Chapter 12. Iterators and Generators

Iterable objects and their associated iterators are a feature of ES6 that we've seen several times throughout this book. Arrays (including TypedArrays) are iterable, as are strings and Set and Map objects. This means that the contents of these data structures can be iterated—looped over—with the `for/of` loop, as we saw in §5.4.4:

```
let sum = 0;
for(let i of [1,2,3]) { // Loop once for each of these values
    sum += i;
}
sum   // => 6
```

Iterators can also be used with the `...` operator to expand or "spread" an iterable object into an array initializer or function invocation, as we saw in §7.1.2:

```
let chars = [..."abcd"]; // chars == ["a", "b", "c", "d"]
let data = [1, 2, 3, 4, 5];
Math.max(...data)        // => 5
```

Iterators can be used with destructuring assignment:

```
let purpleHaze = Uint8Array.of(255, 0, 255, 128);
let [r, g, b, a] = purpleHaze; // a == 128
```

When you iterate a Map object, the returned values are `[key, value]` pairs, which work well with destructuring assignment in a `for/of` loop:

```
let m = new Map([["one", 1], ["two", 2]]);
for(let [k,v] of m) console.log(k, v); // Logs 'one 1' and 'two 2'
```

If you want to iterate just the keys or just the values rather than the pairs, you can use the `keys()` and `values()` methods:

```
[...m]            // => [["one", 1], ["two", 2]]: default iteration
[...m.entries()]  // => [["one", 1], ["two", 2]]: entries() method is the sa
```

```
[...m.keys()]      // => ["one", "two"]: keys() method iterates just map keys
[...m.values()]    // => [1, 2]: values() method iterates just map values
```

Finally, a number of built-in functions and constructors that are commonly used with Array objects are actually written (in ES6 and later) to accept arbitrary iterators instead. The `Set()` constructor is one such API:

```
// Strings are iterable, so the two sets are the same:
new Set("abc") // => new Set(["a", "b", "c"])
```

This chapter explains how iterators work and demonstrates how to create your own data structures that are iterable. After explaining basic iterators, this chapter covers generators, a powerful new feature of ES6 that is primarily used as a particularly easy way to create iterators.

12.1 How Iterators Work

The `for/of` loop and spread operator work seamlessly with iterable objects, but it is worth understanding what is actually happening to make the iteration work. There are three separate types that you need to understand to understand iteration in JavaScript. First, there are the *iterable* objects: these are types like Array, Set, and Map that can be iterated. Second, there is the *iterator* object itself, which performs the iteration. And third, there is the *iteration result* object that holds the result of each step of the iteration.

An *iterable* object is any object with a special iterator method that returns an iterator object. An *iterator* is any object with a `next()` method that returns an iteration result object. And an *iteration result* object is an object with properties named `value` and `done`. To iterate an iterable object, you first call its iterator method to get an iterator object. Then, you call the `next()` method of the iterator object repeatedly until the returned value has its `done` property set to `true`. The tricky thing about this is that the iterator method of an iterable object does not have a conventional name but uses the Symbol `Symbol.iterator` as its name. So a simple `for/of` loop over an iterable object `iterable` could also be written the hard way, like this:

```
let iterable = [99];
let iterator = iterable[Symbol.iterator]();
for(let result = iterator.next(); !result.done; result = iterator.next()) {
    console.log(result.value)  // result.value == 99
}
```

The iterator object of the built-in iterable datatypes is itself iterable. (That is, it has a method named `Symbol.iterator` that just returns itself.) This is occasionally useful in code like the following when you want to iterate though a "partially used" iterator:

```
let list = [1,2,3,4,5];
let iter = list[Symbol.iterator]();
let head = iter.next().value;  // head == 1
let tail = [...iter];          // tail == [2,3,4,5]
```

12.2 Implementing Iterable Objects

Iterable objects are so useful in ES6 that you should consider making your own datatypes iterable whenever they represent something that can be iterated. The Range classes shown in Examples 9-2 and 9-3 in Chapter 9 were iterable. Those classes used generator functions to make themselves iterable. We'll document generators later in this chapter, but first, we will implement the Range class one more time, making it iterable without relying on a generator.

