You Don't Know JS: ES6 & Beyond

Chapter 3: Organization

It's one thing to write JS code, but it's another to properly organize it. Utilizing common patterns for organization and reuse goes a long way to improving the readability and understandability of your code. Remember: code is at least as much about communicating to other developers as it is about feeding the computer instructions.

ES6 has several important features that help significantly improve these patterns, including: iterators, generators, modules, and classes.

Iterators

An *iterator* is a structured pattern for pulling information from a source in one-at-a-time fashion. This pattern has been around programming for a long time. And to be sure, JS developers have been ad hoc designing and implementing iterators in JS programs since before anyone can remember, so it's not at all a new topic.

What ES6 has done is introduce an implicit standardized interface for iterators. Many of the built-in data structures in JavaScript will now expose an iterator implementing this standard. And you can also construct your own iterators adhering to the same standard, for maximal interoperability.

Iterators are a way of organizing ordered, sequential, pull-based consumption of data.

For example, you may implement a utility that produces a new unique identifier each time it's requested. Or you may produce an infinite series of values that rotate through a fixed list, in round-robin fashion. Or you could attach an iterator to a database query result to pull out new rows one at a time.

Although they have not commonly been used in JS in such a manner, iterators can also be thought of as controlling behavior one step at a time. This can be illustrated quite clearly when considering generators (see "Generators" later in this chapter), though you can certainly do the same without generators.

Interfaces

At the time of this writing, ES6 section 25.1.1.2 (https://people.mozilla.org/~jorendorff/es6-draft.html#sec-iterator-interface) details the `Iterator` interface as having the following requirement:

```
Iterator [required]
        next() {method}: retrieves next IteratorResult
```

There are two optional members that some iterators are extended with:

```
Iterator [optional]
        return() {method}: stops iterator and returns IteratorResult
        throw() {method}: signals error and returns IteratorResult
```

The `IteratorResult` interface is specified as:

```
IteratorResult
        value {property}: current iteration value or final return value
                (optional if `undefined`)
        done {property}: boolean, indicates completion status
```

Note: I call these interfaces implicit not because they're not explicitly called out in the specification -- they are! -- but because they're not exposed as direct objects accessible to code. JavaScript does not, in ES6, support any notion of "interfaces," so adherence for your own code is purely conventional. However, wherever JS expects an iterator -- a `for..of` loop, for instance -- what you provide must adhere to these interfaces or the code will fail.

There's also an `Iterable` interface, which describes objects that must be able to produce iterators:

```
Iterable
        @@iterator() {method}: produces an Iterator
```

If you recall from "Built-In Symbols" in Chapter 2, `@@iterator` is the special built-in symbol representing the method that can produce iterator(s) for the object.

IteratorResult

The `IteratorResult` interface specifies that the return value from any iterator operation will be an object of the form:

```
{ value: .. , done: true / false }
```

Built-in iterators will always return values of this form, but more properties are, of course, allowed to be present on the return value, as necessary.

For example, a custom iterator may add additional metadata to the result object (e.g., where the data came from, how long it took to retrieve, cache expiration length, frequency for the appropriate next request, etc.).

Note: Technically, `value` is optional if it would otherwise be considered absent or unset, such as in the case of the value `undefined`. Because accessing `res.value` will produce `undefined` whether it's present with that value or absent entirely, the presence/absence of the property is more an implementation detail or an optimization (or both), rather than a functional issue.

`next()` Iteration

Let's look at an array, which is an iterable, and the iterator it can produce to consume its values:

```
var arr = [1,2,3];

var it = arr[Symbol.iterator]();

it.next();              // { value: 1, done: false }
it.next();              // { value: 2, done: false }
it.next();              // { value: 3, done: false }

it.next();              // { value: undefined, done: true }
```

Each time the method located at `Symbol.iterator` (see Chapters 2 and 7) is invoked on this `arr` value, it will produce a new fresh iterator. Most structures will do the same, including all the built-in data structures in JS.

However, a structure like an event queue consumer might only ever produce a single iterator (singleton pattern). Or a structure might only allow one unique iterator at a time, requiring the current one to be completed before a new one can be created.

The `it` iterator in the previous snippet doesn't report `done: true` when you receive the `3` value. You have to call `next()` again, in essence going beyond the end of the array's values, to get the complete signal `done: true`. It may not be clear why until later in this section, but that design decision will typically be considered a best practice.

