } else {
      let parcels = this.parcels.map(p => {
        if (p.place != this.place) return p;
        return {place: destination, address: p.address};
      }).filter(p => p.place != p.address);
      return new VillageState(destination, parcels);
    }
  }
}
```

The `move` method is where the action happens. It first checks whether there is a road going from the current place to the destination, and if not, it returns the old state since this is not a valid move.

Then it creates a new state with the destination as the robot's new place. But it also needs to create a new set of parcels—parcels that the robot is carrying (that are at the robot's current place) need to be moved along to the new place. And parcels that are addressed to the new place need to be delivered—that is, they need to be removed from the set of undelivered parcels. The call to `map` takes care of the moving, and the call to `filter` does the delivering.

Parcel objects aren't changed when they are moved but re-created. The `move` method gives us a new village state but leaves the old one entirely intact.

```
let first = new VillageState(
  "Post Office",
  [{place: "Post Office", address: "Alice's House"}]
);
let next = first.move("Alice's House");

console.log(next.place);
// → Alice's House
console.log(next.parcels);
// → []
console.log(first.place);
// → Post Office
```

The move causes the parcel to be delivered, and this is reflected in the next state. But the initial state still describes the situation where the robot is at the post office and the parcel is undelivered.

Persistent data

Data structures that don't change are called *immutable* or *persistent*. They behave a lot like strings and numbers in that they are who they are and stay that way, rather than containing different things at different times.

In JavaScript, just about everything *can* be changed, so working with values that are supposed to be persistent requires some restraint. There is a function called `Object.freeze` that changes an object so that writing to its properties is ignored. You could use that to make sure your objects aren't changed, if you want to be careful. Freezing does require the computer to do some extra work, and having updates ignored is just about as likely to confuse someone as having them do the wrong thing. So I usually prefer to just tell people that a given object shouldn't be messed with and hope they remember it.

```
let object = Object.freeze({value: 5});
object.value = 10;
console.log(object.value);
// → 5
```

Why am I going out of my way to not change objects when the language is obviously expecting me to?

Because it helps me understand my programs. This is about complexity management again. When the objects in my system are fixed, stable things, I can consider operations on them in isolation—moving to Alice's house from a given start state always produces the same new state. When objects change over time, that adds a whole new dimension of complexity to this kind of reasoning.

For a small system like the one we are building in this chapter, we could handle that bit of extra complexity. But the most important limit on what kind of systems we can build is how much we can understand. Anything that makes your code easier to understand makes it possible to build a more ambitious system.

Unfortunately, although understanding a system built on persistent data structures is easier, *designing* one, especially when your programming language isn't helping, can be a little harder. We'll look for opportunities to use persistent data structures in this book, but we'll also be using changeable ones.

Simulation

A delivery robot looks at the world and decides in which direction it wants to move. As such, we could say that a robot is a function that takes a `VillageState` object and returns the name of a nearby place.

Because we want robots to be able to remember things, so that they can make and execute plans, we also pass them their memory and allow them to return a new memory. Thus, the thing a robot returns is an object containing both the direction it wants to move in and a memory value that will be given back to it the next time it is called.

```
function runRobot(state, robot, memory) {
  for (let turn = 0;; turn++) {
    if (state.parcels.length == 0) {
      console.log(`Done in ${turn} turns`);
      break;
    }
    let action = robot(state, memory);
    state = state.move(action.direction);
    memory = action.memory;
    console.log(`Moved to ${action.direction}`);
  }
}
```

Consider what a robot has to do to "solve" a given state. It must pick up all parcels by visiting every location that has a parcel and deliver them by visiting every location that a parcel is addressed to, but only after picking up the parcel.

What is the dumbest strategy that could possibly work? The robot could just walk in a random direction every turn. That means, with great likelihood, it will eventually run into all parcels and then also at some point reach the place where they should be delivered.

Here's what that could look like:

```
function randomPick(array) {
  let choice = Math.floor(Math.random() * array.length);
  return array[choice];
}

function randomRobot(state) {
  return {direction: randomPick(roadGraph[state.place])};
}
```

Remember that `Math.random()` returns a number between zero and one—but always below one. Multiplying such a number by the length of an array and then applying `Math.floor` to it gives us a random index for the array.

Since this robot does not need to remember anything, it ignores its second argument (remember that JavaScript functions can be called with extra arguments without ill effects) and omits the `memory` property in its returned object.

To put this sophisticated robot to work, we'll first need a way to create a new state with some parcels. A static method (written here by directly adding a property to the constructor) is a good place to put that functionality.

```
VillageState.random = function(parcelCount = 5) {
  let parcels = [];
  for (let i = 0; i < parcelCount; i++) {
    let address = randomPick(Object.keys(roadGraph));
    let place;
    do {
      place = randomPick(Object.keys(roadGraph));
    } while (place == address);
    parcels.push({place, address});
  }
  return new VillageState("Post Office", parcels);
};
```

We don't want any parcels that are sent from the same place that they are addressed to. For this reason, the `do` loop keeps picking new places when it gets one that's equal to the address.

Let's start up a virtual world.

```
runRobot(VillageState.random(), randomRobot);
// → Moved to Marketplace
// → Moved to Town Hall
// →…
// → Done in 63 turns
```

It takes the robot a lot of turns to deliver the parcels because it isn't planning ahead very well. We'll address that soon.

The mail truck's route

We should be able to do a lot better than the random robot. An easy improvement would be to take a hint from the way real-world mail delivery works. If we find a route that passes all places in the village, the robot could run that route twice, at which point it is guaranteed to be done. Here is one such route (starting from the post office):

```
const mailRoute = [
  "Alice's House", "Cabin", "Alice's House", "Bob's House",
  "Town Hall", "Daria's House", "Ernie's House",
  "Grete's House", "Shop", "Grete's House", "Farm",
  "Marketplace", "Post Office"
];
```

To implement the route-following robot, we'll need to make use of robot memory. The robot keeps the rest of its route in its memory and drops the first element every turn.

```
function routeRobot(state, memory) {
  if (memory.length == 0) {
    memory = mailRoute;
  }
  return {direction: memory[0], memory: memory.slice(1)};
}
```

This robot is a lot faster already. It'll take a maximum of 26 turns (twice the 13-step route) but usually less.

Pathfinding

Still, I wouldn't really call blindly following a fixed route intelligent behavior. The robot could work more efficiently if it adjusted its behavior to the actual work that needs to be done.

To do that, it has to be able to deliberately move toward a given parcel or toward the location where a parcel has to be delivered. Doing that, even when the goal is more than one move away, will require some kind of route-finding function.

The problem of finding a route through a graph is a typical *search problem*. We can tell whether a given solution (a route) is a valid solution, but we can't

directly compute the solution the way we could for 2 + 2. Instead, we have to keep creating potential solutions until we find one that works.

The number of possible routes through a graph is infinite. But when searching for a route from *A* to *B*, we are interested only in the ones that start at *A*. We also don't care about routes that visit the same place twice—those are definitely not the most efficient route anywhere. So that cuts down on the number of routes that the route finder has to consider.

In fact, we are mostly interested in the *shortest* route. So we want to make sure we look at short routes before we look at longer ones. A good approach would be to "grow" routes from the starting point, exploring every reachable place that hasn't been visited yet, until a route reaches the goal. That way, we'll only explore routes that are potentially interesting, and we'll find the shortest route (or one of the shortest routes, if there are more than one) to the goal.

Here is a function that does this:

