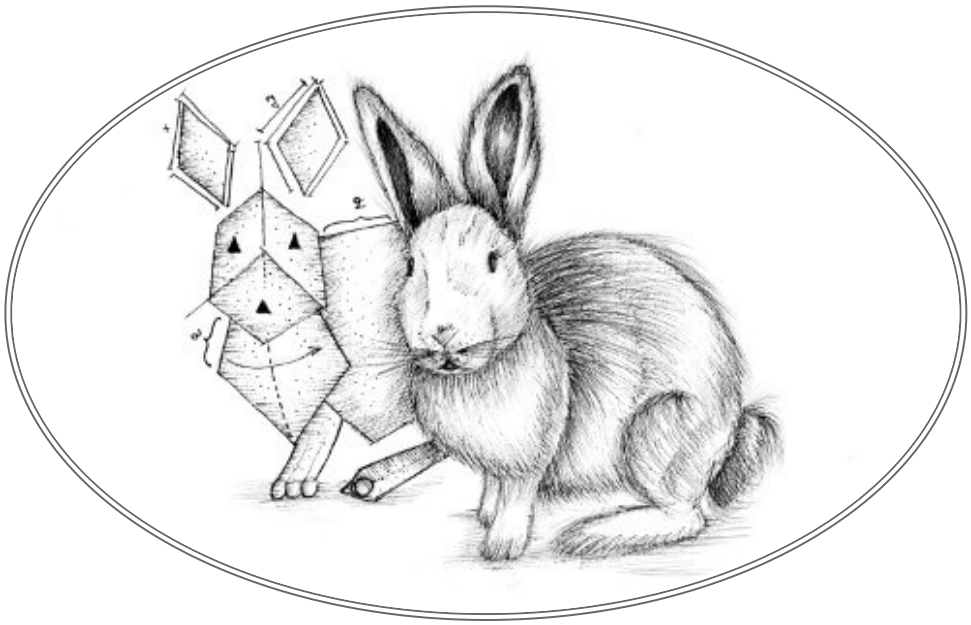
The Secret Life of Objects

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



Chapter 4 introduced JavaScript's objects as containers that hold other data.

In programming culture, *object-oriented programming* is a set of techniques that use objects 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 describes the way these ideas can be applied in JavaScript.

Abstract Data Types

The main idea in object-oriented programming is to use objects, or rather *types* of objects, as the unit of program organization. Setting up a program as a number of strictly separated object types provides a way to think about its structure and thus to enforce some kind of discipline, preventing everything from becoming entangled.

The way to do this is to think of objects somewhat like you'd think of an electric mixer or other consumer appliance. The people who design and assemble a mixer have to do specialized work requiring material science and understanding of electricity. They cover all that up in a smooth plastic shell so that the people who only want to mix pancake batter don't have to worry about all that—they only have to understand the few knobs that the mixer can be operated with.

Similarly, an *abstract data type*, or *object class*, is a subprogram that may contain arbitrarily complicated code but exposes a limited set of methods and properties that people working with it are supposed to use. This allows large programs to be built up out of a number of appliance types, limiting the degree to which these different parts are entangled by requiring them to only interact with each other in specific ways.

If a problem is found in one such object class, it can often be repaired or even completely rewritten without impacting the rest of the program. Even better, it may be possible to use object classes in multiple different programs, avoiding the need to recreate their functionality from scratch. You can think of JavaScript's built-in data structures, such as arrays and strings, as such reusable abstract data types.

Each abstract data type has an *interface*, the collection of operations that external code can perform on it. Any details beyond that interface are *encapsulated*, treated as internal to the type and of no concern to the rest of the program.

Even basic things like numbers can be thought of as an abstract data type whose interface allows us to add them, multiply them, compare them, and so on. In fact, the fixation on single *objects* as the main unit of organization in classical object-oriented programming is somewhat unfortunate since useful pieces of functionality often involve a group of different object classes working closely together.

METHODS

In JavaScript, methods are nothing more than properties that hold function values. This is a simple method:

```
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 fur and whiskers");
// → The white rabbit says 'Oh my fur and whiskers'
hungryRabbit.speak("Got any carrots?");
// → The hungry rabbit says 'Got any carrots?'
```

Typically a method needs to do something with the object on which it was called. 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 on which it was called.

