# Chapter 14. Metaprogramming

This chapter covers a number of advanced JavaScript features that are not commonly used in day-to-day programming but that may be valuable to programmers writing reusable libraries and of interest to anyone who wants to tinker with the details about how JavaScript objects behave.

Many of the features described here can loosely be described as "metaprogramming": if regular programming is writing code to manipulate data, then metaprogramming is writing code to manipulate other code. In a dynamic language like JavaScript, the lines between programming and metaprogramming are blurry—even the simple ability to iterate over the properties of an object with a `for/in` loop might be considered "meta" by programmers accustomed to more static languages.

The metaprogramming topics covered in this chapter include:

- §14.1 Controlling the enumerability, deleteability, and configurability of object properties
- §14.2 Controlling the extensibility of objects, and creating "sealed" and "frozen" objects
- §14.3 Querying and setting the prototypes of objects
- §14.4 Fine-tuning the behavior of your types with well-known Symbols
- §14.5 Creating DSLs (domain-specific languages) with template tag functions
- §14.6 Probing objects with `reflect` methods
- §14.7 Controlling object behavior with Proxy

## 14.1 Property Attributes

The properties of a JavaScript object have names and values, of course, but each property also has three associated attributes that specify how that property behaves and what you can do with it:

- The *writable* attribute specifies whether or not the value of a property can change.
- The *enumerable* attribute specifies whether the property is enumerated by the `for/in` loop and the `Object.keys()` method.

- The *configurable* attribute specifies whether a property can be deleted and also whether the property's attributes can be changed.

Properties defined in object literals or by ordinary assignment to an object are writable, enumerable, and configurable. But many of the properties defined by the JavaScript standard library are not.

This section explains the API for querying and setting property attributes. This API is particularly important to library authors because:

- It allows them to add methods to prototype objects and make them non-enumerable, like built-in methods.
- It allows them to "lock down" their objects, defining properties that cannot be changed or deleted.

Recall from §6.10.6 that, while "data properties" have a value, "accessor properties" have a getter and/or a setter method instead. For the purposes of this section, we are going to consider the getter and setter methods of an accessor property to be property attributes. Following this logic, we'll even say that the value of a data property is an attribute as well. Thus, we can say that a property has a name and four attributes. The four attributes of a data property are *value*, *writable*, *enumerable*, and *configurable*. Accessor properties don't have a *value* attribute or a *writable* attribute: their writability is determined by the presence or absence of a setter. So the four attributes of an accessor property are *get*, *set*, *enumerable*, and *configurable*.

The JavaScript methods for querying and setting the attributes of a property use an object called a *property descriptor* to represent the set of four attributes. A property descriptor object has properties with the same names as the attributes of the property it describes. Thus, the property descriptor object of a data property has properties named `value`, `writable`, `enumerable`, and `configurable`. And the descriptor for an accessor property has `get` and `set` properties instead of `value` and `writable`. The `writable`, `enumerable`, and `configurable` properties are boolean values, and the `get` and `set` properties are function values.

To obtain the property descriptor for a named property of a specified object, call `Object.getOwnPropertyDescriptor()`:

```
// Returns {value: 1, writable:true, enumerable:true, configurable:true}
Object.getOwnPropertyDescriptor({x: 1}, "x");

// Here is an object with a read-only accessor property
const random = {
    get octet() { return Math.floor(Math.random()*256); },
};

// Returns { get: /*func*/, set:undefined, enumerable:true, configurable:tru
Object.getOwnPropertyDescriptor(random, "octet");

// Returns undefined for inherited properties and properties that don't exis
Object.getOwnPropertyDescriptor({}, "x")        // => undefined; no such pro
Object.getOwnPropertyDescriptor({}, "toString") // => undefined; inherited
```

As its name implies, `Object.getOwnPropertyDescriptor()` works only for own properties. To query the attributes of inherited properties, you must explicitly traverse the prototype chain. (See `Object.getPrototypeOf()` in §14.3); see also the similar `Reflect.getOwnPropertyDescriptor()` function in §14.6.)

To set the attributes of a property or to create a new property with the specified attributes, call `Object.defineProperty()`, passing the object to be modified, the name of the property to be created or altered, and the property descriptor object:

```
let o = {};  // Start with no properties at all
// Add a non-enumerable data property x with value 1.
Object.defineProperty(o, "x", {
    value: 1,
    writable: true,
    enumerable: false,
    configurable: true
});

// Check that the property is there but is non-enumerable
o.x            // => 1
Object.keys(o) // => []

// Now modify the property x so that it is read-only
Object.defineProperty(o, "x", { writable: false });

// Try to change the value of the property
o.x = 2;      // Fails silently or throws TypeError in strict mode
o.x           // => 1
```

```
// The property is still configurable, so we can change its value like this:
Object.defineProperty(o, "x", { value: 2 });
o.x           // => 2

// Now change x from a data property to an accessor property
Object.defineProperty(o, "x", { get: function() { return 0; } });
o.x           // => 0
```

The property descriptor you pass to `Object.defineProperty()` does not have to include all four attributes. If you're creating a new property, then omitted attributes are taken to be `false` or `undefined`. If you're modifying an existing property, then the attributes you omit are simply left unchanged. Note that this method alters an existing own property or creates a new own property, but it will not alter an inherited property. See also the very similar function `Reflect.defineProperty()` in §14.6.

If you want to create or modify more than one property at a time, use `Object.defineProperties()`. The first argument is the object that is to be modified. The second argument is an object that maps the names of the properties to be created or modified to the property descriptors for those properties. For example:

```
let p = Object.defineProperties({}, {
    x: { value: 1, writable: true, enumerable: true, configurable: true },
    y: { value: 1, writable: true, enumerable: true, configurable: true },
    r: {
        get() { return Math.sqrt(this.x*this.x + this.y*this.y); },
        enumerable: true,
        configurable: true
    }
});
p.r  // => Math.SQRT2
```

This code starts with an empty object, then adds two data properties and one read-only accessor property to it. It relies on the fact that `Object.defineProperties()` returns the modified object (as does `Object.defineProperty()`).

The `Object.create()` method was introduced in §6.2. We learned there that the first argument to that method is the prototype object for the newly created object. This method also accepts a second optional argument, which is the same as the second argument to `Object.defineProperties()`. If you pass a set of property descriptors to `Object.create()`, then they are used to add properties to the newly created object.