In order to make a class iterable, you must implement a method whose name is the Symbol `Symbol.iterator`. That method must return an iterator object that has a `next()` method. And the `next()` method must return an iteration result object that has a `value` property and/or a boolean `done` property. Example 12-1 implements an iterable Range class and demonstrates how to create iterable, iterator, and iteration result objects.

Example 12-1. An iterable numeric Range class

```
/*
 * A Range object represents a range of numbers {x: from <= x <= to}
 * Range defines a has() method for testing whether a given number is a memb
 * of the range. Range is iterable and iterates all integers within the rang
 */
```

```
class Range {
    constructor (from, to) {
        this.from = from;
        this.to = to;
    }

    // Make a Range act like a Set of numbers
    has(x) { return typeof x === "number" && this.from <= x && x <= this.to;

    // Return string representation of the range using set notation
    toString() { return `{ x | ${this.from} ≤ x ≤ ${this.to} }`; }

    // Make a Range iterable by returning an iterator object.
    // Note that the name of this method is a special symbol, not a string.
    [Symbol.iterator]() {
        // Each iterator instance must iterate the range independently of
        // others. So we need a state variable to track our location in the
        // iteration. We start at the first integer >= from.
        let next = Math.ceil(this.from);  // This is the next value we retur
        let last = this.to;               // We won't return anything > this
        return {                          // This is the iterator object
            // This next() method is what makes this an iterator object.
            // It must return an iterator result object.
            next() {
                return (next <= last)    // If we haven't returned last value
                    ? { value: next++ } // return next value and increment i
                    : { done: true };   // otherwise indicate that we're don
            },

            // As a convenience, we make the iterator itself iterable.
            [Symbol.iterator]() { return this; }
        };
    }
}

for(let x of new Range(1,10)) console.log(x); // Logs numbers 1 to 10
[...new Range(-2,2)]                          // => [-2, -1, 0, 1, 2]
```

In addition to making your classes iterable, it can be quite useful to define functions that return iterable values. Consider these iterable-based alternatives to the `map()` and `filter()` methods of JavaScript arrays:

```
// Return an iterable object that iterates the result of applying f()
// to each value from the source iterable
function map(iterable, f) {
    let iterator = iterable[Symbol.iterator]();
```

```
        return {     // This object is both iterator and iterable
            [Symbol.iterator]() { return this; },
            next() {
                let v = iterator.next();
                if (v.done) {
                    return v;
                } else {
                    return { value: f(v.value) };
                }
            }
        };
    }

    // Map a range of integers to their squares and convert to an array
    [...map(new Range(1,4), x => x*x)]  // => [1, 4, 9, 16]

    // Return an iterable object that filters the specified iterable,
    // iterating only those elements for which the predicate returns true
    function filter(iterable, predicate) {
        let iterator = iterable[Symbol.iterator]();
        return { // This object is both iterator and iterable
            [Symbol.iterator]() { return this; },
            next() {
                for(;;) {
                    let v = iterator.next();
                    if (v.done || predicate(v.value)) {
                        return v;
                    }
                }
            }
        };
    }

    // Filter a range so we're left with only even numbers
    [...filter(new Range(1,10), x => x % 2 === 0)]  // => [2,4,6,8,10]
```

One key feature of iterable objects and iterators is that they are inherently lazy: when computation is required to compute the next value, that computation can be deferred until the value is actually needed. Suppose, for example, that you have a very long string of text that you want to tokenize into space-separated words. You could simply use the `split()` method of your string, but if you do this, then the entire string has to be processed before you can use even the first word. And you end up allocating lots of memory for the returned array and all of the strings within it. Here is a function that allows you to lazily iterate the words of a string without keeping them all in memory at once (in ES2020, this function

would be much easier to implement using the iterator-returning `matchAll()` method described in §11.3.2):

```
function words(s) {
    var r = /\s+|$/g;                       // Match one or more spaces or end
    r.lastIndex = s.match(/[^ ]/).index;    // Start matching at first nonspac
    return {                                // Return an iterable iterator obj
        [Symbol.iterator]() {               // This makes us iterable
            return this;
        },
        next() {                            // This makes us an iterator
            let start = r.lastIndex;        // Resume where the last match end
            if (start < s.length) {         // If we're not done
                let match = r.exec(s);      // Match the next word boundary
                if (match) {                // If we found one, return the wor
                    return { value: s.substring(start, match.index) };
                }
            }
            return { done: true };          // Otherwise, say that we're done
        }
    };
}

[...words(" abc def  ghi! ")] // => ["abc", "def", "ghi!"]
```