You can think of `this` as an extra parameter that is passed to the function in a different way than regular parameters. If you want to provide 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(whiteRabbit, "Hurry");
// → The white rabbit says 'Hurry'
```

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:

```
let finder = {
  find(array) {
    return array.some(v => v == this.value);
  },
  value: 5
};
console.log(finder.find([4, 5]));
// → true
```

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

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

Prototypes

One way to create a rabbit object type with a `speak` method would be to create a helper function that has a rabbit type as parameter and returns an object holding that as its `type` property and our speak function in its `speak` property.

All rabbits share that same method. Especially for types with many methods, it would be nice if there was a way to keep a type's methods in a single place, rather than adding them to each object individually.

In JavaScript, *prototypes* are the way to do that. Objects can be linked to other objects, to magically get all the properties that other object has. Plain old objects created with `{}` notation are linked to an object called `Object.prototype`.

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

It looks like we just pulled a property out of an empty object. But in fact, `toString` is a method stored in `Object.prototype`, meaning it is available in most objects.

When an object gets a request for a property that it doesn't have, its prototype will be searched for the property. If that doesn't have it, the *prototype's* prototype is searched, and so on until an object without prototype is reached (`Object.prototype` is such an object).

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

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

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 blackRabbit = Object.create(protoRabbit);
blackRabbit.type = "black";
blackRabbit.speak("I am fear and darkness");
// → The black rabbit says 'I am fear and darkness'
```

The "proto" rabbit acts as a container for the properties shared by all rabbits. An individual rabbit object, like the black 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 free-form take on abstract data types or 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. Properties that differ per instance, such as our rabbits' `type` property, need to be stored directly in the objects themselves.

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's class notation makes it easier to define this type of function, along with a prototype object.

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

The `class` keyword starts a class declaration, which allows us to define a constructor and a set of methods together. Any number of methods may be written inside the declaration's braces. This code has the effect of defining a binding called `Rabbit`, which holds a function that runs the code in `constructor` and has a `prototype` property which holds the `speak` method.

This function cannot be called like a normal function. Constructors, in JavaScript, are called by putting the keyword `new` in front of them. Doing so

creates a fresh instance object whose prototype is the object from the function's `prototype` property, then runs the function with `this` bound to the new object, and finally returns the object.

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

In fact, `class` was only introduced in the 2015 edition of JavaScript. Any function can be used as a constructor, and before 2015, the way to define a class was to write a regular function and then manipulate its `prototype` property.

```
function ArchaicRabbit(type) {
  this.type = type;
}
ArchaicRabbit.prototype.speak = function(line) {
  console.log(`The ${this.type} rabbit says '${line}'`);
};
let oldSchoolRabbit = new ArchaicRabbit("old school");
```

For this reason, all non-arrow functions start with a `prototype` property holding an empty object.

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. The constructor function's `prototype` *property* holds the prototype used for instances created through it.

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

Constructors will typically add some per-instance properties to `this`. It is also possible to declare properties directly in the class declaration. Unlike methods, such properties are added to instance objects and not the prototype.

```
class Particle {
  speed = 0;
  constructor(position) {
    this.position = position;
  }
}
```

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
```

Private Properties

It is common for classes to define some properties and methods for internal use that are not part of their interface. These are called *private* properties, as opposed to *public* ones, which are part of the object's external interface.

To declare a private method, put a `#` sign in front of its name. Such methods can only be called from inside the `class` declaration that defines them.