Primitive string values are also iterables by default:

```
var greeting = "hello world";
```

```
var it = greeting[Symbol.iterator]();

it.next();              // { value: "h", done: false }
it.next();              // { value: "e", done: false }
..
```

Note: Technically, the primitive value itself isn't iterable, but thanks to "boxing", `"hello world"` is coerced/converted to its `String` object wrapper form, which *is* an iterable. See the *Types & Grammar* title of this series for more information.

ES6 also includes several new data structures, called collections (see Chapter 5). These collections are not only iterables themselves, but they also provide API method(s) to generate an iterator, such as:

```
var m = new Map();
m.set( "foo", 42 );
m.set( { cool: true }, "hello world" );

var it1 = m[Symbol.iterator]();
var it2 = m.entries();

it1.next();             // { value: [ "foo", 42 ], done: false }
it2.next();             // { value: [ "foo", 42 ], done: false }
..
```

The `next(..)` method of an iterator can optionally take one or more arguments. The built-in iterators mostly do not exercise this capability, though a generator's iterator definitely does (see "Generators" later in this chapter).

By general convention, including all the built-in iterators, calling `next(..)` on an iterator that's already been exhausted is not an error, but will simply continue to return the result `{ value: undefined, done: true }`.

Optional: `return(..)` and `throw(..)`

The optional methods on the iterator interface -- `return(..)` and `throw(..)` -- are not implemented on most of the built-in iterators. However, they definitely do mean something in the context of generators, so see "Generators" for more specific information.

`return(..)` is defined as sending a signal to an iterator that the consuming code is complete and will not be pulling any more values from it. This signal can be used to notify the producer (the iterator responding to `next(..)` calls) to perform any cleanup it may need to do, such as releasing/closing network, database, or file handle resources.

If an iterator has a `return(..)` present and any condition occurs that can automatically be interpreted as abnormal or early termination of consuming the iterator, `return(..)` will automatically be called. You can call `return(..)` manually as well.

`return(..)` will return an `IteratorResult` object just like `next(..)` does. In general, the optional value you send to `return(..)` would be sent back as `value` in this `IteratorResult`, though there are nuanced cases where that might not be true.

`throw(..)` is used to signal an exception/error to an iterator, which possibly may be used differently by the iterator than the completion signal implied by `return(..)`. It does not necessarily imply a complete stop of the iterator as `return(..)` generally does.

For example, with generator iterators, `throw(..)` actually injects a thrown exception into the generator's paused execution context, which can be caught with a `try..catch`. An uncaught `throw(..)` exception would end up abnormally aborting the generator's iterator.

Note: By general convention, an iterator should not produce any more results after having called `return(..)` or `throw(..)`.

Iterator Loop

As we covered in the "`for..of`" section in Chapter 2, the ES6 `for..of` loop directly consumes a conforming iterable.

If an iterator is also an iterable, it can be used directly with the `for..of` loop. You make an iterator an iterable by giving it a `Symbol.iterator` method that simply returns the iterator itself:

```
var it = {
        // make the `it` iterator an iterable
        [Symbol.iterator]() { return this; },

        next() { .. },
        ..
};

it[Symbol.iterator]() === it;           // true
```

Now we can consume the `it` iterator with a `for..of` loop:

```
for (var v of it) {
        console.log( v );
}
```

To fully understand how such a loop works, recall the `for` equivalent of a `for..of` loop from Chapter 2:

```
for (var v, res; (res = it.next()) && !res.done; ) {
        v = res.value;
        console.log( v );
}
```

If you look closely, you'll see that `it.next()` is called before each iteration, and then `res.done` is consulted. If `res.done` is `true`, the expression evaluates to `false` and the iteration doesn't occur.

Recall earlier that we suggested iterators should in general not return `done: true` along with the final intended value from the iterator. Now you can see why.

If an iterator returned `{ done: true, value: 42 }`, the `for..of` loop would completely discard the `42` value and it'd be lost. For this reason, assuming that your iterator may be consumed by patterns like the `for..of` loop or its manual `for` equivalent, you should probably wait to return `done: true` for signaling completion until after you've already returned all relevant iteration values.

Warning: You can, of course, intentionally design your iterator to return some relevant `value` at the same time as returning `done: true`. But don't do this unless you've documented that as the case, and thus implicitly forced consumers of your iterator to use a different pattern for iteration than is implied by `for..of` or its manual equivalent we depicted.

Custom Iterators

In addition to the standard built-in iterators, you can make your own! All it takes to make them interoperate with ES6's consumption facilities (e.g., the `for..of` loop and the `...` operator) is to adhere to the proper interface(s).