`Object.defineProperty()` and `Object.defineProperties()` throw TypeError if the attempt to create or modify a property is not allowed. This happens if you attempt to add a new property to a non-extensible (see §14.2) object. The other reasons that these methods might throw TypeError have to do with the attributes themselves. The *writable* attribute governs attempts to change the *value* attribute. And the *configurable* attribute governs attempts to change the other attributes (and also specifies whether a property can be deleted). The rules are not completely straightforward, however. It is possible to change the value of a nonwritable property if that property is configurable, for example. Also, it is possible to change a property from writable to nonwritable even if that property is nonconfigurable. Here are the complete rules. Calls to `Object.defineProperty()` or `Object.defineProperties()` that attempt to violate them throw a TypeError:

- If an object is not extensible, you can edit its existing own properties, but you cannot add new properties to it.
- If a property is not configurable, you cannot change its configurable or enumerable attributes.
- If an accessor property is not configurable, you cannot change its getter or setter method, and you cannot change it to a data property.
- If a data property is not configurable, you cannot change it to an accessor property.
- If a data property is not configurable, you cannot change its *writable* attribute from `false` to `true`, but you can change it from `true` to `false`.
- If a data property is not configurable and not writable, you cannot change its value. You can change the value of a property that is configurable but nonwritable, however (because that would be the same as making it writable, then changing the value, then converting it back to nonwritable).

§6.7 described the `Object.assign()` function that copies property values from one or more source objects into a target object. `Object.assign()` only copies enumerable properties, and property values, not property attributes. This is normally what we want, but it does mean, for example, that if one of the source objects has an accessor property, it is the value returned by the getter function that is copied to the target object, not the getter function itself. Example 14-1 demonstrates how we can use `Object.getOwnPropertyDescriptor()` and `Object.defineProperty()` to create a variant of `Object.assign()` that copies entire property descriptors rather than just copying property values.

**Example 14-1. Copying properties and their attributes from one object to another**

```
/*
 * Define a new Object.assignDescriptors() function that works like
 * Object.assign() except that it copies property descriptors from
 * source objects into the target object instead of just copying
 * property values. This function copies all own properties, both
 * enumerable and non-enumerable. And because it copies descriptors,
 * it copies getter functions from source objects and overwrites setter
 * functions in the target object rather than invoking those getters and
 * setters.
 *
 * Object.assignDescriptors() propagates any TypeErrors thrown by
 * Object.defineProperty(). This can occur if the target object is sealed
 * or frozen or if any of the source properties try to change an existing
 * non-configurable property on the target object.
 *
 * Note that the assignDescriptors property is added to Object with
 * Object.defineProperty() so that the new function can be created as
 * a non-enumerable property like Object.assign().
 */
Object.defineProperty(Object, "assignDescriptors", {
    // Match the attributes of Object.assign()
    writable: true,
    enumerable: false,
    configurable: true,
    // The function that is the value of the assignDescriptors property.
    value: function(target, ...sources) {
        for(let source of sources) {
            for(let name of Object.getOwnPropertyNames(source)) {
                let desc = Object.getOwnPropertyDescriptor(source, name);
                Object.defineProperty(target, name, desc);
            }

            for(let symbol of Object.getOwnPropertySymbols(source)) {
                let desc = Object.getOwnPropertyDescriptor(source, symbol);
                Object.defineProperty(target, symbol, desc);
            }
        }
        return target;
    }
});

let o = {c: 1, get count() {return this.c++;}}; // Define object with getter
let p = Object.assign({}, o);                   // Copy the property values
let q = Object.assignDescriptors({}, o);        // Copy the property descrip
p.count   // => 1: This is now just a data property so
p.count   // => 1: ...the counter does not increment.
q.count   // => 2: Incremented once when we copied it the first time,
q.count   // => 3: ...but we copied the getter method so it increments.
```

## 14.2 Object Extensibility

The *extensible* attribute of an object specifies whether new properties can be added to the object or not. Ordinary JavaScript objects are extensible by default, but you can change that with the functions described in this section.

To determine whether an object is extensible, pass it to `Object.isExtensible()`. To make an object non-extensible, pass it to `Object.preventExtensions()`. Once you have done this, any attempt to add a new property to the object will throw a TypeError in strict mode and simply fail silently without an error in non-strict mode. In addition, attempting to change the prototype (see §14.3) of a non-extensible object will always throw a TypeError.

Note that there is no way to make an object extensible again once you have made it non-extensible. Also note that calling `Object.preventExtensions()` only affects the extensibility of the object itself. If new properties are added to the prototype of a non-extensible object, the non-extensible object will inherit those new properties.

Two similar functions, `Reflect.isExtensible()` and `Reflect.preventExtensions()`, are described in §14.6.

The purpose of the *extensible* attribute is to be able to “lock down” objects into a known state and prevent outside tampering. The *extensible* attribute of objects is often used in conjunction with the *configurable* and *writable* attributes of properties, and JavaScript defines functions that make it easy to set these attributes together:

- `Object.seal()` works like `Object.preventExtensions()`, but in addition to making the object non-extensible, it also makes all of the own properties of that object nonconfigurable. This means that new properties cannot be added to the object, and existing properties cannot be deleted or configured. Existing properties that are writable can still be set, however. There is no way to unseal a sealed object. You can use `Object.isSealed()` to determine whether an object is sealed.
- `Object.freeze()` locks objects down even more tightly. In addition to making the object non-extensible and its properties nonconfigurable, it also makes all of the object's own data properties read-only. (If the object has accessor properties with setter methods, these are not affected and can still be invoked by assignment to the property.) Use `Object.isFrozen()` to determine if an object is frozen.

It is important to understand that `Object.seal()` and `Object.freeze()` affect only the object they are passed: they have no effect on the prototype of that object. If you want to thoroughly lock down an object, you probably need to seal or freeze the objects in the prototype chain as well.

`Object.preventExtensions()`, `Object.seal()`, and `Object.freeze()` all return the object that they are passed, which means that you can use them in nested function invocations:

```
// Create a sealed object with a frozen prototype and a non-enumerable prope
let o = Object.seal(Object.create(Object.freeze({x: 1}),
                                  {y: {value: 2, writable: true}}));
```

If you are writing a JavaScript library that passes objects to callback functions written by the users of your library, you might use `Object.freeze()` on those objects to prevent the user's code from modifying them. This is easy and convenient to do, but there are trade-offs: frozen objects can interfere with common JavaScript testing strategies, for example.

## 14.3 The prototype Attribute

An object's `prototype` attribute specifies the object from which it inherits its properties. (Review §6.2.3 and §6.3.2 for more on prototypes and property inheritance.) This is such an important attribute that we usually simply say "the prototype of `o`" rather than "the `prototype` attribute of `o`." Remember also that when `prototype` appears in code font, it refers to an ordinary object property, not to the `prototype` attribute: Chapter 9 explained that the `prototype` property of a constructor function specifies the `prototype` attribute of the objects created with that constructor.

The `prototype` attribute is set when an object is created. Objects created from object literals use `Object.prototype` as their prototype. Objects created with `new` use the value of the `prototype` property of their constructor function as their prototype. And objects created with `Object.create()` use the first argument to that function (which may be `null`) as their prototype.

You can query the prototype of any object by passing that object to `Object.getPrototypeOf()`:

```
Object.getPrototypeOf({})       // => Object.prototype
Object.getPrototypeOf([])       // => Array.prototype
Object.getPrototypeOf(()=>{})   // => Function.prototype
```

A very similar function, `Reflect.getPrototypeOf()`, is described in §14.6.