```
class SecretiveObject {
  #getSecret() {
    return "I ate all the plums";
  }
  interrogate() {
    let shallISayIt = this.#getSecret();
    return "never";
  }
}
```

When a class does not declare a constructor, it will automatically get an empty one.

If you try to call `#getSecret` from outside the class, you get an error. Its existence is entirely hidden inside the class declaration.

To use private instance properties, you must declare them. Regular properties can be created by just assigning to them, but private properties *must* be declared in the class declaration to be available at all.

This class implements an appliance for getting a random whole number below a given maximum number. It only has one public property: `getNumber`.

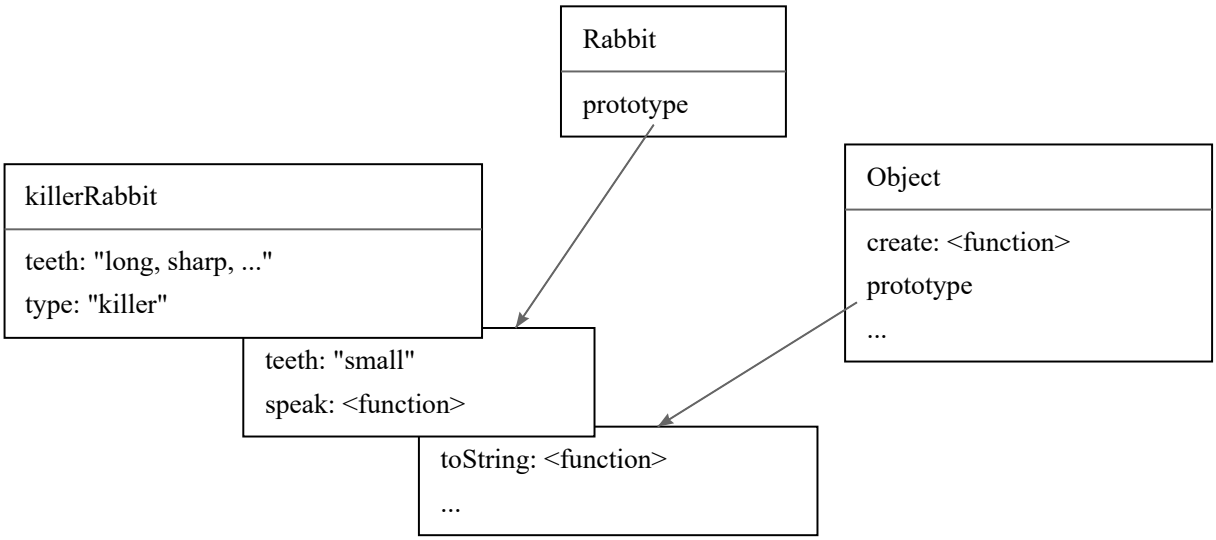
```
class RandomSource {
  #max;
  constructor(max) {
    this.#max = max;
  }
  getNumber() {
    return Math.floor(Math.random() * this.#max);
  }
}
```

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((new Rabbit("basic")).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 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, you can 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 `Object.hasOwn` function, which ignores the object's prototype.

```
console.log(Object.hasOwn({x: 1}, "x"));
// → true
console.log(Object.hasOwn({x: 1}, "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(killerRabbit));
// → a killer 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 will be able to work with it.

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

An example of a widely used interface is that of array-like objects which have a `length` property holding a number and numbered properties for each of their elements. Both arrays and strings support this interface, as do various other objects, some of which we'll see later in the chapters about the browser. Our implementation of `forEach` from Chapter 5 works on anything that provides this interface. In fact, so does `Array.prototype.forEach`.

```
Array.prototype.forEach.call({
  length: 2,
  0: "A",
  1: "B"
}, elt => console.log(elt));
// → A
// → B
```

Getters, setters, and statics

Interfaces often contain plain properties, not just methods. For example, `Map` objects have a `size` property that tells you how many keys are stored in them.

It is not 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 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 or properties that have `static` written before their name are stored on the constructor. For example, the `Temperature` class allows you to write `Temperature.fromFahrenheit(100)` to create a temperature using degrees Fahrenheit:

```
let boil = Temperature.fromFahrenheit(212);
console.log(boil.celsius);
// → 100
```

Symbols

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 our own objects! But before we can do that, we need to briefly take a look at the symbol type.

It is possible for multiple interfaces to use the same property name for different things. For example, on array-like objects, `length` refers to the number of elements in the collection. But an object interface describing a hiking route could use `length` to provide the length of the route in meters. It would not be possible for an object to conform to both these interfaces.

An object trying to be a route and array-like (maybe to enumerate its waypoints) is somewhat far-fetched, and this kind of problem isn't that common in practice. For things like the iteration protocol, though, the language designers needed a type of property that *really* doesn't conflict with any others. So in 2015, *symbols* were added to the language.

Most properties, including all those we have seen so far, are named with strings. But it is also possible to use symbols as property names. 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(killerRabbit[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 length = Symbol("length");
Array.prototype[length] = 0;

console.log([1, 2].length);
// → 2
console.log([1, 2][length]);
// → 0
```

It is possible to include symbol properties in object expressions and classes by using square brackets around the property name. That causes the expression between the brackets to be evaluated to produce the property name, analogous to the square bracket property access notation.