12.2.1 "Closing" an Iterator: The Return Method

Imagine a (server-side) JavaScript variant of the `words()` iterator that, instead of taking a source string as its argument, takes the name of a file, opens the file, reads lines from it, and iterates the words from those lines. In most operating systems, programs that open files to read from them need to remember to close those files when they are done reading, so this hypothetical iterator would be sure to close the file after the `next()` method returns the last word in it.

But iterators don't always run all the way to the end: a `for/of` loop might be terminated with a `break` or `return` or by an exception. Similarly, when an iterator is used with destructuring assignment, the `next()` method is only called enough times to obtain values for each of the specified variables. The iterator may have many more values it could return, but they will never be requested.

If our hypothetical words-in-a-file iterator never runs all the way to the end, it still needs to close the file it opened. For this reason, iterator ob-

jects may implement a `return()` method to go along with the `next()` method. If iteration stops before `next()` has returned an iteration result with the `done` property set to `true` (most commonly because you left a `for/of` loop early via a `break` statement), then the interpreter will check to see if the iterator object has a `return()` method. If this method exists, the interpreter will invoke it with no arguments, giving the iterator the chance to close files, release memory, and otherwise clean up after itself. The `return()` method must return an iterator result object. The properties of the object are ignored, but it is an error to return a non-object value.

The `for/of` loop and the spread operator are really useful features of JavaScript, so when you are creating APIs, it is a good idea to use them when possible. But having to work with an iterable object, its iterator object, and the iterator's result objects makes the process somewhat complicated. Fortunately, generators can dramatically simplify the creation of custom iterators, as we'll see in the rest of this chapter.

12.3 Generators

A *generator* is a kind of iterator defined with powerful new ES6 syntax; it's particularly useful when the values to be iterated are not the elements of a data structure, but the result of a computation.

To create a generator, you must first define a *generator function*. A generator function is syntactically like a regular JavaScript function but is defined with the keyword `function*` rather than `function`. (Technically, this is not a new keyword, just a `*` after the keyword `function` and before the function name.) When you invoke a generator function, it does not actually execute the function body, but instead returns a generator object. This generator object is an iterator. Calling its `next()` method causes the body of the generator function to run from the start (or whatever its current position is) until it reaches a `yield` statement. `yield` is new in ES6 and is something like a `return` statement. The value of the `yield` statement becomes the value returned by the `next()` call on the iterator. An example makes this clearer:

```
// A generator function that yields the set of one digit (base-10) primes.
function* oneDigitPrimes() { // Invoking this function does not run the code
    yield 2;                 // but just returns a generator object. Calling
    yield 3;                 // the next() method of that generator runs
```

```
    yield 5;                // the code until a yield statement provides
    yield 7;                // the return value for the next() method.
}

// When we invoke the generator function, we get a generator
let primes = oneDigitPrimes();

// A generator is an iterator object that iterates the yielded values
primes.next().value         // => 2
primes.next().value         // => 3
primes.next().value         // => 5
primes.next().value         // => 7
primes.next().done          // => true

// Generators have a Symbol.iterator method to make them iterable
primes[Symbol.iterator]()   // => primes

// We can use generators like other iterable types
[...oneDigitPrimes()]       // => [2,3,5,7]
let sum = 0;
for(let prime of oneDigitPrimes()) sum += prime;
sum                         // => 17
```

In this example, we used a `function*` statement to define a generator. Like regular functions, however, we can also define generators in expression form. Once again, we just put an asterisk after the `function` keyword:

```
const seq = function*(from,to) {
    for(let i = from; i <= to; i++) yield i;
};
[...seq(3,5)]  // => [3, 4, 5]
```

In classes and object literals, we can use shorthand notation to omit the `function` keyword entirely when we define methods. To define a generator in this context, we simply use an asterisk before the method name where the `function` keyword would have been, had we used it:

```
let o = {
    x: 1, y: 2, z: 3,
    // A generator that yields each of the keys of this object
    *g() {
        for(let key of Object.keys(this)) {
            yield key;
```

```
            }
        }
};
[...o.g()] // => ["x", "y", "z", "g"]
```

Note that there is no way to write a generator function using arrow function syntax.