```
function findRoute(graph, from, to) {
  let work = [{at: from, route: []}];
  for (let i = 0; i < work.length; i++) {
    let {at, route} = work[i];
    for (let place of graph[at]) {
      if (place == to) return route.concat(place);
      if (!work.some(w => w.at == place)) {
        work.push({at: place, route: route.concat(place)});
      }
    }
  }
}
```

The exploring has to be done in the right order—the places that were reached first have to be explored first. We can't immediately explore a place as soon as we reach it because that would mean places reached *from there* would also be explored immediately, and so on, even though there may be other, shorter paths that haven't yet been explored.

Therefore, the function keeps a *work list*. This is an array of places that should be explored next, along with the route that got us there. It starts with just the start position and an empty route.

The search then operates by taking the next item in the list and exploring that, which means all roads going from that place are looked at. If one of them is the goal, a finished route can be returned. Otherwise, if we haven't looked at this place before, a new item is added to the list. If we have looked at it

before, since we are looking at short routes first, we've found either a longer route to that place or one precisely as long as the existing one, and we don't need to explore it.

You can visually imagine this as a web of known routes crawling out from the start location, growing evenly on all sides (but never tangling back into itself). As soon as the first thread reaches the goal location, that thread is traced back to the start, giving us our route.

Our code doesn't handle the situation where there are no more work items on the work list because we know that our graph is *connected*, meaning that every location can be reached from all other locations. We'll always be able to find a route between two points, and the search can't fail.

```
function goalOrientedRobot({place, parcels}, route) {
  if (route.length == 0) {
    let parcel = parcels[0];
    if (parcel.place != place) {
      route = findRoute(roadGraph, place, parcel.place);
    } else {
      route = findRoute(roadGraph, place, parcel.address);
    }
  }
  return {direction: route[0], memory: route.slice(1)};
}
```

This robot uses its memory value as a list of directions to move in, just like the route-following robot. Whenever that list is empty, it has to figure out what to do next. It takes the first undelivered parcel in the set and, if that parcel hasn't been picked up yet, plots a route toward it. If the parcel *has* been picked up, it still needs to be delivered, so the robot creates a route toward the delivery address instead.

This robot usually finishes the task of delivering 5 parcels in about 16 turns. That's slightly better than `routeRobot` but still definitely not optimal.

Exercises

Measuring a robot

It's hard to objectively compare robots by just letting them solve a few scenarios. Maybe one robot just happened to get easier tasks or the kind of tasks that it is good at, whereas the other didn't.

Write a function `compareRobots` that takes two robots (and their starting

memory). It should generate 100 tasks and let each of the robots solve each of these tasks. When done, it should output the average number of steps each robot took per task.

For the sake of fairness, make sure you give each task to both robots, rather than generating different tasks per robot.

Robot efficiency

Can you write a robot that finishes the delivery task faster than `goalOrientedRobot`? If you observe that robot's behavior, what obviously stupid things does it do? How could those be improved?

If you solved the previous exercise, you might want to use your `compareRobots` function to verify whether you improved the robot.

Persistent group

Most data structures provided in a standard JavaScript environment aren't very well suited for persistent use. Arrays have `slice` and `concat` methods, which allow us to easily create new arrays without damaging the old one. But `Set`, for example, has no methods for creating a new set with an item added or removed.

Write a new class `PGroup`, similar to the `Group` class from Chapter 6, which stores a set of values. Like `Group`, it has `add`, `delete`, and `has` methods.

Its `add` method, however, should return a *new* `PGroup` instance with the given member added and leave the old one unchanged. Similarly, `delete` creates a new instance without a given member.

The class should work for values of any type, not just strings. It does *not* have to be efficient when used with large amounts of values.

The constructor shouldn't be part of the class's interface (though you'll definitely want to use it internally). Instead, there is an empty instance, `PGroup.empty`, that can be used as a starting value.

Why do you need only one `PGroup.empty` value, rather than having a function that creates a new, empty map every time?

> *"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."*
>
> —Brian Kernighan and P.J. Plauger, The Elements of Programming Style

CHAPTER 8

Bugs and Errors

Flaws in computer programs are usually called *bugs*. It makes programmers feel good to imagine them as little things that just happen to crawl into our work. In reality, of course, we put them there ourselves.

If a program is crystallized thought, you can roughly categorize bugs into those caused by the thoughts being confused and those caused by mistakes introduced while converting a thought to code. The former type is generally harder to diagnose and fix than the latter.

Language

Many mistakes could be pointed out to us automatically by the computer, if it knew enough about what we're trying to do. But here JavaScript's looseness is a hindrance. Its concept of bindings and properties is vague enough that it will rarely catch typos before actually running the program. And even then, it allows you to do some clearly nonsensical things without complaint, such as computing `true * "monkey"`.

There are some things that JavaScript does complain about. Writing a program that does not follow the language's grammar will immediately make the computer complain. Other things, such as calling something that's not a function or looking up a property on an undefined value, will cause an error to be reported when the program tries to perform the action.

But often, your nonsense computation will merely produce `NaN` (not a number) or an undefined value, while the program happily continues, convinced that it's doing something meaningful. The mistake will manifest itself only later, after the bogus value has traveled through several functions. It might not trigger an error at all but silently cause the program's output to be wrong. Finding the source of such problems can be difficult.

The process of finding mistakes—bugs—in programs is called *debugging*.

Strict mode

JavaScript can be made a *little* stricter by enabling *strict mode*. This is done by putting the string `"use strict"` at the top of a file or a function body. Here's an example:

```
function canYouSpotTheProblem() {
  "use strict";
  for (counter = 0; counter < 10; counter++) {
    console.log("Happy happy");
  }
}

canYouSpotTheProblem();
// → ReferenceError: counter is not defined
```

Normally, when you forget to put `let` in front of your binding, as with `counter` in the example, JavaScript quietly creates a global binding and uses that. In strict mode, an error is reported instead. This is very helpful. It should be noted, though, that this doesn't work when the binding in question already exists as a global binding. In that case, the loop will still quietly overwrite the value of the binding.

Another change in strict mode is that the `this` binding holds the value `undefined` in functions that are not called as methods. When making such a call outside of strict mode, `this` refers to the global scope object, which is an object whose properties are the global bindings. So if you accidentally call a method or constructor incorrectly in strict mode, JavaScript will produce an error as soon as it tries to read something from `this`, rather than happily writing to the global scope.

For example, consider the following code, which calls a constructor function without the `new` keyword so that its `this` will *not* refer to a newly constructed object:

```
function Person(name) { this.name = name; }
let ferdinand = Person("Ferdinand"); // oops
console.log(name);
// → Ferdinand
```

So the bogus call to `Person` succeeded but returned an undefined value and created the global binding `name`. In strict mode, the result is different.

```
"use strict";
```

```
function Person(name) { this.name = name; }
let ferdinand = Person("Ferdinand"); // forgot new
// → TypeError: Cannot set property 'name' of undefined
```

We are immediately told that something is wrong. This is helpful.

Fortunately, constructors created with the `class` notation will always complain if they are called without `new`, making this less of a problem even in non-strict mode.

Strict mode does a few more things. It disallows giving a function multiple parameters with the same name and removes certain problematic language features entirely (such as the `with` statement, which is so wrong it is not further discussed in this book).

In short, putting `"use strict"` at the top of your program rarely hurts and might help you spot a problem.

Types

Some languages want to know the types of all your bindings and expressions before even running a program. They will tell you right away when a type is used in an inconsistent way. JavaScript considers types only when actually running the program, and even there often tries to implicitly convert values to the type it expects, so it's not much help.

Still, types provide a useful framework for talking about programs. A lot of mistakes come from being confused about the kind of value that goes into or comes out of a function. If you have that information written down, you're less likely to get confused.

You could add a comment like the following before the `goalOrientedRobot` function from the previous chapter to describe its type:

