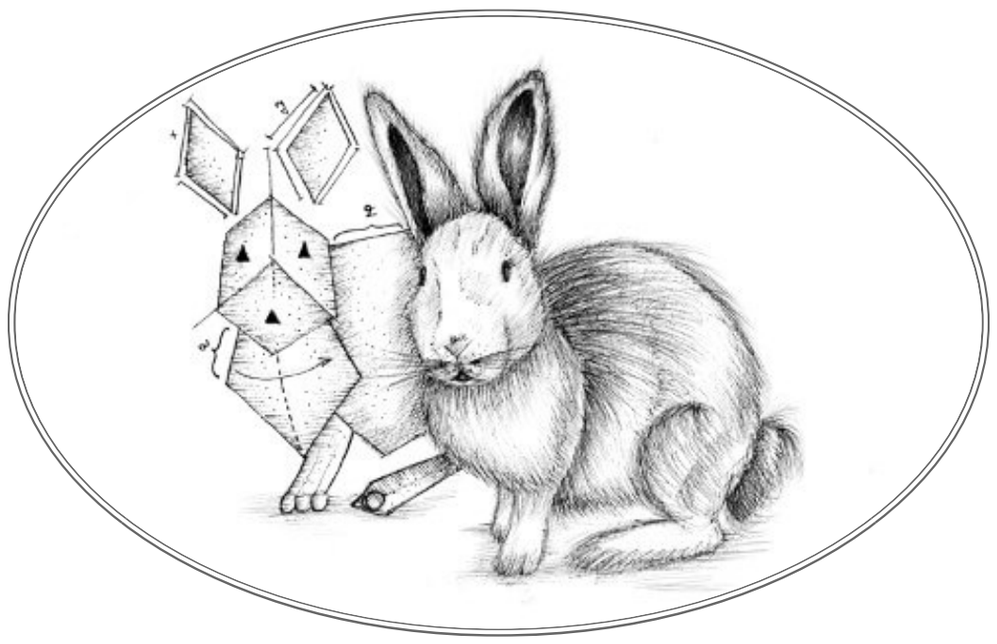
THE SECRET LIFE OF OBJECTS

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



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

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

ENCAPSULATION

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

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

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

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

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

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

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

METHODS

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

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

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

edit & run code by clicking it

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.

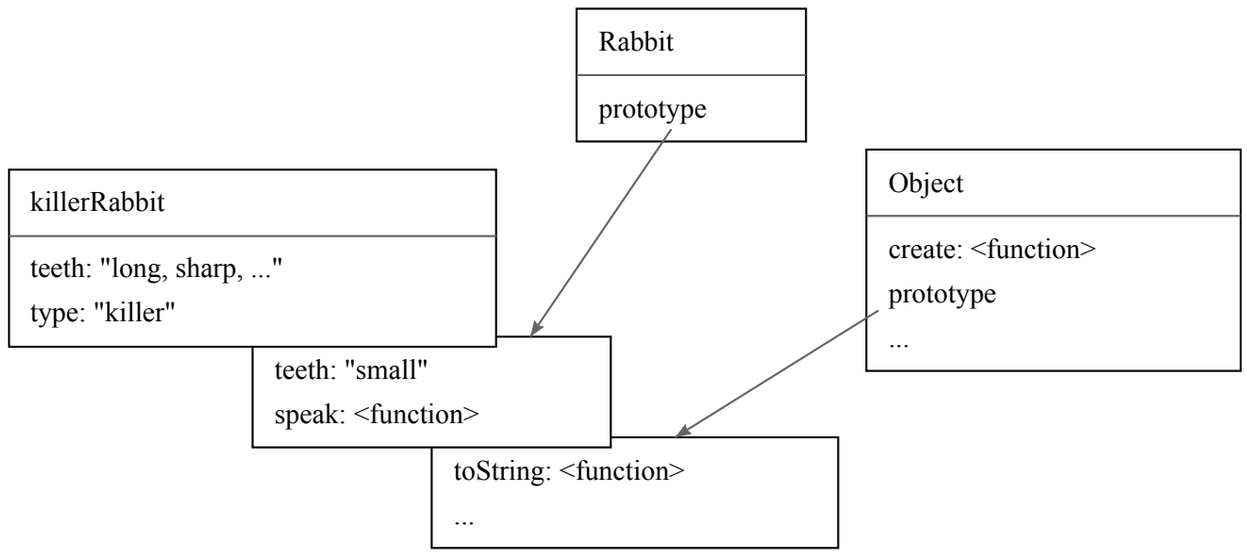
```
let object = new class { getWord() { return "hello"; } };
console.log(object.getWord());
// → hello
```

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 our own objects! But before we can do that, we need to know what symbols are.

Symbols

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

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

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

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

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

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

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

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

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

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

The iterator interface

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

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

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

We can directly use this interface ourselves.

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

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

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

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

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

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

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

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

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

```
  }

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

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

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

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

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

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

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

Getters, setters, and statics

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

It is not even necessary for such an object to compute and store such a property directly in the instance. Even properties that are accessed directly may hide a method call. Such methods are called *getters*, and they are defined by writing `get` in front of the method name in an object expression or class declaration.

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

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

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

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

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

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

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

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

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

Inheritance

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

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

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

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

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

```
    });
  }

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

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

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

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

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

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

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

The instanceof operator

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

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

The operator will see through inherited types, so a `SymmetricMatrix` is an

instance of `Matrix`. The operator can also be applied to standard constructors like `Array`. Almost every object is an instance of `Object`.

Summary

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

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

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

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

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

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

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

Exercises

A vector type

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

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

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

```
// Your code here.

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

» Display hints…

Groups

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

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

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

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

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

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

» Display hints…

Iterable groups

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

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

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

```
// Your code here (and the code from the previous exercise)
for (let value of Group.from(["a", "b", "c"])) {
  console.log(value);
}
// → a
// → b
// → c
```

» Display hints…

Borrowing a method

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

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

```
let map = {one: true, two: true, hasOwnProperty: true};

// Fix this call
console.log(map.hasOwnProperty("one"));
// → true
```

» Display hints…