Generators often make it particularly easy to define iterable classes. We can replace the `[Symbol.iterator]()` method show in Example 12-1 with a much shorter `*[Symbol.iterator&rbrack;()` generator function that looks like this:

```
*[Symbol.iterator]() {
    for(let x = Math.ceil(this.from); x <= this.to; x++) yield x;
}
```

See Example 9-3 in Chapter 9 to see this generator-based iterator function in context.

12.3.1 Generator Examples

Generators are more interesting if they actually *generate* the values they yield by doing some kind of computation. Here, for example, is a generator function that yields Fibonacci numbers:

```
function* fibonacciSequence() {
    let x = 0, y = 1;
    for(;;) {
        yield y;
        [x, y] = [y, x+y];  // Note: destructuring assignment
    }
}
```

Note that the `fibonacciSequence()` generator function here has an infinite loop and yields values forever without returning. If this generator is used with the `...` spread operator, it will loop until memory is exhausted and the program crashes. With care, it is possible to use it in a `for/of` loop, however:

```
// Return the nth Fibonacci number
function fibonacci(n) {
```

```
    for(let f of fibonacciSequence()) {
        if (n-- <= 0) return f;
    }
}
fibonacci(20)    // => 10946
```

This kind of infinite generator becomes more useful with a `take()` generator like this:

```
// Yield the first n elements of the specified iterable object
function* take(n, iterable) {
    let it = iterable[Symbol.iterator](); // Get iterator for iterable objec
    while(n-- > 0) {           // Loop n times:
        let next = it.next();  // Get the next item from the iterator.
        if (next.done) return; // If there are no more values, return early
        else yield next.value; // otherwise, yield the value
    }
}

// An array of the first 5 Fibonacci numbers
[...take(5, fibonacciSequence())]  // => [1, 1, 2, 3, 5]
```

Here is another useful generator function that interleaves the elements of multiple iterable objects:

```
// Given an array of iterables, yield their elements in interleaved order.
function* zip(...iterables) {
    // Get an iterator for each iterable
    let iterators = iterables.map(i => i[Symbol.iterator]());
    let index = 0;
    while(iterators.length > 0) {       // While there are still some iterat
        if (index >= iterators.length) {    // If we reached the last iterat
            index = 0;                      // go back to the first one.
        }
        let item = iterators[index].next(); // Get next item from next itera
        if (item.done) {                    // If that iterator is done
            iterators.splice(index, 1);     // then remove it from the array
        }
        else {                              // Otherwise,
            yield item.value;               // yield the iterated value
            index++;                        // and move on to the next itera
        }
    }
}
```

```
// Interleave three iterable objects
[...zip(oneDigitPrimes(),"ab",[0])]     // => [2,"a",0,3,"b",5,7]
```

12.3.2 yield* and Recursive Generators

In addition to the `zip()` generator defined in the preceding example, it might be useful to have a similar generator function that yields the elements of multiple iterable objects sequentially rather than interleaving them. We could write that generator like this:

```
function* sequence(...iterables) {
    for(let iterable of iterables) {
        for(let item of iterable) {
            yield item;
        }
    }
}

[...sequence("abc",oneDigitPrimes())]  // => ["a","b","c",2,3,5,7]
```

This process of yielding the elements of some other iterable object is common enough in generator functions that ES6 has special syntax for it. The `yield*` keyword is like `yield` except that, rather than yielding a single value, it iterates an iterable object and yields each of the resulting values. The `sequence()` generator function that we've used can be simplified with `yield*` like this:

```
function* sequence(...iterables) {
    for(let iterable of iterables) {
        yield* iterable;
    }
}

[...sequence("abc",oneDigitPrimes())]  // => ["a","b","c",2,3,5,7]
```

The array `forEach()` method is often an elegant way to loop over the elements of an array, so you might be tempted to write the `sequence()` function like this:

```
function* sequence(...iterables) {
    iterables.forEach(iterable => yield* iterable );  // Error
}
```

This does not work, however. `yield` and `yield*` can only be used within generator functions, but the nested arrow function in this code is a regular function, not a `function*` generator function, so `yield` is not allowed.