```
// (VillageState, Array) → {direction: string, memory: Array}
function goalOrientedRobot(state, memory) {
  // ...
}
```

There are a number of different conventions for annotating JavaScript programs with types.

One thing about types is that they need to introduce their own complexity to be able to describe enough code to be useful. What do you think would be the type of the `randomPick` function that returns a random element from an

array? You'd need to introduce a *type variable*, T, which can stand in for any type, so that you can give `randomPick` a type like `([T])→T` (function from an array of Ts to a T).

When the types of a program are known, it is possible for the computer to *check* them for you, pointing out mistakes before the program is run. There are several JavaScript dialects that add types to the language and check them. The most popular one is called TypeScript. If you are interested in adding more rigor to your programs, I recommend you give it a try.

In this book, we'll continue using raw, dangerous, untyped JavaScript code.

Testing

If the language is not going to do much to help us find mistakes, we'll have to find them the hard way: by running the program and seeing whether it does the right thing.

Doing this by hand, again and again, is a really bad idea. Not only is it annoying, it also tends to be ineffective since it takes too much time to exhaustively test everything every time you make a change.

Computers are good at repetitive tasks, and testing is the ideal repetitive task. Automated testing is the process of writing a program that tests another program. Writing tests is a bit more work than testing manually, but once you've done it, you gain a kind of superpower: it takes you only a few seconds to verify that your program still behaves properly in all the situations you wrote tests for. When you break something, you'll immediately notice, rather than randomly running into it at some later time.

Tests usually take the form of little labeled programs that verify some aspect of your code. For example, a set of tests for the (standard, probably already tested by someone else) `toUpperCase` method might look like this:

```
function test(label, body) {
  if (!body()) console.log(`Failed: ${label}`);
}

test("convert Latin text to uppercase", () => {
  return "hello".toUpperCase() == "HELLO";
});
test("convert Greek text to uppercase", () => {
  return "Χαίρετε".toUpperCase() == "ΧΑΊΡΕΤΕ";
});
test("don't convert case-less characters", () => {
  return "مرحبا".toUpperCase() == "مرحبا";
```

```
});
```

Writing tests like this tends to produce rather repetitive, awkward code. Fortunately, there exist pieces of software that help you build and run collections of tests (*test suites*) by providing a language (in the form of functions and methods) suited to expressing tests and by outputting informative information when a test fails. These are usually called *test runners*.

Some code is easier to test than other code. Generally, the more external objects that the code interacts with, the harder it is to set up the context in which to test it. The style of programming shown in the previous chapter, which uses self-contained persistent values rather than changing objects, tends to be easy to test.

Debugging

Once you notice there is something wrong with your program because it misbehaves or produces errors, the next step is to figure out *what* the problem is.

Sometimes it is obvious. The error message will point at a specific line of your program, and if you look at the error description and that line of code, you can often see the problem.

But not always. Sometimes the line that triggered the problem is simply the first place where a flaky value produced elsewhere gets used in an invalid way. If you have been solving the exercises in earlier chapters, you will probably have already experienced such situations.

The following example program tries to convert a whole number to a string in a given base (decimal, binary, and so on) by repeatedly picking out the last digit and then dividing the number to get rid of this digit. But the strange output that it currently produces suggests that it has a bug.

```
function numberToString(n, base = 10) {
  let result = "", sign = "";
  if (n < 0) {
    sign = "-";
    n = -n;
  }
  do {
    result = String(n % base) + result;
    n /= base;
  } while (n > 0);
  return sign + result;
```

```
}
console.log(numberToString(13, 10));
// → 1.5e-3231.3e-3221.3e-3211.3e-3201.3e-3191.3e…-3181.3
```

Even if you see the problem already, pretend for a moment that you don't. We know that our program is malfunctioning, and we want to find out why.

This is where you must resist the urge to start making random changes to the code to see whether that makes it better. Instead, *think*. Analyze what is happening and come up with a theory of why it might be happening. Then, make additional observations to test this theory—or, if you don't yet have a theory, make additional observations to help you come up with one.

Putting a few strategic `console.log` calls into the program is a good way to get additional information about what the program is doing. In this case, we want `n` to take the values `13`, `1`, and then `0`. Let's write out its value at the start of the loop.

```
13
1.3
0.13
0.013…

1.5e-323
```

Right. Dividing 13 by 10 does not produce a whole number. Instead of `n /= base`, what we actually want is `n = Math.floor(n / base)` so that the number is properly "shifted" to the right.

An alternative to using `console.log` to peek into the program's behavior is to use the *debugger* capabilities of your browser. Browsers come with the ability to set a *breakpoint* on a specific line of your code. When the execution of the program reaches a line with a breakpoint, it is paused, and you can inspect the values of bindings at that point. I won't go into details, as debuggers differ from browser to browser, but look in your browser's developer tools or search the Web for more information.

Another way to set a breakpoint is to include a `debugger` statement (consisting of simply that keyword) in your program. If the developer tools of your browser are active, the program will pause whenever it reaches such a statement.

Error propagation

Not all problems can be prevented by the programmer, unfortunately. If your program communicates with the outside world in any way, it is possible to get malformed input, to become overloaded with work, or to have the network fail.

If you're programming only for yourself, you can afford to just ignore such problems until they occur. But if you build something that is going to be used by anybody else, you usually want the program to do better than just crash. Sometimes the right thing to do is take the bad input in stride and continue running. In other cases, it is better to report to the user what went wrong and then give up. But in either situation, the program has to actively do something in response to the problem.

Say you have a function `promptNumber` that asks the user for a number and returns it. What should it return if the user inputs "orange"?

One option is to make it return a special value. Common choices for such values are `null`, `undefined`, or -1.

```
function promptNumber(question) {
  let result = Number(prompt(question));
  if (Number.isNaN(result)) return null;
  else return result;
}

console.log(promptNumber("How many trees do you see?"));
```

Now any code that calls `promptNumber` must check whether an actual number was read and, failing that, must somehow recover—maybe by asking again or by filling in a default value. Or it could again return a special value to *its* caller to indicate that it failed to do what it was asked.

In many situations, mostly when errors are common and the caller should be explicitly taking them into account, returning a special value is a good way to indicate an error. It does, however, have its downsides. First, what if the function can already return every possible kind of value? In such a function, you'll have to do something like wrap the result in an object to be able to distinguish success from failure.

```
function lastElement(array) {
  if (array.length == 0) {
    return {failed: true};
  } else {
    return {element: array[array.length - 1]};
  }
```

```
}
```

The second issue with returning special values is that it can lead to awkward code. If a piece of code calls `promptNumber` 10 times, it has to check 10 times whether `null` was returned. And if its response to finding `null` is to simply return `null` itself, callers of the function will in turn have to check for it, and so on.

Exceptions

When a function cannot proceed normally, what we would *like* to do is just stop what we are doing and immediately jump to a place that knows how to handle the problem. This is what *exception handling* does.

Exceptions are a mechanism that makes it possible for code that runs into a problem to *raise* (or *throw*) an exception. An exception can be any value. Raising one somewhat resembles a super-charged return from a function: it jumps out of not just the current function but also its callers, all the way down to the first call that started the current execution. This is called *unwinding the stack*. You may remember the stack of function calls that was mentioned in Chapter 3. An exception zooms down this stack, throwing away all the call contexts it encounters.

If exceptions always zoomed right down to the bottom of the stack, they would not be of much use. They'd just provide a novel way to blow up your program. Their power lies in the fact that you can set "obstacles" along the stack to *catch* the exception as it is zooming down. Once you've caught an exception, you can do something with it to address the problem and then continue to run the program.

Here's an example:

```
function promptDirection(question) {
  let result = prompt(question);
  if (result.toLowerCase() == "left") return "L";
  if (result.toLowerCase() == "right") return "R";
  throw new Error("Invalid direction: " + result);
}

function look() {
  if (promptDirection("Which way?") == "L") {
    return "a house";
  } else {
    return "two angry bears";
```