Let's try constructing an iterator that produces the infinite series of numbers in the Fibonacci sequence:

```
var Fib = {
        [Symbol.iterator]() {
                var n1 = 1, n2 = 1;

                return {
                        // make the iterator an iterable
                        [Symbol.iterator]() { return this; },

                        next() {
                                var current = n2;
                                n2 = n1;
                                n1 = n1 + current;
                                return { value: current, done: false };
```

```
                        },

                        return(v) {
                                console.log(
                                        "Fibonacci sequence abandoned."
                                );
                                return { value: v, done: true };
                        }
                };
        }
};

for (var v of Fib) {
        console.log( v );

        if (v > 50) break;
}
// 1 1 2 3 5 8 13 21 34 55
// Fibonacci sequence abandoned.
```

Warning: If we hadn't inserted the `break` condition, this `for..of` loop would have run forever, which is probably not the desired result in terms of breaking your program!

The `Fib[Symbol.iterator]()` method when called returns the iterator object with `next()` and `return(..)` methods on it. State is maintained via `n1` and `n2` variables, which are kept by the closure.

Let's *next* consider an iterator that is designed to run through a series (aka a queue) of actions, one item at a time:

```
var tasks = {
        [Symbol.iterator]() {
                var steps = this.actions.slice();

                return {
                        // make the iterator an iterable
                        [Symbol.iterator]() { return this; },

                        next(...args) {
                                if (steps.length > 0) {
                                        let res = steps.shift()( ...args );
                                        return { value: res, done: false };
                                }
                                else {
                                        return { done: true }
                                }
                        },

                        return(v) {
                                steps.length = 0;
                                return { value: v, done: true };
                        }
                };
        },
        actions: []
};
```

The iterator on `tasks` steps through functions found in the `actions` array property, if any, and executes them one at a time, passing in whatever arguments you pass to `next(..)`, and returning any return value to you in the standard `IteratorResult` object.

Here's how we could use this `tasks` queue:

```
tasks.actions.push(
        function step1(x){
                console.log( "step 1:", x );
                return x * 2;
        },
```

```
        function step2(x,y){
                console.log( "step 2:", x, y );
                return x + (y * 2);
        },
        function step3(x,y,z){
                console.log( "step 3:", x, y, z );
                return (x * y) + z;
        }
);

var it = tasks[Symbol.iterator]();

it.next( 10 );                  // step 1: 10
                                                // { value:   20, done: false }

it.next( 20, 50 );              // step 2: 20 50
                                                // { value:  120, done: false }

it.next( 20, 50, 120 ); // step 3: 20 50 120
                                                // { value: 1120, done: false }

it.next();                              // { done: true }
```

This particular usage reinforces that iterators can be a pattern for organizing functionality, not just data. It's also reminiscent of what we'll see with generators in the next section.

You could even get creative and define an iterator that represents meta operations on a single piece of data. For example, we could define an iterator for numbers that by default ranges from `0` up to (or down to, for negative numbers) the number in question.

Consider:

```
if (!Number.prototype[Symbol.iterator]) {
        Object.defineProperty(
                Number.prototype,
                Symbol.iterator,
                {
                        writable: true,
                        configurable: true,
                        enumerable: false,
                        value: function iterator(){
                                var i, inc, done = false, top = +this;

                                // iterate positively or negatively?
                                inc = 1 * (top < 0 ? -1 : 1);

                                return {
                                        // make the iterator itself an iterable!
                                        [Symbol.iterator](){ return this; },

                                        next() {
                                                if (!done) {
                                                        // initial iteration always 0
                                                        if (i == null) {
                                                                i = 0;
                                                        }
                                                        // iterating positively
                                                        else if (top >= 0) {
                                                                i = Math.min(top,i + inc);
                                                        }
                                                        // iterating negatively
                                                        else {
                                                                i = Math.max(top,i + inc);
                                                        }

                                                        // done after this iteration?
                                                        if (i == top) done = true;
```

```
                                                        return { value: i, done: false };
                                                }
                                                else {
                                                        return { done: true };
                                                }
                                        }
                                };
                        }
                }
        );
}
```

Now, what tricks does this creativity afford us?

```
for (var i of 3) {
        console.log( i );
}
// 0 1 2 3

[...-3];                                // [0,-1,-2,-3]
```

Those are some fun tricks, though the practical utility is somewhat debatable. But then again, one might wonder why ES6 didn't just ship with such a minor but delightful feature easter egg!?

I'd be remiss if I didn't at least remind you that extending native prototypes as I'm doing in the previous snippet is something you should only do with caution and awareness of potential hazards.