To determine whether one object is the prototype of (or is part of the prototype chain of) another object, use the `isPrototypeOf()` method:

```
let p = {x: 1};                    // Define a prototype object.
let o = Object.create(p);          // Create an object with that prototype.
p.isPrototypeOf(o)                 // => true: o inherits from p
Object.prototype.isPrototypeOf(p)  // => true: p inherits from Object.prototy
Object.prototype.isPrototypeOf(o)  // => true: o does too
```

Note that `isPrototypeOf()` performs a function similar to the `instanceof` operator (see §4.9.4).

The `prototype` attribute of an object is set when the object is created and normally remains fixed. You can, however, change the prototype of an object with `Object.setPrototypeOf()`:

```
let o = {x: 1};
let p = {y: 2};
Object.setPrototypeOf(o, p); // Set the prototype of o to p
o.y       // => 2: o now inherits the property y
let a = [1, 2, 3];
Object.setPrototypeOf(a, p); // Set the prototype of array a to p
a.join    // => undefined: a no longer has a join() method
```

There is generally no need to ever use `Object.setPrototypeOf()`. JavaScript implementations may make aggressive optimizations based on the assumption that the prototype of an object is fixed and unchanging. This means that if you ever call `Object.setPrototypeOf()`, any code that uses the altered objects may run much slower than it would normally.

A similar function, `Reflect.setPrototypeOf()`, is described in §14.6.

Some early browser implementations of JavaScript exposed the `prototype` attribute of an object through the `__proto__` property (written with two underscores at the start and end). This has long since been deprecated, but enough existing code on the web depends on `__proto__` that the ECMAScript standard mandates it for all JavaScript implementations that run in web browsers. (Node supports it, too, though the standard does not require it for Node.) In modern JavaScript, `__proto__` is readable and writeable, and you can (though you shouldn't) use it as an alternative to `Object.getPrototypeOf()` and `Object.setPrototypeOf()`. One interesting use of `__proto__`, however, is to define the prototype of an object literal:

```
let p = {z: 3};
let o = {
    x: 1,
    y: 2,
    __proto__: p
};
o.z  // => 3: o inherits from p
```

## 14.4 Well-Known Symbols

The Symbol type was added to JavaScript in ES6, and one of the primary reasons for doing so was to safely add extensions to the language without breaking compatibility with code already deployed on the web. We saw an example of this in Chapter 12, where we learned that you can make a class iterable by implementing a method whose "name" is the Symbol `Symbol.iterator`.

`Symbol.iterator` is the best-known example of the "well-known Symbols." These are a set of Symbol values stored as properties of the `Symbol()` factory function that are used to allow JavaScript code to control certain low-level behaviors of objects and classes. The subsections that follow describe each of these well-known Symbols and explain how they can be used.

### 14.4.1 Symbol.iterator and Symbol.asyncIterator

The `Symbol.iterator` and `Symbol.asyncIterator` Symbols allow objects or classes to make themselves iterable or asynchronously iterable. They were covered in detail in Chapter 12 and §13.4.2, respectively, and are mentioned again here only for completeness.

### 14.4.2 Symbol.hasInstance

When the `instanceof` operator was described in §4.9.4, we said that the righthand side must be a constructor function and that the expression `o instanceof f` was evaluated by looking for the value `f.prototype` within the prototype chain of `o`. That is still true, but in ES6 and beyond, `Symbol.hasInstance` provides an alternative. In ES6, if the righthand side of `instanceof` is any object with a `[Symbol.hasInstance]` method, then that method is invoked with the lefthand side value as its argument, and the return value of the method, converted to a boolean, becomes the value of the `instanceof` operator. And, of course, if the value on the righthand side does not have a `[Symbol.hasInstance]` method but is a function, then the `instanceof` operator behaves in its ordinary way.

`Symbol.hasInstance` means that we can use the `instanceof` operator to do generic type checking with suitably defined pseudotype objects. For example:

```
// Define an object as a "type" we can use with instanceof
let uint8 = {
    [Symbol.hasInstance](x) {
        return Number.isInteger(x) && x >= 0 && x <= 255;
    }
};
128 instanceof uint8      // => true
256 instanceof uint8      // => false: too big
Math.PI instanceof uint8 // => false: not an integer
```

Note that this example is clever but confusing because it uses a nonclass object where a class would normally be expected. It would be just as easy —and clearer to readers of your code—to write a `isUint8()` function instead of relying on this `Symbol.hasInstance` behavior.

### 14.4.3 Symbol.toStringTag

If you invoke the `toString()` method of a basic JavaScript object, you get the string "[object Object]":

```
{}.toString()  // => "[object Object]"
```

If you invoke this same `Object.prototype.toString()` function as a method of instances of built-in types, you get some interesting results:

```
Object.prototype.toString.call([])      // => "[object Array]"
Object.prototype.toString.call(/./)     // => "[object RegExp]"
Object.prototype.toString.call(()=>{}) // => "[object Function]"
Object.prototype.toString.call("")      // => "[object String]"
Object.prototype.toString.call(0)       // => "[object Number]"
Object.prototype.toString.call(false)   // => "[object Boolean]"
```

It turns out that you can use this `Object.prototype.toString().call()` technique with any JavaScript value to obtain the "class attribute" of an object that contains type information that is not otherwise available. The following `classof()` function is arguably more useful than the `typeof` operator, which makes no distinction between types of objects:

```
function classof(o) {
    return Object.prototype.toString.call(o).slice(8,-1);
}

classof(null)       // => "Null"
classof(undefined)  // => "Undefined"
classof(1)          // => "Number"
classof(10n**100n)  // => "BigInt"
classof("")         // => "String"
classof(false)      // => "Boolean"
classof(Symbol())   // => "Symbol"
classof({})         // => "Object"
classof([])         // => "Array"
classof(/./)        // => "RegExp"
classof(()=>{})     // => "Function"
classof(new Map())  // => "Map"
classof(new Set())  // => "Set"
classof(new Date()) // => "Date"
```

Prior to ES6, this special behavior of the `Object.prototype.toString()` method was available only to instances of built-in types, and if you called this `classof()` function on an instance of a class you had defined yourself, it would simply return "Object". In ES6, however, `Object.prototype.toString()` looks for a property with the symbolic name `Symbol.toStringTag` on its argument, and if such a property exists, it uses the property value in its output. This means that if you define a class of your own, you can easily make it work with functions like `classof()`:

```
class Range {
    get [Symbol.toStringTag]() { return "Range"; }
    // the rest of this class is omitted here
}
let r = new Range(1, 10);
Object.prototype.toString.call(r)   // => "[object Range]"
classof(r)                          // => "Range"
```

## 14.4.4 Symbol.species

Prior to ES6, JavaScript did not provide any real way to create robust subclasses of built-in classes like Array. In ES6, however, you can extend any built-in class simply by using the `class` and `extends` keywords. §9.5.2 demonstrated that with this simple subclass of Array:

```
// A trivial Array subclass that adds getters for the first and last element
class EZArray extends Array {
    get first() { return this[0]; }
    get last() { return this[this.length-1]; }
}
```

```
let e = new EZArray(1,2,3);
let f = e.map(x => x * x);
e.last  // => 3: the last element of EZArray e
f.last  // => 9: f is also an EZArray with a last property
```

Array defines methods `concat()`, `filter()`, `map()`, `slice()`, and `splice()`, which return arrays. When we create an array subclass like EZArray that inherits these methods, should the inherited method return instances of Array or instances of EZArray? Good arguments can be made for either choice, but the ES6 specification says that (by default) the five array-returning methods will return instances of the subclass.