```
  }
}

try {
  console.log("You see", look());
} catch (error) {
  console.log("Something went wrong: " + error);
}
```

The `throw` keyword is used to raise an exception. Catching one is done by wrapping a piece of code in a `try` block, followed by the keyword `catch`. When the code in the `try` block causes an exception to be raised, the `catch` block is evaluated, with the name in parentheses bound to the exception value. After the `catch` block finishes—or if the `try` block finishes without problems—the program proceeds beneath the entire `try/catch` statement.

In this case, we used the `Error` constructor to create our exception value. This is a standard JavaScript constructor that creates an object with a `message` property. In most JavaScript environments, instances of this constructor also gather information about the call stack that existed when the exception was created, a so-called *stack trace*. This information is stored in the `stack` property and can be helpful when trying to debug a problem: it tells us the function where the problem occurred and which functions made the failing call.

Note that the `look` function completely ignores the possibility that `promptDirection` might go wrong. This is the big advantage of exceptions: error-handling code is necessary only at the point where the error occurs and at the point where it is handled. The functions in between can forget all about it.

Well, almost...

Cleaning up after exceptions

The effect of an exception is another kind of control flow. Every action that might cause an exception, which is pretty much every function call and property access, might cause control to suddenly leave your code.

This means when code has several side effects, even if its "regular" control flow looks like they'll always all happen, an exception might prevent some of them from taking place.

Here is some really bad banking code.

```
const accounts = {
  a: 100,
```

```
  b: 0,
  c: 20
};

function getAccount() {
  let accountName = prompt("Enter an account name");
  if (!accounts.hasOwnProperty(accountName)) {
    throw new Error(`No such account: ${accountName}`);
  }
  return accountName;
}

function transfer(from, amount) {
  if (accounts[from] < amount) return;
  accounts[from] -= amount;
  accounts[getAccount()] += amount;
}
```

The `transfer` function transfers a sum of money from a given account to another, asking for the name of the other account in the process. If given an invalid account name, `getAccount` throws an exception.

But `transfer` *first* removes the money from the account and *then* calls `getAccount` before it adds it to another account. If it is broken off by an exception at that point, it'll just make the money disappear.

That code could have been written a little more intelligently, for example by calling `getAccount` before it starts moving money around. But often problems like this occur in more subtle ways. Even functions that don't look like they will throw an exception might do so in exceptional circumstances or when they contain a programmer mistake.

One way to address this is to use fewer side effects. Again, a programming style that computes new values instead of changing existing data helps. If a piece of code stops running in the middle of creating a new value, no one ever sees the half-finished value, and there is no problem.

But that isn't always practical. So there is another feature that `try` statements have. They may be followed by a `finally` block either instead of or in addition to a `catch` block. A `finally` block says "no matter *what* happens, run this code after trying to run the code in the `try` block."

```
function transfer(from, amount) {
  if (accounts[from] < amount) return;
  let progress = 0;
  try {
```

```
    accounts[from] -= amount;
    progress = 1;
    accounts[getAccount()] += amount;
    progress = 2;
  } finally {
    if (progress == 1) {
      accounts[from] += amount;
    }
  }
}
```

This version of the function tracks its progress, and if, when leaving, it notices that it was aborted at a point where it had created an inconsistent program state, it repairs the damage it did.

Note that even though the `finally` code is run when an exception is thrown in the `try` block, it does not interfere with the exception. After the `finally` block runs, the stack continues unwinding.

Writing programs that operate reliably even when exceptions pop up in unexpected places is hard. Many people simply don't bother, and because exceptions are typically reserved for exceptional circumstances, the problem may occur so rarely that it is never even noticed. Whether that is a good thing or a really bad thing depends on how much damage the software will do when it fails.

Selective catching

When an exception makes it all the way to the bottom of the stack without being caught, it gets handled by the environment. What this means differs between environments. In browsers, a description of the error typically gets written to the JavaScript console (reachable through the browser's Tools or Developer menu). Node.js, the browserless JavaScript environment we will discuss in Chapter 20, is more careful about data corruption. It aborts the whole process when an unhandled exception occurs.

For programmer mistakes, just letting the error go through is often the best you can do. An unhandled exception is a reasonable way to signal a broken program, and the JavaScript console will, on modern browsers, provide you with some information about which function calls were on the stack when the problem occurred.

For problems that are *expected* to happen during routine use, crashing with an unhandled exception is a terrible strategy.

Invalid uses of the language, such as referencing a nonexistent binding, looking up a property on `null`, or calling something that's not a function, will also result in exceptions being raised. Such exceptions can also be caught.

When a `catch` body is entered, all we know is that *something* in our `try` body caused an exception. But we don't know *what* did or *which* exception it caused.

JavaScript (in a rather glaring omission) doesn't provide direct support for selectively catching exceptions: either you catch them all or you don't catch any. This makes it tempting to *assume* that the exception you get is the one you were thinking about when you wrote the `catch` block.

But it might not be. Some other assumption might be violated, or you might have introduced a bug that is causing an exception. Here is an example that *attempts* to keep on calling `promptDirection` until it gets a valid answer:

```
for (;;) {
  try {
    let dir = promtDirection("Where?"); // ← typo!
    console.log("You chose ", dir);
    break;
  } catch (e) {
    console.log("Not a valid direction. Try again.");
  }
}
```

The `for (;;)` construct is a way to intentionally create a loop that doesn't terminate on its own. We break out of the loop only when a valid direction is given. *But* we misspelled `promptDirection`, which will result in an "undefined variable" error. Because the `catch` block completely ignores its exception value (`e`), assuming it knows what the problem is, it wrongly treats the binding error as indicating bad input. Not only does this cause an infinite loop, it "buries" the useful error message about the misspelled binding.

As a general rule, don't blanket-catch exceptions unless it is for the purpose of "routing" them somewhere—for example, over the network to tell another system that our program crashed. And even then, think carefully about how you might be hiding information.

So we want to catch a *specific* kind of exception. We can do this by checking in the `catch` block whether the exception we got is the one we are interested in and rethrowing it otherwise. But how do we recognize an exception?

We could compare its `message` property against the error message we happen to expect. But that's a shaky way to write code—we'd be using information that's intended for human consumption (the message) to make a programmatic

decision. As soon as someone changes (or translates) the message, the code will stop working.

Rather, let's define a new type of error and use `instanceof` to identify it.

```
class InputError extends Error {}

function promptDirection(question) {
  let result = prompt(question);
  if (result.toLowerCase() == "left") return "L";
  if (result.toLowerCase() == "right") return "R";
  throw new InputError("Invalid direction: " + result);
}
```

The new error class extends `Error`. It doesn't define its own constructor, which means that it inherits the `Error` constructor, which expects a string message as argument. In fact, it doesn't define anything at all—the class is empty. `InputError` objects behave like `Error` objects, except that they have a different class by which we can recognize them.

Now the loop can catch these more carefully.