`yield*` can be used with any kind of iterable object, including iterables implemented with generators. This means that `yield*` allows us to define recursive generators, and you might use this feature to allow simple non-recursive iteration over a recursively defined tree structure, for example.

12.4 Advanced Generator Features

The most common use of generator functions is to create iterators, but the fundamental feature of generators is that they allow us to pause a computation, yield intermediate results, and then resume the computation later. This means that generators have features beyond those of iterators, and we explore those features in the following sections.

12.4.1 The Return Value of a Generator Function

The generator functions we've seen so far have not had `return` statements, or if they have, they have been used to cause an early return, not to return a value. Like any function, though, a generator function can return a value. In order to understand what happens in this case, recall how iteration works. The return value of the `next()` function is an object that has a `value` property and/or a `done` property. With typical iterators and generators, if the `value` property is defined, then the `done` property is undefined or is `false`. And if `done` is `true`, then `value` is undefined. But in the case of a generator that returns a value, the final call to `next` returns an object that has both `value` and `done` defined. The `value` property holds the return value of the generator function, and the `done` property is `true`, indicating that there are no more values to iterate. This final value is ignored by the `for/of` loop and by the spread operator, but it is available to code that manually iterates with explicit calls to `next()`:

```
function *oneAndDone() {
    yield 1;
```

```
    return "done";
}

// The return value does not appear in normal iteration.
[...oneAndDone()]   // => [1]

// But it is available if you explicitly call next()
let generator = oneAndDone();
generator.next()           // => { value: 1, done: false}
generator.next()           // => { value: "done", done: true }
// If the generator is already done, the return value is not returned again
generator.next()           // => { value: undefined, done: true }
```

12.4.2 The Value of a yield Expression

In the preceding discussion, we've treated `yield` as a statement that takes a value but has no value of its own. In fact, however, `yield` is an expression, and it can have a value.

When the `next()` method of a generator is invoked, the generator function runs until it reaches a `yield` expression. The expression that follows the `yield` keyword is evaluated, and that value becomes the return value of the `next()` invocation. At this point, the generator function stops executing right in the middle of evaluating the `yield` expression. The next time the `next()` method of the generator is called, the argument passed to `next()` becomes the value of the `yield` expression that was paused. So the generator returns values to its caller with `yield`, and the caller passes values in to the generator with `next()`. The generator and caller are two separate streams of execution passing values (and control) back and forth. The following code illustrates:

```
function* smallNumbers() {
    console.log("next() invoked the first time; argument discarded");
    let y1 = yield 1;    // y1 == "b"
    console.log("next() invoked a second time with argument", y1);
    let y2 = yield 2;    // y2 == "c"
    console.log("next() invoked a third time with argument", y2);
    let y3 = yield 3;    // y3 == "d"
    console.log("next() invoked a fourth time with argument", y3);
    return 4;
}

let g = smallNumbers();
console.log("generator created; no code runs yet");
```

```
let n1 = g.next("a");   // n1.value == 1
console.log("generator yielded", n1.value);
let n2 = g.next("b");   // n2.value == 2
console.log("generator yielded", n2.value);
let n3 = g.next("c");   // n3.value == 3
console.log("generator yielded", n3.value);
let n4 = g.next("d");   // n4 == { value: 4, done: true }
console.log("generator returned", n4.value);
```

When this code runs, it produces the following output that demonstrates the back-and-forth between the two blocks of code:

```
generator created; no code runs yet
next() invoked the first time; argument discarded
generator yielded 1
next() invoked a second time with argument b
generator yielded 2
next() invoked a third time with argument c
generator yielded 3
next() invoked a fourth time with argument d
generator returned 4
```

Note the asymmetry in this code. The first invocation of `next()` starts the generator, but the value passed to that invocation is not accessible to the generator.