Here's how it works:

- In ES6 and later, the `Array()` constructor has a property with the symbolic name `Symbol.species`. (Note that this Symbol is used as the name of a property of the constructor function. Most of the other well-known Symbols described here are used as the name of methods of a prototype object.)
- When we create a subclass with `extends`, the resulting subclass constructor inherits properties from the superclass constructor. (This is in addition to the normal kind of inheritance, where instances of the subclass inherit methods of the superclass.) This means that the constructor for every subclass of Array also has an inherited property with name `Symbol.species`. (Or a subclass can define its own property with this name, if it wants.)
- Methods like `map()` and `slice()` that create and return new arrays are tweaked slightly in ES6 and later. Instead of just creating a regular Array, they (in effect) invoke `new this.constructor[Symbol.species]()` to create the new array.

Now here's the interesting part. Suppose that `Array[Symbol.species]` was just a regular data property, defined like this:

```
Array[Symbol.species] = Array;
```

In that case, then subclass constructors would inherit the `Array()` constructor as their "species," and invoking `map()` on an array subclass would return an instance of the superclass rather than an instance of the subclass. That is not how ES6 actually behaves, however. The reason is that `Array[Symbol.species]` is a read-only accessor property whose getter function simply returns `this`. Subclass constructors inherit this getter function, which means that by default, every subclass constructor is its own "species."

Sometimes this default behavior is not what you want, however. If you wanted the array-returning methods of EZArray to return regular Array objects, you just need to set `EZArray[Symbol.species]` to `Array`. But since the inherited property is a read-only accessor, you can't just set it with an assignment operator. You can use `defineProperty()`, however:

```
EZArray[Symbol.species] = Array; // Attempt to set a read-only property fail

// Instead we can use defineProperty():
Object.defineProperty(EZArray, Symbol.species, {value: Array});
```

The simplest option is probably to explicitly define your own `Symbol.species` getter when creating the subclass in the first place:

```
class EZArray extends Array {
    static get [Symbol.species]() { return Array; }
    get first() { return this[0]; }
    get last() { return this[this.length-1]; }
}

let e = new EZArray(1,2,3);
let f = e.map(x => x - 1);
e.last  // => 3
f.last  // => undefined: f is a regular array with no last getter
```

Creating useful subclasses of Array was the primary use case that motivated the introduction of `Symbol.species`, but it is not the only place that this well-known Symbol is used. Typed array classes use the Symbol in the same way that the Array class does. Similarly, the `slice()` method of ArrayBuffer looks at the `Symbol.species` property of `this.constructor` instead of simply creating a new ArrayBuffer. And Promise methods like `then()` that return new Promise objects create those objects via this species protocol as well. Finally, if you find yourself subclassing Map (for example) and defining methods that return new Map objects, you might want to use `Symbol.species` yourself for the benefit of subclasses of your subclass.

### 14.4.5 Symbol.isConcatSpreadable

The Array method `concat()` is one of the methods described in the previous section that uses `Symbol.species` to determine what constructor to use for the returned array. But `concat()` also uses `Symbol.isConcatSpreadable`. Recall from §7.8.3 that the `concat()` method of an array treats its `this` value and its array arguments differently than its nonarray arguments: nonarray arguments are simply appended to the new array, but the `this` array and any array arguments are flattened or "spread" so that the elements of the array are concatenated rather than the array argument itself.

Before ES6, `concat()` just used `Array.isArray()` to determine whether to treat a value as an array or not. In ES6, the algorithm is changed slightly: if the argument (or the `this` value) to `concat()` is an object and has a property with the symbolic name `Symbol.isConcatSpreadable`, then the boolean value of that property is used to determine whether the argument should be "spread." If no such property exists, then `Array.isArray()` is used as in previous versions of the language.

There are two cases when you might want to use this Symbol:

- If you create an Array-like (see §7.9) object and want it to behave like a real array when passed to `concat()`, you can simply add the symbolic property to your object:

```
let arraylike = {
    length: 1,
    0: 1,
    [Symbol.isConcatSpreadable]: true
};
[].concat(arraylike)  // => [1]: (would be [[1]] if not spread)
```

- Array subclasses are spreadable by default, so if you are defining an array subclass that you do not want to act like an array when used with `concat()`, then you can[1] add a getter like this to your subclass:

```
class NonSpreadableArray extends Array {
    get [Symbol.isConcatSpreadable]() { return false; }
}
```

```
let a = new NonSpreadableArray(1,2,3);
[].concat(a).length // => 1; (would be 3 elements long if a was spread)
```

### 14.4.6 Pattern-Matching Symbols

§11.3.2 documented the String methods that perform pattern-matching operations using a RegExp argument. In ES6 and later, these methods have been generalized to work with RegExp objects or any object that defines pattern-matching behavior via properties with symbolic names. For each of the string methods `match()`, `matchAll()`, `search()`, `replace()`, and `split()`, there is a corresponding well-known Symbol: `Symbol.match`, `Symbol.search`, and so on.

RegExps are a general and very powerful way to describe textual patterns, but they can be complicated and not well suited to fuzzy matching. With the generalized string methods, you can define your own pattern classes using the well-known Symbol methods to provide custom matching. For example, you could perform string comparisons using Intl.Collator (see §11.7.3) to ignore accents when matching. Or you could define a pattern class based on the *Soundex* algorithm to match words based on their approximate sounds or to loosely match strings up to a given Levenshtein distance.