```
for (;;) {
  try {
    let dir = promptDirection("Where?");
    console.log("You chose ", dir);
    break;
  } catch (e) {
    if (e instanceof InputError) {
      console.log("Not a valid direction. Try again.");
    } else {
      throw e;
    }
  }
}
```

This will catch only instances of `InputError` and let unrelated exceptions through. If you reintroduce the typo, the undefined binding error will be properly reported.

Assertions

Assertions are checks inside a program that verify that something is the way it is supposed to be. They are used not to handle situations that can come up

in normal operation but to find programmer mistakes.

If, for example, `firstElement` is described as a function that should never be called on empty arrays, we might write it like this:

```
function firstElement(array) {
  if (array.length == 0) {
    throw new Error("firstElement called with []");
  }
  return array[0];
}
```

Now, instead of silently returning undefined (which you get when reading an array property that does not exist), this will loudly blow up your program as soon as you misuse it. This makes it less likely for such mistakes to go unnoticed and easier to find their cause when they occur.

I do not recommend trying to write assertions for every possible kind of bad input. That'd be a lot of work and would lead to very noisy code. You'll want to reserve them for mistakes that are easy to make (or that you find yourself making).

Summary

Mistakes and bad input are facts of life. An important part of programming is finding, diagnosing, and fixing bugs. Problems can become easier to notice if you have an automated test suite or add assertions to your programs.

Problems caused by factors outside the program's control should usually be handled gracefully. Sometimes, when the problem can be handled locally, special return values are a good way to track them. Otherwise, exceptions may be preferable.

Throwing an exception causes the call stack to be unwound until the next enclosing `try/catch` block or until the bottom of the stack. The exception value will be given to the `catch` block that catches it, which should verify that it is actually the expected kind of exception and then do something with it. To help address the unpredictable control flow caused by exceptions, `finally` blocks can be used to ensure that a piece of code *always* runs when a block finishes.

EXERCISES

RETRY

Say you have a function `primitiveMultiply` that in 20 percent of cases multiplies two numbers and in the other 80 percent of cases raises an exception of type `MultiplicatorUnitFailure`. Write a function that wraps this clunky function and just keeps trying until a call succeeds, after which it returns the result.

Make sure you handle only the exceptions you are trying to handle.

THE LOCKED BOX

Consider the following (rather contrived) object:

```
const box = {
  locked: true,
  unlock() { this.locked = false; },
  lock() { this.locked = true;  },
  _content: [],
  get content() {
    if (this.locked) throw new Error("Locked!");
    return this._content;
  }
};
```

It is a box with a lock. There is an array in the box, but you can get at it only when the box is unlocked. Directly accessing the private `_content` property is forbidden.

Write a function called `withBoxUnlocked` that takes a function value as argument, unlocks the box, runs the function, and then ensures that the box is locked again before returning, regardless of whether the argument function returned normally or threw an exception.

For extra points, make sure that if you call `withBoxUnlocked` when the box is already unlocked, the box stays unlocked.

> *"Some people, when confronted with a problem, think 'I know, I'll use regular expressions.' Now they have two problems."*
>
> —Jamie Zawinski

CHAPTER 9

Regular Expressions

Programming tools and techniques survive and spread in a chaotic, evolutionary way. It's not always the pretty or 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 eighteenPlus = /eighteen\+/;
```

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 allow us to express 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]`.

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

So 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 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. But the period itself, between square brackets, loses its special meaning. The same goes for other special characters, such as `+`.

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 notBinary = /[^01]/;
console.log(notBinary.test("1100100010100110"));
// → false
console.log(notBinary.test("1100100010200110"));
// → true
```

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. So, `{5,}` means five or more times.

Grouping subexpressions

To use an operator like `*` or `+` on more than one element at a time, you have to 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("Boohoooohoohooo"));
// → 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`. Similarly, when a group is matched multiple times, only the last match ends up in the array.

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

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, in which we discuss the built-in way to represent date and time values in JavaScript.

The Date class

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

```
console.log(new Date());
// → Mon Nov 13 2017 16:19:11 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 (`98` or `119`) 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`.

Word and string boundaries

Unfortunately, `getDate` will also happily extract the nonsensical date 00-1-3000 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 `$`. The caret matches the start of the input string, whereas the dollar sign matches the end. So, `/^\d+$/` 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).

If, on the other hand, we just want to make sure the date starts and ends on a word boundary, we can use the marker `\b`. A word boundary can be the start or end of the string or any point in the string that has a word character (as in `\w`) on one side and a nonword character on the other.

```
console.log(/cat/.test("concatenate"));
// → true
console.log(/\bcat\b/.test("concatenate"));
// → false
```

Note that a boundary marker doesn't match an actual character. It just enforces that the regular expression matches only when a certain condition holds at the place where it appears in the pattern.

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. So I can say this:

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

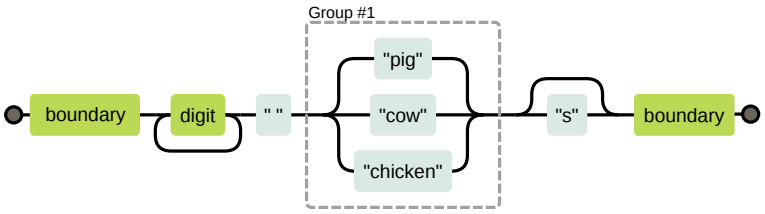
```
// → true
console.log(animalCount.test("15 pigchickens"));
// → false
```

Parentheses can be used to limit the part of the pattern that the pipe operator applies to, 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:



Our expression matches if we can find a path from the left side of the diagram to the right side. 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.

So if we try to match `"the 3 pigs"` from position 4, our progress through the flow chart would look like this:

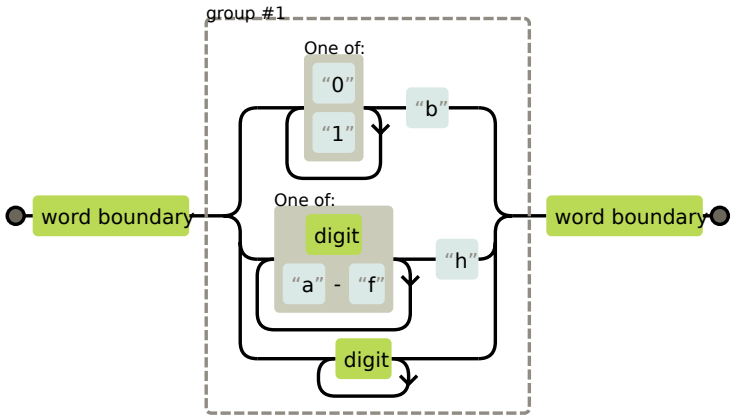
- At position 4, there is a word boundary, so we can move past the first box.
- Still at position 4, we find a digit, so we can also move past the second box.
- At position 5, one path loops back to before the second (digit) box, while the other moves forward through the box that holds a single space

character. There is a space here, not a digit, so we must take the second path.

- We are now at position 6 (the start of *pigs*) and at the three-way branch in the diagram. We don't see *cow* or *chicken* here, but we do see *pig*, so we take that branch.
- At position 9, after the three-way branch, one path skips the *s* box and goes straight to the final word boundary, while the other path matches an *s*. There is an *s* character here, not a word boundary, so we go through the *s* box.
- We're at position 10 (the end of the string) and can match only a word boundary. The end of a string counts as a word boundary, so we go through the last box and have successfully matched this string.