12.4.3 The return() and throw() Methods of a Generator

We've seen that you can receive values yielded by or returned by a generator function. And you can pass values to a running generator by passing those values when you call the `next()` method of the generator.

In addition to providing input to a generator with `next()`, you can also alter the flow of control inside the generator by calling its `return()` and `throw()` methods. As the names suggest, calling these methods on a generator causes it to return a value or throw an exception as if the next statement in the generator was a `return` or `throw`.

Recall from earlier in the chapter that, if an iterator defines a `return()` method and iteration stops early, then the interpreter automatically calls the `return()` method to give the iterator a chance to close files or do

other cleanup. In the case of generators, you can't define a custom `return()` method to handle cleanup, but you can structure the generator code to use a `try/finally` statement that ensures the necessary cleanup is done (in the `finally` block) when the generator returns. By forcing the generator to return, the generator's built-in `return()` method ensures that the cleanup code is run when the generator will no longer be used.

Just as the `next()` method of a generator allows us to pass arbitrary values into a running generator, the `throw()` method of a generator gives us a way to send arbitrary signals (in the form of exceptions) into a generator. Calling the `throw()` method always causes an exception inside the generator. But if the generator function is written with appropriate exception-handling code, the exception need not be fatal but can instead be a means of altering the behavior of the generator. Imagine, for example, a counter generator that yields an ever-increasing sequence of integers. This could be written so that an exception sent with `throw()` would reset the counter to zero.

When a generator uses `yield*` to yield values from some other iterable object, then a call to the `next()` method of the generator causes a call to the `next()` method of the iterable object. The same is true of the `return()` and `throw()` methods. If a generator uses `yield*` on an iterable object that has these methods defined, then calling `return()` or `throw()` on the generator causes the iterator's `return()` or `throw()` method to be called in turn. All iterators *must* have a `next()` method. Iterators that need to clean up after incomplete iteration *should* define a `return()` method. And any iterator *may* define a `throw()` method, though I don't know of any practical reason to do so.

12.4.4 A Final Note About Generators

Generators are a very powerful generalized control structure. They give us the ability to pause a computation with `yield` and restart it again at some arbitrary later time with an arbitrary input value. It is possible to use generators to create a kind of cooperative threading system within single-threaded JavaScript code. And it is possible to use generators to mask asynchronous parts of your program so that your code appears sequential and synchronous, even though some of your function calls are actually asynchronous and depend on events from the network.

Trying to do these things with generators leads to code that is mind-bendingly hard to understand or to explain. It has been done, however, and the only really practical use case has been for managing asynchronous code. JavaScript now has `async` and `await` keywords (see Chapter 13) for this very purpose, however, and there is no longer any reason to abuse generators in this way.

12.5 Summary

In this chapter, you have learned:

- The `for/of` loop and the `...` spread operator work with iterable objects.
- An object is iterable if it has a method with the symbolic name `[Symbol.iterator]` that returns an iterator object.
- An iterator object has a `next()` method that returns an iteration result object.
- An iteration result object has a `value` property that holds the next iterated value, if there is one. If the iteration has completed, then the result object must have a `done` property set to `true`.
- You can implement your own iterable objects by defining a `[Symbol.iterator]()` method that returns an object with a `next()` method that returns iteration result objects. You can also implement functions that accept iterator arguments and return iterator values.
- Generator functions (functions defined with `function*` instead of `function`) are another way to define iterators.
- When you invoke a generator function, the body of the function does not run right away; instead, the return value is an iterable iterator object. Each time the `next()` method of the iterator is called, another chunk of the generator function runs.
- Generator functions can use the `yield` operator to specify the values that are returned by the iterator. Each call to `next()` causes the generator function to run up to the next `yield` expression. The value of that `yield` expression then becomes the value returned by the iterator. When there are no more `yield` expressions, then the generator function returns, and the iteration is complete.