In general, when you invoke one of these five String methods on a pattern object like this:

```
string.method(pattern, arg)
```

that invocation turns into an invocation of a symbolically named method on your pattern object:

```
pattern[symbol](string, arg)
```

As an example, consider the pattern-matching class in the next example, which implements pattern matching using the simple `*` and `?` wildcards that you are probably familar with from filesystems. This style of pattern matching dates back to the very early days of the Unix operating system, and the patterns are often called *globs*:

```
class Glob {
    constructor(glob) {
        this.glob = glob;

        // We implement glob matching using RegExp internally.
        // ? matches any one character except /, and * matches zero or more
        // of those characters. We use capturing groups around each.
        let regexpText = glob.replace("?", "([^/])").replace("*", "([^/]*)")

        // We use the u flag to get Unicode-aware matching.
        // Globs are intended to match entire strings, so we use the ^ and $
        // anchors and do not implement search() or matchAll() since they
        // are not useful with patterns like this.
        this.regexp = new RegExp(`^${regexpText}$`, "u");
    }

    toString() { return this.glob; }

    [Symbol.search](s) { return s.search(this.regexp); }
    [Symbol.match](s)  { return s.match(this.regexp); }
    [Symbol.replace](s, replacement) {
        return s.replace(this.regexp, replacement);
    }
}

let pattern = new Glob("docs/*.txt");
"docs/js.txt".search(pattern)   // => 0: matches at character 0
"docs/js.htm".search(pattern)   // => -1: does not match
let match = "docs/js.txt".match(pattern);
match[0]     // => "docs/js.txt"
match[1]     // => "js"
match.index  // => 0
"docs/js.txt".replace(pattern, "web/$1.htm")  // => "web/js.htm"
```

### 14.4.7 Symbol.toPrimitive

§3.9.3 explained that JavaScript has three slightly different algorithms for converting objects to primitive values. Loosely speaking, for conversions where a string value is expected or preferred, JavaScript invokes an object's `toString()` method first and falls back on the `valueOf()` method if `toString()` is not defined or does not return a primitive value. For conversions where a numeric value is preferred, JavaScript tries the `valueOf()` method first and falls back on `toString()` if `valueOf()` is not defined or if it does not return a primitive value. And finally, in cases where there is no preference, it lets the class decide how to do the conversion. Date objects convert using `toString()` first, and all other types try `valueOf()` first.

In ES6, the well-known Symbol `Symbol.toPrimitive` allows you to override this default object-to-primitive behavior and gives you complete control over how instances of your own classes will be converted to primitive values. To do this, define a method with this symbolic name. The method must return a primitive value that somehow represents the object. The method you define will be invoked with a single string argument that tells you what kind of conversion JavaScript is trying to do on your object:

- If the argument is `"string"`, it means that JavaScript is doing the conversion in a context where it would expect or prefer (but not require) a string. This happens when you interpolate the object into a template literal, for example.
- If the argument is `"number"`, it means that JavaScript is doing the conversion in a context where it would expect or prefer (but not require) a numeric value. This happens when you use the object with a `<` or `>` operator or with arithmetic operators like `-` and `*`.
- If the argument is `"default"`, it means that JavaScript is converting your object in a context where either a numeric or string value could work. This happens with the `+`, `==`, and `!=` operators.

Many classes can ignore the argument and simply return the same primitive value in all cases. If you want instances of your class to be comparable and sortable with `<` and `>`, then that is a good reason to define a `[Symbol.toPrimitive]` method.

### 14.4.8 Symbol.unscopables

The final well-known Symbol that we'll cover here is an obscure one that was introduced as a workaround for compatibility issues caused by the deprecated `with` statement. Recall that the `with` statement takes an object and executes its statement body as if it were in a scope where the properties of that object were variables. This caused compatibility problems when new methods were added to the Array class, and it broke some existing code. `Symbol.unscopables` is the result. In ES6 and later, the `with` statement has been slightly modified. When used with an object `o`, a `with` statement computes `Object.keys(o[Symbol.unscopables]||{})` and ignores properties whose names are in the resulting array when creating the simulated scope in which to execute its body. ES6 uses this to add new methods to `Array.prototype` without breaking existing code on the web. This means that you can find a list of the newest Array methods by evaluating:

```
let newArrayMethods = Object.keys(Array.prototype[Symbol.unscopables]);
```

## 14.5 Template Tags

Strings within backticks are known as "template literals" and were covered in §3.3.4. When an expression whose value is a function is followed by a template literal, it turns into a function invocation, and we call it a "tagged template literal." Defining a new tag function for use with tagged template literals can be thought of as metaprogramming, because tagged templates are often used to define DSLs—domain-specific languages—and defining a new tag function is like adding new syntax to JavaScript. Tagged template literals have been adopted by a number of frontend JavaScript packages. The GraphQL query language uses a ``gql`` `` tag function to allow queries to be embedded within JavaScript code. And the Emotion library uses a ``css`` `` tag function to enable CSS styles to be embedded in JavaScript. This section demonstrates how to write your own tag functions like these.

There is nothing special about tag functions: they are ordinary JavaScript functions, and no special syntax is required to define them. When a function expression is followed by a template literal, the function is invoked. The first argument is an array of strings, and this is followed by zero or more additional arguments, which can have values of any type.

The number of arguments depends on the number of values that are interpolated into the template literal. If the template literal is simply a constant string with no interpolations, then the tag function will be called with an array of that one string and no additional arguments. If the template literal includes one interpolated value, then the tag function is called with two arguments. The first is an array of two strings, and the second is the interpolated value. The strings in that initial array are the string to the left of the interpolated value and the string to its right, and either one of them may be the empty string. If the template literal includes two interpolated values, then the tag function is invoked with three arguments: an array of three strings and the two interpolated values. The three strings (any or all of which may be empty) are the text to the left of the first value, the text between the two values, and the text to the right of the second value. In the general case, if the template literal has `n` interpolated values, then the tag function will be invoked with `n+1` arguments. The first argument will be an array of `n+1` strings, and the remaining arguments are the `n` interpolated values, in the order that they appear in the template literal.

The value of a template literal is always a string. But the value of a tagged template literal is whatever value the tag function returns. This may be a string, but when the tag function is used to implement a DSL, the return value is typically a non-string data structure that is a parsed representation of the string.

As an example of a template tag function that returns a string, consider the following ``html`` `` template, which is useful when you want to safely interpolate values into a string of HTML. The tag performs HTML escaping on each of the values before using it to build the final string:

```
function html(strings, ...values) {
    // Convert each value to a string and escape special HTML characters
    let escaped = values.map(v => String(v)
                                  .replace("&", "&amp;")
                                  .replace("<", "&lt;")
                                  .replace(">", "&gt;")
                                  .replace('"', "&quot;")
                                  .replace("'", "&#39;"));

    // Return the concatenated strings and escaped values
    let result = strings[0];
    for(let i = 0; i < escaped.length; i++) {
        result += escaped[i] + strings[i+1];
    }
    return result;
}

let operator = "<";
html`<b>x ${operator} y</b>`             // => "<b>x &lt; y</b>"

let kind = "game", name = "D&D";
html`<div class="${kind}">${name}</div>` // =>'<div class="game">D&amp;D</di
```

For an example of a tag function that does not return a string but instead a parsed representation of a string, think back to the Glob pattern class defined in §14.4.6. Since the `Glob()` constructor takes a single string argument, we can define a tag function for creating new Glob objects:

```
function glob(strings, ...values) {
    // Assemble the strings and values into a single string
    let s = strings[0];
    for(let i = 0; i < values.length; i++) {
        s += values[i] + strings[i+1];
    }
    // Return a parsed representation of that string
    return new Glob(s);
}

let root = "/tmp";
let filePattern = glob`${root}/*.html`;  // A RegExp alternative
"/tmp/test.html".match(filePattern)[1]   // => "test"
```