```
let myTrip = {
  length: 2,
  0: "Lankwitz",
  1: "Babelsberg",
  [length]: 21500
};
console.log(myTrip[length], myTrip.length);
// → 21500 2
```

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 similar to the linked list from the exercise in Chapter 4. We'll write the list as a class this time.

```
class List {
  constructor(value, rest) {
    this.value = value;
    this.rest = rest;
  }

  get length() {
    return 1 + (this.rest ? this.rest.length : 0);
  }

  static fromArray(array) {
    let result = null;
    for (let i = array.length - 1; i >= 0; i--) {
      result = new this(array[i], result);
```

```
      }
      return result;
    }
  }
```

Note that `this`, in a static method, points at the constructor of the class, not an instance—there is no instance around when a static method is called.

Iterating over a list should return all the list's elements from start to end. We'll write a separate class for the iterator.

```
class ListIterator {
  constructor(list) {
    this.list = list;
  }

  next() {
    if (this.list == null) {
      return {done: true};
    }
    let value = this.list.value;
    this.list = this.list.rest;
    return {value, done: false};
  }
}
```

The class tracks the progress of iterating through the list by updating its `list` property to move to the next list object whenever a value is returned and reports that it is done when that list is empty (null).

Let's set up the `List` 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.

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

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

```
let list = List.fromArray([1, 2, 3]);
for (let element of list) {
  console.log(element);
}
// → 1
// → 2
// → 3
```

The `...` syntax in array notation and function calls similarly works with any iterable object. For example, you can use `[...value]` to create an array containing the elements in an arbitrary iterable object.

```
console.log([..."PCI"]);
// → ["P", "C", "I"]
```

Inheritance

Imagine we need a list type much like the `List` class we saw before, but because we will be asking for its length all the time, we don't want it to have to scan through its `rest` every time. Instead, we want to store the length in every instance for efficient access.

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 `length` getter.

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

```
class LengthList extends List {
  #length;

  constructor(value, rest) {
    super(value, rest);
    this.#length = super.length;
  }

  get length() {
    return this.#length;
  }
```

```
}

console.log(LengthList.fromArray([1, 2, 3]).length);
// → 3
```

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 `LengthList` instance, the constructor calls the constructor of its superclass through the `super` keyword. This is necessary because if this new object is to behave (roughly) like a `List`, it is going to need the instance properties that lists have.

The constructor then stores the list's length in a private property. If we had written `this.length` there, the class's own getter would have been called, which doesn't work yet since `#length` hasn't been filled in yet. We can use `super.something` to call methods and getters on the superclass's prototype, which is often useful.

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 one another, 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 to make some types of programs more succinct, 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 LengthList(1, null) instanceof LengthList);
// → true
console.log(new LengthList(2, null) instanceof List);
// → true
console.log(new List(3, null) instanceof LengthList);
// → false
console.log([1] instanceof Array);
// → true
```

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

Summary

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. You can use private properties to hide a part of your object from the outside world.

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), that it saves to properties of the same name.

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

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

```
// Your code here.

console.log(new Vec(1, 2).plus(new Vec(2, 3)));
// → Vec{x: 3, y: 5}
console.log(new Vec(1, 2).minus(new Vec(2, 3)));
// → Vec{x: -1, y: -1}
console.log(new Vec(3, 4).length);
// → 5
```

▶ Display hints...

Groups

The standard JavaScript environment provides another data structure called `Set`. Like an instance of `Map`, a set holds a collection of values. Unlike `Map`, it does not associate other values with those—it just tracks which values are part

of the set. A value can be part of a set only once—adding it again doesn't have any effect.

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

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

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

```
class Group {
  // Your code here.
}

let group = Group.from([10, 20]);
console.log(group.has(10));
// → true
console.log(group.has(30));
// → false
group.add(10);
group.delete(10);
console.log(group.has(10));
// → false
```

▶ Display hints...

ITERABLE GROUPS

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

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

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

```
// Your code here (and the code from the previous exercise)

for (let value of Group.from(["a", "b", "c"])) {
  console.log(value);
}
// → a
// → b
// → c
```

▶ Display hints...