Backtracking

The regular expression `/\b([01]+b|[\da-f]+h|\d+)\b/` 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:



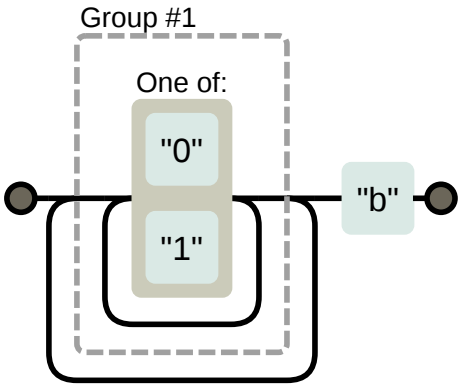
When matching this expression, it will often happen that the top (binary) branch is entered even though the input does not actually contain a binary number. When matching the string `"103"`, for example, it becomes clear only at the 3 that we are in the wrong branch. The string *does* match the expression, 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, it will start trying the branch for hexadecimal numbers, which fails again because there is no *h* after the number. So it 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 to 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
```

It would have been sensible if the choice between replacing one match or all matches was made through an additional argument to `replace` or by providing a different method, `replaceAll`. But for some unfortunate reason, the choice relies on a property of the regular expression instead.

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\nWadler, Philip"
    .replace(/(\w+), (\w+)/g, "$2 $1"));
// → Barbara Liskov
//   John McCarthy
//   Philip Wadler
```

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 a small example:

```
let s = "the cia and fbi";
console.log(s.replace(/\b(fbi|cia)\b/g,
            str => str.toUpperCase()));
// → the CIA and FBI
```

Here's a more interesting one:

```
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+) (\w+)/g, minusOne));
// → no lemon, 1 cabbage, and 100 eggs
```

This takes a string, finds all occurrences of a number followed by an alphanumeric word, and returns a string wherein every such occurrence is decremented by one.

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

Greed

It is possible to 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 *or* 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, consider the nongreedy variant first.

Dynamically creating RegExp objects

There are cases where you might not know the exact pattern you need to match against when you are writing your code. Say you want to look for the user's name in a piece of text and enclose it in underscore characters to make it stand out. Since you will know the name only once the program is actually running, you can't use the slash-based notation.

But you can build up a string and use the `RegExp` constructor on that. Here's an example:

```
let name = "harry";
let text = "Harry is a suspicious character.";
let regexp = new RegExp("\\b(" + name + ")\\b", "gi");
console.log(text.replace(regexp, "_$1_"));
// → _Harry_ is a suspicious character.
```

When creating the `\b` boundary markers, 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 `"dea+hl[]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 = "dea+hl[]rd";
let text = "This dea+hl[]rd guy is super annoying.";
let escaped = name.replace(/[\\[.+*?(){|^$]/g, "\\$&");
let regexp = new RegExp("\\b" + escaped + "\\b", "gi");
console.log(text.replace(regexp, "_$&_"));
// → This _dea+hl[]rd_ guy is super annoying.
```

The search method

The `indexOf` method on strings cannot be called with a regular expression. But 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 *in*convenient 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("xyzzy");
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 zero, 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 that was 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 only places where you want to use them.

Looping over matches

A common thing to do is to scan through all occurrences of a pattern in a string, in a way that gives us access to the match object in the loop body. We can do this by using `lastIndex` and `exec`.

```
let input = "A string with 3 numbers in it... 42 and 88.";
let number = /\b\d+\b/g;
let match;
while (match = number.exec(input)) {
  console.log("Found", match[0], "at", match.index);
}
// → Found 3 at 14
```

```
//   Found 42 at 33
//   Found 88 at 40
```

This makes use of the fact that the value of an assignment expression (`=`) is the assigned value. So by using `match = number.exec(input)` as the condition in the `while` statement, we perform the match at the start of each iteration, save its result in a binding, and stop looping when no more matches are found.

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 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;
  string.split(/\r?\n/).forEach(line => {
    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(...))` is similar to the trick of using an assignment as the condition for `while`. 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*(;.*)?$/`. Do you see how it works? The part between the parentheses will match comments, and the `?` makes sure it also matches lines containing only whitespace. When a line doesn't match any of the expected forms, the function throws an exception.

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 *ß*, 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.

Another problem is that, by default, regular expressions 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 the `{3}` part is applied only to the second one. Similarly, the dot matches a single code unit, not the two that make up the rose emoji.

You must add a `u` option (for Unicode) to your regular expression to make it treat such characters properly. The wrong behavior remains the default, unfortunately, because changing that might cause problems for existing code that depends on it.

Though this was only just standardized and is, at the time of writing, not widely supported yet, it is possible to use `\p` in a regular expression (that must have the Unicode option enabled) to match all characters to which the Unicode standard assigns a given property.

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

Unicode defines a number of useful properties, though finding the one that you need may not always be trivial. You can use the `\p{Property=Value}` notation to match any character that has the given value for that property. If the property name is left off, as in `\p{Name}`, the name is assumed to be either a binary property such as `Alphabetic` or a category such as `Number`.

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 *not* in a set of characters
`/[0-9]/`	Any character in a range of characters
`/x+/`	One or more occurrences of the pattern `x`
`/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
`/\b/`	A word boundary
`/^/`	Start of input
`/$/`	End of input

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 it 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 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 that they cannot cleanly express into them.

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 *https://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 any of the given substrings occur in a string. The regular expression should match only strings containing one of the substrings described. Do not worry about word boundaries unless explicitly mentioned. 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*.

"Write code that is easy to delete, not easy to extend."

—Tef, Programming is Terrible

CHAPTER 10

MODULES

The ideal program has a crystal-clear structure. The way it works is easy to explain, and each part plays a well-defined role.

A typical real program grows organically. New pieces of functionality are added as new needs come up. Structuring—and preserving structure—is additional work. It's work that will pay off only in the future, the *next* time someone works on the program. So it is tempting to neglect it and allow the parts of the program to become deeply entangled.

This causes two practical issues. First, understanding such a system is hard. If everything can touch everything else, it is difficult to look at any given piece in isolation. You are forced to build up a holistic understanding of the entire thing. Second, if you want to use any of the functionality from such a program in another situation, rewriting it may be easier than trying to disentangle it from its context.

The phrase "big ball of mud" is often used for such large, structureless programs. Everything sticks together, and when you try to pick out a piece, the whole thing comes apart, and your hands get dirty.

MODULES

Modules are an attempt to avoid these problems. A module is a piece of program that specifies which other pieces it relies on and which functionality it provides for other modules to use (its *interface*).

Module interfaces have a lot in common with object interfaces, as we saw them in Chapter 6. They make part of the module available to the outside world and keep the rest private. By restricting the ways in which modules interact with each other, the system becomes more like LEGO, where pieces interact through well-defined connectors, and less like mud, where everything mixes with everything.

The relations between modules are called *dependencies*. When a module needs a piece from another module, it is said to depend on that module. When

this fact is clearly specified in the module itself, it can be used to figure out which other modules need to be present to be able to use a given module and to automatically load dependencies.

To separate modules in that way, each needs its own private scope.

Just putting your JavaScript code into different files does not satisfy these requirements. The files still share the same global namespace. They can, intentionally or accidentally, interfere with each other's bindings. And the dependency structure remains unclear. We can do better, as we'll see later in the chapter.

Designing a fitting module structure for a program can be difficult. In the phase where you are still exploring the problem, trying different things to see what works, you might want to not worry about it too much since it can be a big distraction. Once you have something that feels solid, that's a good time to take a step back and organize it.

Packages

One of the advantages of building a program out of separate pieces, and being actually able to run those pieces on their own, is that you might be able to apply the same piece in different programs.

But how do you set this up? Say I want to use the `parseINI` function from Chapter 9 in another program. If it is clear what the function depends on (in this case, nothing), I can just copy all the necessary code into my new project and use it. But then, if I find a mistake in that code, I'll probably fix it in whichever program I'm working with at the time and forget to also fix it in the other program.

Once you start duplicating code, you'll quickly find yourself wasting time and energy moving copies around and keeping them up-to-date.

That's where *packages* come in. A package is a chunk of code that can be distributed (copied and installed). It may contain one or more modules and has information about which other packages it depends on. A package also usually comes with documentation explaining what it does so that people who didn't write it might still be able to use it.

When a problem is found in a package or a new feature is added, the package is updated. Now the programs that depend on it (which may also be packages) can upgrade to the new version.

Working in this way requires infrastructure. We need a place to store and find packages and a convenient way to install and upgrade them. In the JavaScript world, this infrastructure is provided by NPM (*https://npmjs.org*).

NPM is two things: an online service where one can download (and upload) packages and a program (bundled with Node.js) that helps you install and manage them.

At the time of writing, there are more than half a million different packages available on NPM. A large portion of those are rubbish, I should mention, but almost every useful, publicly available package can be found on there. For example, an INI file parser, similar to the one we built in Chapter 9, is available under the package name `ini`.

Chapter 20 will show how to install such packages locally using the `npm` command line program.

Having quality packages available for download is extremely valuable. It means that we can often avoid reinventing a program that 100 people have written before and get a solid, well-tested implementation at the press of a few keys.

Software is cheap to copy, so once someone has written it, distributing it to other people is an efficient process. But writing it in the first place *is* work, and responding to people who have found problems in the code, or who want to propose new features, is even more work.

By default, you own the copyright to the code you write, and other people may use it only with your permission. But because some people are just nice and because publishing good software can help make you a little bit famous among programmers, many packages are published under a license that explicitly allows other people to use it.

Most code on NPM is licensed this way. Some licenses require you to also publish code that you build on top of the package under the same license. Others are less demanding, just requiring that you keep the license with the code as you distribute it. The JavaScript community mostly uses the latter type of license. When using other people's packages, make sure you are aware of their license.

Improvised modules

Until 2015, the JavaScript language had no built-in module system. Yet people had been building large systems in JavaScript for more than a decade, and they *needed* modules.

So they designed their own module systems on top of the language. You can use JavaScript functions to create local scopes and objects to represent module interfaces.

This is a module for going between day names and numbers (as returned

by `Date`'s `getDay` method). Its interface consists of `weekDay.name` and `weekDay.number`, and it hides its local binding `names` inside the scope of a function expression that is immediately invoked.