One of the features mentioned in passing in §3.3.4 is the `String.raw``` tag function that returns a string in its "raw" form without interpreting any of the backslash escape sequences. This is implemented using a feature of tag function invocation that we have not discussed yet. When a tag function is invoked, we've seen that its first argument is an array of strings. But this array also has a property named `raw`, and the value of that property is another array of strings, with the same number of elements. The argument array includes strings that have had escape sequences interpreted as usual. And the raw array includes strings in which escape sequences are not interpreted. This obscure feature is important if you want to define a DSL with a grammar that uses backslashes. For example, if we wanted our `glob``` tag function to support pattern matching on Windows-style paths (which use backslashes instead of forward slashes) and we did not want users of the tag to have to double every backslash, we could rewrite that function to use `strings.raw[]` instead of `strings[]`. The downside, of course, would be that we could no longer use escapes like `\u` in our glob literals.

## 14.6 The Reflect API

The Reflect object is not a class; like the Math object, its properties simply define a collection of related functions. These functions, added in ES6, define an API for "reflecting upon" objects and their properties. There is little new functionality here: the Reflect object defines a convenient set of functions, all in a single namespace, that mimic the behavior of core language syntax and duplicate the features of various pre-existing Object functions.

Although the `Reflect` functions do not provide any new features, they do group the features together in one convenient API. And, importantly, the set of `Reflect` functions maps one-to-one with the set of Proxy handler methods that we'll learn about in §14.7.

The Reflect API consists of the following functions:

*`Reflect.apply(f, o, args)`*

This function invokes the function `f` as a method of `o` (or invokes it as a function with no `this` value if `o` is `null`) and passes the values in the `args` array as arguments. It is equivalent to `f.apply(o, args)`.

*`Reflect.construct(c, args, newTarget)`*

This function invokes the constructor `c` as if the `new` keyword had been used and passes the elements of the array `args` as arguments. If the optional `newTarget` argument is specified, it is used as the value of `new.target` within the constructor invocation. If not specified, then the `new.target` value will be `c`.

*`Reflect.defineProperty(o, name, descriptor)`*

This function defines a property on the object `o`, using `name` (a string or symbol) as the name of the property. The Descriptor object should define the value (or getter and/or setter) and attributes of the property. `Reflect.defineProperty()` is very similar to `Object.defineProperty()` but returns `true` on success and `false` on failures. (`Object.defineProperty()` returns `o` on success and throws TypeError on failure.)

*`Reflect.deleteProperty(o, name)`*

This function deletes the property with the specified string or symbolic name from the object `o`, returning `true` if successful (or if no such property existed) and `false` if the property could not be deleted. Calling this function is similar to writing `delete o[name]`.

*`Reflect.get(o, name, receiver)`*

This function returns the value of the property of `o` with the specified name (a string or symbol). If the property is an accessor method with a getter, and if the optional `receiver` argument is specified, then the getter function is called as a method of `receiver` instead of as a method of `o`. Calling this function is similar to evaluating `o[name]`.

*`Reflect.getOwnPropertyDescriptor(o, name)`*

This function returns a property descriptor object that describes the attributes of the property named `name` of the object `o`, or returns `undefined` if no such property exists. This function is nearly identical to `Object.getOwnPropertyDescriptor()`, except that the Reflect API version of the function requires that the first argument be an object and throws TypeError if it is not.

*`Reflect.getPrototypeOf(o)`*

This function returns the prototype of object `o` or `null` if the object has no prototype. It throws a TypeError if `o` is a primitive value instead of an object. This function is almost identical to `Object.getPrototypeOf()` except that `Object.getPrototypeOf()` only throws a TypeError for `null` and `undefined` arguments and coerces other primitive values to their wrapper objects.

*`Reflect.has(o, name)`*

This function returns `true` if the object `o` has a property with the specified `name` (which must be a string or a symbol). Calling this function is similar to evaluating `name in o`.

*`Reflect.isExtensible(o)`*

This function returns `true` if the object `o` is extensible (§14.2) and `false` if it is not. It throws a TypeError if `o` is not an object. `Object.isExtensible()` is similar but simply returns `false` when passed an argument that is not an object.

*`Reflect.ownKeys(o)`*

This function returns an array of the names of the properties of the object `o` or throws a TypeError if `o` is not an object. The names in the returned array will be strings and/or symbols. Calling this function is similar to calling `Object.getOwnPropertyNames()` and `Object.getOwnPropertySymbols()` and combining their results.

*`Reflect.preventExtensions(o)`*

This function sets the *extensible* attribute (§14.2) of the object `o` to `false` and returns `true` to indicate success. It throws a TypeError if `o` is not an object. `Object.preventExtensions()` has the same effect but returns `o` instead of `true` and does not throw TypeError for nonobject arguments.

*`Reflect.set(o, name, value, receiver)`*

This function sets the property with the specified `name` of the object `o` to the specified `value`. It returns `true` on success and `false` on failure (which can happen if the property is read-only). It throws TypeError if `o` is not an object. If the specified property is an accessor property with a setter function, and if the optional `receiver` argument is passed, then the setter will be invoked as a method of `receiver` instead of being invoked as a method of `o`. Calling this function is usually the same as evaluating `o[name] = value`.

*`Reflect.setPrototypeOf(o, p)`*

This function sets the prototype of the object `o` to `p`, returning `true` on success and `false` on failure (which can occur if `o` is not extensible or if the operation would cause a circular prototype chain). It throws a TypeError if `o` is not an object or if `p` is neither an object nor `null`. `Object.setPrototypeOf()` is similar, but returns `o` on success and throws TypeError on failure. Remember that calling either of these functions is likely to make your code slower by disrupting JavaScript interpreter optimizations.

## 14.7 Proxy Objects

The Proxy class, available in ES6 and later, is JavaScript's most powerful metaprogramming feature. It allows us to write code that alters the fundamental behavior of JavaScript objects. The Reflect API described in §14.6 is a set of functions that gives us direct access to a set of fundamental operations on JavaScript objects. What the Proxy class does is allows us a way to implement those fundamental operations ourselves and create objects that behave in ways that are not possible for ordinary objects.

When we create a Proxy object, we specify two other objects, the target object and the handlers object:

```
let proxy = new Proxy(target, handlers);
```

The resulting Proxy object has no state or behavior of its own. Whenever you perform an operation on it (read a property, write a property, define a new property, look up the prototype, invoke it as a function), it dispatches those operations to the handlers object or to the target object.