In this case, the chances that you'll have a collision with other code or even a future JS feature is probably exceedingly low. But just beware of the slight possibility. And document what you're doing verbosely for posterity's sake.

Note: I've expounded on this particular technique in this blog post (http://blog.getify.com/iterating-es6-numbers/) if you want more details. And this comment (http://blog.getify.com/iterating-es6-numbers/comment-page-1/#comment-535294) even suggests a similar trick but for making string character ranges.

Iterator Consumption

We've already shown consuming an iterator item by item with the `for..of` loop. But there are other ES6 structures that can consume iterators.

Let's consider the iterator attached to this array (though any iterator we choose would have the following behaviors):

```
var a = [1,2,3,4,5];
```

The `...` spread operator fully exhausts an iterator. Consider:

```
function foo(x,y,z,w,p) {
        console.log( x + y + z + w + p );
}

foo( ...a );                    // 15
```

`...` can also spread an iterator inside an array:

```
var b = [ 0, ...a, 6 ];
b;                                          // [0,1,2,3,4,5,6]
```

Array destructuring (see "Destructuring" in Chapter 2) can partially or completely (if paired with a `...` rest/gather operator) consume an iterator:

```
var it = a[Symbol.iterator]();

var [x,y] = it;                 // take just the first two elements from `it`
var [z, ...w] = it;             // take the third, then the rest all at once

// is `it` fully exhausted? Yep.
it.next();                              // { value: undefined, done: true }

x;                                              // 1
y;                                              // 2
z;                                              // 3
w;                                              // [4,5]
```

Generators

All functions run to completion, right? In other words, once a function starts running, it finishes before anything else can interrupt.

At least that's how it's been for the whole history of JavaScript up to this point. As of ES6, a new somewhat exotic form of function is being introduced, called a generator. A generator can pause itself in mid-execution, and can be resumed either right away or at a later time. So it clearly does not hold the run-to-completion guarantee that normal functions do.

Moreover, each pause/resume cycle in mid-execution is an opportunity for two-way message passing, where the generator can return a value, and the controlling code that resumes it can send a value back in.

As with iterators in the previous section, there are multiple ways to think about what a generator is, or rather what it's most useful for. There's no one right answer, but we'll try to consider several angles.

Note: See the *Async & Performance* title of this series for more information about generators, and also see Chapter 4 of this current title.

Syntax

The generator function is declared with this new syntax:

```
function *foo() {
        // ..
}
```

The position of the `*` is not functionally relevant. The same declaration could be written as any of the following:

```
function *foo()  { .. }
function* foo()  { .. }
function * foo() { .. }
function*foo()   { .. }
..
```

The *only* difference here is stylistic preference. Most other literature seems to prefer `function* foo(..) { .. }`. I prefer `function *foo(..) { .. }`, so that's how I'll present them for the rest of this title.

My reason is purely didactic in nature. In this text, when referring to a generator function, I will use `*foo(..)`, as opposed to `foo(..)` for a normal function. I observe that `*foo(..)` more closely matches the `*` positioning of `function *foo(..) { .. }`.

Moreover, as we saw in Chapter 2 with concise methods, there's a concise generator form in object literals:

```
var a = {
        *foo() { .. }
};
```

I would say that with concise generators, `*foo() { .. }` is rather more natural than `* foo() { .. }`. So that further argues for matching the consistency with `*foo()`.

Consistency eases understanding and learning.

Executing a Generator

Though a generator is declared with `*`, you still execute it like a normal function:

```
foo();
```

You can still pass it arguments, as in:

```
function *foo(x,y) {
	// ..
}

foo( 5, 10 );
```

The major difference is that executing a generator, like `foo(5,10)` doesn't actually run the code in the generator. Instead, it produces an iterator that will control the generator to execute its code.

We'll come back to this later in "Iterator Control," but briefly:

```
function *foo() {
	// ..
}

var it = foo();

// to start/advanced `*foo()`, call
// `it.next(..)`
```

`yield`

Generators also have a new keyword you can use inside them, to signal the pause point: `yield`. Consider:

```
function *foo() {
	var x = 10;
	var y = 20;

	yield;

	var z = x + y;
}
```

In this `*foo()` generator, the operations on the first two lines would run at the beginning, then `yield` would pause the generator. If and when resumed, the last line of `*foo()` would run. `yield` can appear any number of times (or not at all, technically!) in a generator.

You can even put `yield` inside a loop, and it can represent a repeated pause point. In fact, a loop that never completes just means a generator that never completes, which is completely valid, and sometimes entirely what you need.