```
const weekDay = function() {
  const names = ["Sunday", "Monday", "Tuesday", "Wednesday",
                 "Thursday", "Friday", "Saturday"];
  return {
    name(number) { return names[number]; },
    number(name) { return names.indexOf(name); }
  };
}();

console.log(weekDay.name(weekDay.number("Sunday")));
// → Sunday
```

This style of modules provides isolation, to a certain degree, but it does not declare dependencies. Instead, it just puts its interface into the global scope and expects its dependencies, if any, to do the same. For a long time this was the main approach used in web programming, but it is mostly obsolete now.

If we want to make dependency relations part of the code, we'll have to take control of loading dependencies. Doing that requires being able to execute strings as code. JavaScript can do this.

Evaluating data as code

There are several ways to take data (a string of code) and run it as part of the current program.

The most obvious way is the special operator `eval`, which will execute a string in the *current* scope. This is usually a bad idea because it breaks some of the properties that scopes normally have, such as it being easily predictable which binding a given name refers to.

```
const x = 1;
function evalAndReturnX(code) {
  eval(code);
  return x;
}

console.log(evalAndReturnX("var x = 2"));
// → 2
console.log(x);
```

```
// → 1
```

A less scary way of interpreting data as code is to use the `Function` constructor. It takes two arguments: a string containing a comma-separated list of argument names and a string containing the function body. It wraps the code in a function value so that it gets its own scope and won't do odd things with other scopes.

```
let plusOne = Function("n", "return n + 1;");
console.log(plusOne(4));
// → 5
```

This is precisely what we need for a module system. We can wrap the module's code in a function and use that function's scope as module scope.

CommonJS

The most widely used approach to bolted-on JavaScript modules is called *CommonJS modules*. Node.js uses it and is the system used by most packages on NPM.

The main concept in CommonJS modules is a function called `require`. When you call this with the module name of a dependency, it makes sure the module is loaded and returns its interface.

Because the loader wraps the module code in a function, modules automatically get their own local scope. All they have to do is call `require` to access their dependencies and put their interface in the object bound to `exports`.

This example module provides a date-formatting function. It uses two packages from NPM—`ordinal` to convert numbers to strings like `"1st"` and `"2nd"`, and `date-names` to get the English names for weekdays and months. It exports a single function, `formatDate`, which takes a `Date` object and a template string.

The template string may contain codes that direct the format, such as `YYYY` for the full year and `Do` for the ordinal day of the month. You could give it a string like `"MMMM Do YYYY"` to get output like "November 22nd 2017".

```
const ordinal = require("ordinal");
const {days, months} = require("date-names");

exports.formatDate = function(date, format) {
  return format.replace(/YYYY|M(MMM)?|Do?|dddd/g, tag => {
    if (tag == "YYYY") return date.getFullYear();
```

```
    if (tag == "M") return date.getMonth();
    if (tag == "MMMM") return months[date.getMonth()];
    if (tag == "D") return date.getDate();
    if (tag == "Do") return ordinal(date.getDate());
    if (tag == "dddd") return days[date.getDay()];
  });
};
```

The interface of `ordinal` is a single function, whereas `date-names` exports an object containing multiple things—`days` and `months` are arrays of names. Destructuring is very convenient when creating bindings for imported interfaces.

The module adds its interface function to `exports` so that modules that depend on it get access to it. We could use the module like this:

```
const {formatDate} = require("./format-date");

console.log(formatDate(new Date(2017, 9, 13),
                       "dddd the Do"));
// → Friday the 13th
```

We can define `require`, in its most minimal form, like this:

```
require.cache = Object.create(null);

function require(name) {
  if (!(name in require.cache)) {
    let code = readFile(name);
    let module = {exports: {}};
    require.cache[name] = module;
    let wrapper = Function("require, exports, module", code);
    wrapper(require, module.exports, module);
  }
  return require.cache[name].exports;
}
```

In this code, `readFile` is a made-up function that reads a file and returns its contents as a string. Standard JavaScript provides no such functionality—but different JavaScript environments, such as the browser and Node.js, provide their own ways of accessing files. The example just pretends that `readFile` exists.

To avoid loading the same module multiple times, `require` keeps a store (cache) of already loaded modules. When called, it first checks if the requested

module has been loaded and, if not, loads it. This involves reading the module's code, wrapping it in a function, and calling it.

The interface of the `ordinal` package we saw before is not an object but a function. A quirk of the CommonJS modules is that, though the module system will create an empty interface object for you (bound to `exports`), you can replace that with any value by overwriting `module.exports`. This is done by many modules to export a single value instead of an interface object.

By defining `require`, `exports`, and `module` as parameters for the generated wrapper function (and passing the appropriate values when calling it), the loader makes sure that these bindings are available in the module's scope.

The way the string given to `require` is translated to an actual filename or web address differs in different systems. When it starts with `"./"` or `"../"`, it is generally interpreted as relative to the current module's filename. So `"./format-date"` would be the file named `format-date.js` in the same directory.

When the name isn't relative, Node.js will look for an installed package by that name. In the example code in this chapter, we'll interpret such names as referring to NPM packages. We'll go into more detail on how to install and use NPM modules in Chapter 20.

Now, instead of writing our own INI file parser, we can use one from NPM.

```
const {parse} = require("ini");

console.log(parse("x = 10\ny = 20"));
// → {x: "10", y: "20"}
```

ECMAScript modules

CommonJS modules work quite well and, in combination with NPM, have allowed the JavaScript community to start sharing code on a large scale.

But they remain a bit of a duct-tape hack. The notation is slightly awkward—the things you add to `exports` are not available in the local scope, for example. And because `require` is a normal function call taking any kind of argument, not just a string literal, it can be hard to determine the dependencies of a module without running its code.

This is why the JavaScript standard from 2015 introduces its own, different module system. It is usually called *ES modules*, where *ES* stands for ECMAScript. The main concepts of dependencies and interfaces remain the same, but the details differ. For one thing, the notation is now integrated into

the language. Instead of calling a function to access a dependency, you use a special `import` keyword.