The operations supported by Proxy objects are the same as those defined by the Reflect API. Suppose that `p` is a Proxy object and you write `delete p.x`. The `Reflect.deleteProperty()` function has the same behavior as the `delete` operator. And when you use the `delete` operator to delete a property of a Proxy object, it looks for a `deleteProperty()` method on the handlers object. If such a method exists, it invokes it. And if no such method exists, then the Proxy object performs the property deletion on the target object instead.

Proxies work this way for all of the fundamental operations: if an appropriate method exists on the handlers object, it invokes that method to perform the operation. (The method names and signatures are the same as those of the Reflect functions covered in §14.6.) And if that method does not exist on the handlers object, then the Proxy performs the fundamental operation on the target object. This means that a Proxy can obtain its behavior from the target object or from the handlers object. If the handlers object is empty, then the proxy is essentially a transparent wrapper around the target object:

```
let t = { x: 1, y: 2 };
let p = new Proxy(t, {});
p.x          // => 1
delete p.y   // => true: delete property y of the proxy
t.y          // => undefined: this deletes it in the target, too
p.z = 3;     // Defining a new property on the proxy
t.z          // => 3: defines the property on the target
```

This kind of transparent wrapper proxy is essentially equivalent to the underlying target object, which means that there really isn't a reason to use it instead of the wrapped object. Transparent wrappers can be useful, however, when created as "revocable proxies." Instead of creating a Proxy with the `Proxy()` constructor, you can use the `Proxy.revocable()` factory function. This function returns an object that includes a Proxy object and also a `revoke()` function. Once you call the `revoke()` function, the proxy immediately stops working:

```
function accessTheDatabase() { /* implementation omitted */ return 42; }
let {proxy, revoke} = Proxy.revocable(accessTheDatabase, {});

proxy()   // => 42: The proxy gives access to the underlying target functior
revoke(); // But that access can be turned off whenever we want
proxy();  // !TypeError: we can no longer call this function
```

Note that in addition to demonstrating revocable proxies, the preceding code also demonstrates that proxies can work with target functions as well as target objects. But the main point here is that revocable proxies are a building block for a kind of code isolation, and you might use them when dealing with untrusted third-party libraries, for example. If you have to pass a function to a library that you don't control, you can pass a revocable proxy instead and then revoke the proxy when you are finished with the library. This prevents the library from keeping a reference to your function and calling it at unexpected times. This kind of defensive programming is not typical in JavaScript programs, but the Proxy class at least makes it possible.

If we pass a non-empty handlers object to the `Proxy()` constructor, then we are no longer defining a transparent wrapper object and are instead implementing custom behavior for our proxy. With the right set of handlers, the underlying target object essentially becomes irrelevant.

In the following code, for example, is how we could implement an object that appears to have an infinite number of read-only properties, where the value of each property is the same as the name of the property:

```
// We use a Proxy to create an object that appears to have every
// possible property, with the value of each property equal to its name
let identity = new Proxy({}, {
    // Every property has its own name as its value
    get(o, name, target) { return name; },
    // Every property name is defined
    has(o, name) { return true; },
    // There are too many properties to enumerate, so we just throw
    ownKeys(o) { throw new RangeError("Infinite number of properties"); },
    // All properties exist and are not writable, configurable or enumerable
    getOwnPropertyDescriptor(o, name) {
        return {
            value: name,
            enumerable: false,
            writable: false,
            configurable: false
        };
    },
    // All properties are read-only so they can't be set
    set(o, name, value, target) { return false; },
    // All properties are non-configurable, so they can't be deleted
    deleteProperty(o, name) { return false; },
    // All properties exist and are non-configurable so we can't define more
    defineProperty(o, name, desc) { return false; },
    // In effect, this means that the object is not extensible
    isExtensible(o) { return false; },
    // All properties are already defined on this object, so it couldn't
    // inherit anything even if it did have a prototype object.
    getPrototypeOf(o) { return null; },
    // The object is not extensible, so we can't change the prototype
    setPrototypeOf(o, proto) { return false; },
});

identity.x                // => "x"
identity.toString         // => "toString"
identity[0]               // => "0"
identity.x = 1;           // Setting properties has no effect
identity.x                // => "x"
delete identity.x         // => false: can't delete properties either
identity.x                // => "x"
Object.keys(identity);    // !RangeError: can't list all the keys
for(let p of identity) ;  // !RangeError
```

Proxy objects can derive their behavior from the target object and from the handlers object, and the examples we have seen so far have used one object or the other. But it is typically more useful to define proxies that use both objects.

The following code, for example, uses Proxy to create a read-only wrapper for a target object. When code tries to read values from the object, those reads are forwarded to the target object normally. But if any code tries to modify the object or its properties, methods of the handler object throw a TypeError. A proxy like this might be helpful for writing tests: suppose you've written a function that takes an object argument and want to ensure that your function does not make any attempt to modify the input argument. If your test passes in a read-only wrapper object, then any writes will throw exceptions that cause the test to fail:

```
function readOnlyProxy(o) {
    function readonly() { throw new TypeError("Readonly"); }
    return new Proxy(o, {
        set: readonly,
        defineProperty: readonly,
        deleteProperty: readonly,
        setPrototypeOf: readonly,
    });
}

let o = { x: 1, y: 2 };     // Normal writable object
let p = readOnlyProxy(o);   // Readonly version of it
p.x                         // => 1: reading properties works
p.x = 2;                    // !TypeError: can't change properties
delete p.y;                 // !TypeError: can't delete properties
p.z = 3;                    // !TypeError: can't add properties
p.__proto__ = {};           // !TypeError: can't change the prototype
```

Another technique when writing proxies is to define handler methods that intercept operations on an object but still delegate the operations to the target object. The functions of the Reflect API (§14.6) have exactly the same signatures as the handler methods, so they make it easy to do that kind of delegation.

Here, for example, is a proxy that delegates all operations to the target object but uses handler methods to log the operations:

```
/*
 * Return a Proxy object that wraps o, delegating all operations to
 * that object after logging each operation. objname is a string that
 * will appear in the log messages to identify the object. If o has own
 * properties whose values are objects or functions, then if you query
 * the value of those properties, you'll get a loggingProxy back, so that
 * logging behavior of this proxy is "contagious".
 */
function loggingProxy(o, objname) {
    // Define handlers for our logging Proxy object.
    // Each handler logs a message and then delegates to the target object.
    const handlers = {
        // This handler is a special case because for own properties
        // whose value is an object or function, it returns a proxy rather
        // than returning the value itself.
        get(target, property, receiver) {
            // Log the get operation
            console.log(`Handler get(${objname},${property.toString()})`);

            // Use the Reflect API to get the property value
            let value = Reflect.get(target, property, receiver);

            // If the property is an own property of the target and
            // the value is an object or function then return a Proxy for it
            if (Reflect.ownKeys(target).includes(property) &&
                (typeof value === "object" || typeof value === "function"))
                return loggingProxy(value, `${objname}.${property.toString()
            }

            // Otherwise return the value unmodified.
            return value;
        },

        // There is nothing special about the following three methods:
        // they log the operation and delegate to the target object.
        // They are a special case simply so we can avoid logging the
        // receiver object which can cause infinite recursion.
        set(target, prop, value, receiver) {
            console.log(`Handler set(${objname},${prop.toString()},${value})
            return Reflect.set(target, prop, value, receiver);
        },
        apply(target, receiver, args) {
            console.log(`Handler ${objname}(${args})`);
            return Reflect.apply(target, receiver, args);
        },
        construct(target, args, receiver) {
            console.log(`Handler ${objname}(${args})`);
            return Reflect.construct(target, args, receiver);
        }
    };

    // We can automatically generate the rest of the handlers.
    // Metaprogramming FTW!
    Reflect.ownKeys(Reflect).forEach(handlerName => {
        if (!(handlerName in handlers)) {
            handlers[handlerName] = function(target, ...args) {
                // Log the operation
                console.log(`Handler ${handlerName}(${objname},${args})`);
```

```
                    // Delegate the operation
                    return Reflect[handlerName](target, ...args);
                };
            }
        });

        // Return a proxy for the object using these logging handlers
        return new Proxy(o, handlers);
    }
```

The `loggingProxy()` function defined earlier creates proxies that log all of the ways they are used. If you are trying to understand how an undocumented function uses the objects you pass it, using a logging proxy can help.

Consider the following examples, which result in some genuine insights about array iteration:

```
// Define an array of data and an object with a function property
let data = [10,20];
let methods = { square: x => x*x };

// Create logging proxies for the array and the object
let proxyData = loggingProxy(data, "data");
let proxyMethods = loggingProxy(methods, "methods");

// Suppose we want to understand how the Array.map() method works
data.map(methods.square)         // => [100, 400]

// First, let's try it with a logging Proxy array
proxyData.map(methods.square)    // => [100, 400]
// It produces this output:
// Handler get(data,map)
// Handler get(data,length)
// Handler get(data,constructor)
// Handler has(data,0)
// Handler get(data,0)
// Handler has(data,1)
// Handler get(data,1)

// Now lets try with a proxy methods object
data.map(proxyMethods.square)    // => [100, 400]
// Log output:
// Handler get(methods,square)
// Handler methods.square(10,0,10,20)
// Handler methods.square(20,1,10,20)
// Finally, let's use a logging proxy to learn about the iteration protocol
for(let x of proxyData) console.log("Datum", x);
// Log output:
// Handler get(data,Symbol(Symbol.iterator))
// Handler get(data,length)
// Handler get(data,0)
// Datum 10
// Handler get(data,length)
// Handler get(data,1)
// Datum 20
// Handler get(data,length)
```

From the first chunk of logging output, we learn that the `Array.map()` method explicitly checks for the existence of each array element (causing the `has()` handler to be invoked) before actually reading the element value (which triggers the `get()` handler). This is presumably so that it can distinguish nonexistent array elements from elements that exist but are undefined.

The second chunk of logging output might remind us that the function we pass to `Array.map()` is invoked with three arguments: the element's value, the element's index, and the array itself. (There is a problem in our logging output: the `Array.toString()` method does not include square brackets in its output, and the log messages would be clearer if they were included in the argument list `(10,0,[10,20])`.)

The third chunk of logging output shows us that the `for/of` loop works by looking for a method with symbolic name `[Symbol.iterator]`. It also demonstrates that the Array class's implementation of this iterator method is careful to check the array length at every iteration and does not assume that the array length remains constant during the iteration.

## 14.7.1 Proxy Invariants

The `readOnlyProxy()` function defined earlier creates Proxy objects that are effectively frozen: any attempt to alter a property value or property attribute or to add or remove properties will throw an exception. But as long as the target object is not frozen, we'll find that if we can query the proxy with `Reflect.isExtensible()` and `Reflect.getOwnPropertyDescriptor()`, and it will tell us that we should be able to set, add, and delete properties. So `readOnlyProxy()` creates objects in an inconsistent state. We could fix this by adding `isExtensible()` and `getOwnPropertyDescriptor()` handlers, or we can just live with this kind of minor inconsistency.

The Proxy handler API allows us to define objects with major inconsistencies, however, and in this case, the Proxy class itself will prevent us from creating Proxy objects that are inconsistent in a bad way. At the start of this section, we described proxies as objects with no behavior of their own because they simply forward all operations to the handlers object and the target object. But this is not entirely true: after forwarding an operation, the Proxy class performs some sanity checks on the result to ensure important JavaScript invariants are not being violated. If it detects a violation, the proxy will throw a TypeError instead of letting the operation proceed.

As an example, if you create a proxy for a non-extensible object, the proxy will throw a TypeError if the `isExtensible()` handler ever returns `true`:

```
let target = Object.preventExtensions({});
let proxy = new Proxy(target, { isExtensible() { return true; }});
Reflect.isExtensible(proxy);  // !TypeError: invariant violation
```

Relatedly, proxy objects for non-extensible targets may not have a `getPrototypeOf()` handler that returns anything other than the real prototype object of the target. Also, if the target object has nonwritable, nonconfigurable properties, then the Proxy class will throw a TypeError if the `get()` handler returns anything other than the actual value:

```
let target = Object.freeze({x: 1});
let proxy = new Proxy(target, { get() { return 99; }});
proxy.x;         // !TypeError: value returned by get() doesn't match target
```

Proxy enforces a number of additional invariants, almost all of them having to do with non-extensible target objects and nonconfigurable properties on the target object.

## 14.8 Summary

In this chapter, you have learned:

- JavaScript objects have an *extensible* attribute and object properties have *writable*, *enumerable*, and *configurable* attributes, as well as a value and a getter and/or setter attribute. You can use these attributes to “lock down” your objects in various ways, including creating “sealed” and “frozen” objects.
- JavaScript defines functions that allow you to traverse the prototype chain of an object and even to change the prototype of an object (though doing this can make your code slower).
- The properties of the `Symbol` object have values that are “well-known Symbols,” which you can use as property or method names for the objects and classes that you define. Doing so allows you to control how your object interacts with JavaScript language features and with the core library. For example, well-known Symbols allow you to make your classes iterable and control the string that is displayed when an instance is passed to `Object.prototype.toString()`. Prior to ES6, this kind of customization was available only to the native classes that were built in to an implementation.
- Tagged template literals are a function invocation syntax, and defining a new tag function is kind of like adding a new literal syntax to the language. Defining a tag function that parses its template string argument allows you to embed DSLs within JavaScript code. Tag functions also provide access to a raw, unescaped form of string literals where backslashes have no special meaning.
- The Proxy class and the related Reflect API allow low-level control over the fundamental behaviors of JavaScript objects. Proxy objects can be used as optionally revocable wrappers to improve code encapsulation, and they can also be used to implement nonstandard object behaviors (like some of the special case APIs defined by early web browsers).

[1] A bug in the V8 JavaScript engine means that this code does not work correctly in Node 13.