```
import ordinal from "ordinal";
import {days, months} from "date-names";

export function formatDate(date, format) { /* ... */ }
```

Similarly, the `export` keyword is used to export things. It may appear in front of a function, class, or binding definition (`let`, `const`, or `var`).

An ES module's interface is not a single value but a set of named bindings. The preceding module binds `formatDate` to a function. When you import from another module, you import the *binding*, not the value, which means an exporting module may change the value of the binding at any time, and the modules that import it will see its new value.

When there is a binding named `default`, it is treated as the module's main exported value. If you import a module like `ordinal` in the example, without braces around the binding name, you get its `default` binding. Such modules can still export other bindings under different names alongside their `default` export.

To create a default export, you write `export default` before an expression, a function declaration, or a class declaration.

```
export default ["Winter", "Spring", "Summer", "Autumn"];
```

It is possible to rename imported bindings using the word `as`.

```
import {days as dayNames} from "date-names";

console.log(dayNames.length);
// → 7
```

Another important difference is that ES module imports happen before a module's script starts running. That means `import` declarations may not appear inside functions or blocks, and the names of dependencies must be quoted strings, not arbitrary expressions.

At the time of writing, the JavaScript community is in the process of adopting this module style. But it has been a slow process. It took a few years, after the format was specified, for browsers and Node.js to start supporting it. And though they mostly support it now, this support still has issues, and the

discussion on how such modules should be distributed through NPM is still ongoing.

Many projects are written using ES modules and then automatically converted to some other format when published. We are in a transitional period in which two different module systems are used side by side, and it is useful to be able to read and write code in either of them.

Building and bundling

In fact, many JavaScript projects aren't even, technically, written in JavaScript. There are extensions, such as the type checking dialect mentioned in Chapter 8, that are widely used. People also often start using planned extensions to the language long before they have been added to the platforms that actually run JavaScript.

To make this possible, they *compile* their code, translating it from their chosen JavaScript dialect to plain old JavaScript—or even to a past version of JavaScript—so that old browsers can run it.

Including a modular program that consists of 200 different files in a web page produces its own problems. If fetching a single file over the network takes 50 milliseconds, loading the whole program takes 10 seconds, or maybe half that if you can load several files simultaneously. That's a lot of wasted time. Because fetching a single big file tends to be faster than fetching a lot of tiny ones, web programmers have started using tools that roll their programs (which they painstakingly split into modules) back into a single big file before they publish it to the Web. Such tools are called *bundlers*.

And we can go further. Apart from the number of files, the *size* of the files also determines how fast they can be transferred over the network. Thus, the JavaScript community has invented *minifiers*. These are tools that take a JavaScript program and make it smaller by automatically removing comments and whitespace, renaming bindings, and replacing pieces of code with equivalent code that take up less space.

So it is not uncommon for the code that you find in an NPM package or that runs on a web page to have gone through *multiple* stages of transformation—converted from modern JavaScript to historic JavaScript, from ES module format to CommonJS, bundled, and minified. We won't go into the details of these tools in this book since they tend to be boring and change rapidly. Just be aware that the JavaScript code you run is often not the code as it was written.

Module design

Structuring programs is one of the subtler aspects of programming. Any non-trivial piece of functionality can be modeled in various ways.

Good program design is subjective—there are trade-offs involved and matters of taste. The best way to learn the value of well-structured design is to read or work on a lot of programs and notice what works and what doesn't. Don't assume that a painful mess is "just the way it is". You can improve the structure of almost everything by putting more thought into it.

One aspect of module design is ease of use. If you are designing something that is intended to be used by multiple people—or even by yourself, in three months when you no longer remember the specifics of what you did—it is helpful if your interface is simple and predictable.

That may mean following existing conventions. A good example is the `ini` package. This module imitates the standard `JSON` object by providing `parse` and `stringify` (to write an INI file) functions, and, like `JSON`, converts between strings and plain objects. So the interface is small and familiar, and after you've worked with it once, you're likely to remember how to use it.

Even if there's no standard function or widely used package to imitate, you can keep your modules predictable by using simple data structures and doing a single, focused thing. Many of the INI-file parsing modules on NPM provide a function that directly reads such a file from the hard disk and parses it, for example. This makes it impossible to use such modules in the browser, where we don't have direct file system access, and adds complexity that would have been better addressed by *composing* the module with some file-reading function.

This points to another helpful aspect of module design—the ease with which something can be composed with other code. Focused modules that compute values are applicable in a wider range of programs than bigger modules that perform complicated actions with side effects. An INI file reader that insists on reading the file from disk is useless in a scenario where the file's content comes from some other source.

Relatedly, stateful objects are sometimes useful or even necessary, but if something can be done with a function, use a function. Several of the INI file readers on NPM provide an interface style that requires you to first create an object, then load the file into your object, and finally use specialized methods to get at the results. This type of thing is common in the object-oriented tradition, and it's terrible. Instead of making a single function call and moving on, you have to perform the ritual of moving your object through various states. And because the data is now wrapped in a specialized object type, all

code that interacts with it has to know about that type, creating unnecessary interdependencies.

Often defining new data structures can't be avoided—only a few basic ones are provided by the language standard, and many types of data have to be more complex than an array or a map. But when an array suffices, use an array.

An example of a slightly more complex data structure is the graph from Chapter 7. There is no single obvious way to represent a graph in JavaScript. In that chapter, we used an object whose properties hold arrays of strings—the other nodes reachable from that node.

There are several different pathfinding packages on NPM, but none of them uses this graph format. They usually allow the graph's edges to have a weight, which is the cost or distance associated with it. That isn't possible in our representation.

For example, there's the `dijkstrajs` package. A well-known approach to pathfinding, quite similar to our `findRoute` function, is called *Dijkstra's algorithm*, after Edsger Dijkstra, who first wrote it down. The `js` suffix is often added to package names to indicate the fact that they are written in JavaScript. This `dijkstrajs` package uses a graph format similar to ours, but instead of arrays, it uses objects whose property values are numbers—the weights of the edges.

So if we wanted to use that package, we'd have to make sure that our graph was stored in the format it expects. All edges get the same weight since our simplified model treats each road as having the same cost (one turn).

```
const {find_path} = require("dijkstrajs");

let graph = {};
for (let node of Object.keys(roadGraph)) {
  let edges = graph[node] = {};
  for (let dest of roadGraph[node]) {
    edges[dest] = 1;
  }
}

console.log(find_path(graph, "Post Office", "Cabin"));
// → ["Post Office", "Alice's House", "Cabin"]
```

This can be a barrier to composition—when various packages are using different data structures to describe similar things, combining them is difficult. Therefore, if you want to design for composability, find out what data struc-

tures other people are using and, when possible, follow their example.

Summary

Modules provide structure to bigger programs by separating the code into pieces with clear interfaces and dependencies. The interface is the part of the module that's visible from other modules, and the dependencies are the other modules that it makes use of.

Because JavaScript historically did not provide a module system, the CommonJS system was built on top of it. Then at some point it *did* get a built-in system, which now coexists uneasily with the CommonJS system.

A package is a chunk of code that can be distributed on its own. NPM is a repository of JavaScript packages. You can download all kinds of useful (and useless) packages from it.

Exercises

A modular robot

These are the bindings that the project from Chapter 7 creates:

```
roads
buildGraph
roadGraph
VillageState
runRobot
randomPick
randomRobot
mailRoute
routeRobot
findRoute
goalOrientedRobot
```

If you were to write that project as a modular program, what modules would you create? Which module would depend on which other module, and what would their interfaces look like?

Which pieces are likely to be available prewritten on NPM? Would you prefer to use an NPM package or write them yourself?