`yield` is not just a pause point. It's an expression that sends out a value when pausing the generator. Here's a `while..true` loop in a generator that for each iteration `yield`s a new random number:

```
function *foo() {
	while (true) {
		yield Math.random();
```

```
        }
}
```

The `yield ..` expression not only sends a value -- `yield` without a value is the same as `yield undefined` -- but also receives (e.g., is replaced by) the eventual resumption value. Consider:

```
function *foo() {
        var x = yield 10;
        console.log( x );
}
```

This generator will first `yield` out the value `10` when pausing itself. When you resume the generator -- using the `it.next(..)` we referred to earlier -- whatever value (if any) you resume with will replace/complete the whole `yield 10` expression, meaning that value will be assigned to the `x` variable.

A `yield ..` expression can appear anywhere a normal expression can. For example:

```
function *foo() {
        var arr = [ yield 1, yield 2, yield 3 ];
        console.log( arr, yield 4 );
}
```

`*foo()` here has four `yield ..` expressions. Each `yield` results in the generator pausing to wait for a resumption value that's then used in the various expression contexts.

`yield` is not technically an operator, though when used like `yield 1` it sure looks like it. Because `yield` can be used all by itself as in `var x = yield;`, thinking of it as an operator can sometimes be confusing.

Technically, `yield ..` is of the same "expression precedence" -- similar conceptually to operator precedence -- as an assignment expression like `a = 3`. That means `yield ..` can basically appear anywhere `a = 3` can validly appear.

Let's illustrate the symmetry:

```
var a, b;

a = 3;                          // valid
b = 2 + a = 3;                  // invalid
b = 2 + (a = 3);                // valid

yield 3;                        // valid
a = 2 + yield 3;                // invalid
a = 2 + (yield 3);              // valid
```

Note: If you think about it, it makes a sort of conceptual sense that a `yield ..` expression would behave similar to an assignment expression. When a paused `yield` expression is resumed, it's completed/replaced by the resumption value in a way that's not terribly dissimilar from being "assigned" that value.

The takeaway: if you need `yield ..` to appear in a position where an assignment like `a = 3` would not itself be allowed, it needs to be wrapped in a `( )`.

Because of the low precedence of the `yield` keyword, almost any expression after a `yield ..` will be computed first before being sent with `yield`. Only the `...` spread operator and the `,` comma operator have lower precedence, meaning they'd bind after the `yield` has been evaluated.

So just like with multiple operators in normal statements, another case where `( )` might be needed is to override (elevate) the low precedence of `yield`, such as the difference between these expressions:

```
yield 2 + 3;                    // same as `yield (2 + 3)`

(yield 2) + 3;                  // `yield 2` first, then `+ 3`
```

Just like `=` assignment, `yield` is also "right-associative," which means that multiple `yield` expressions in succession are treated as having been `( .. )` grouped from right to left. So, `yield yield yield 3` is treated as `yield (yield (yield 3))`. A "left-associative" interpretation like `((yield) yield) yield 3` would make no sense.

Just like with operators, it's a good idea to use `( .. )` grouping, even if not strictly required, to disambiguate your intent if `yield` is combined with other operators or `yield`s.

Note: See the *Types & Grammar* title of this series for more information about operator precedence and associativity.

`yield *`

In the same way that the `*` makes a `function` declaration into `function *` generator declaration, a `*` makes `yield` into `yield *`, which is a very different mechanism, called *yield delegation*. Grammatically, `yield *..` will behave the same as a `yield ..`, as discussed in the previous section.

`yield * ..` requires an iterable; it then invokes that iterable's iterator, and delegates its own host generator's control to that iterator until it's exhausted. Consider:

```
function *foo() {
        yield *[1,2,3];
}
```

Note: As with the `*` position in a generator's declaration (discussed earlier), the `*` positioning in `yield *` expressions is stylistically up to you. Most other literature prefers `yield* ..`, but I prefer `yield *..`, for very symmetrical reasons as already discussed.

The `[1,2,3]` value produces an iterator that will step through its values, so the `*foo()` generator will yield those values out as it's consumed. Another way to illustrate the behavior is in yield delegating to another generator:

```
function *foo() {
        yield 1;
        yield 2;
        yield 3;
}

function *bar() {
        yield *foo();
}
```

The iterator produced when `*bar()` calls `*foo()` is delegated to via `yield *`, meaning whatever value(s) `*foo()` produces will be produced by `*bar()`.

Whereas with `yield ..` the completion value of the expression comes from resuming the generator with `it.next(..)`, the completion value of the `yield *..` expression comes from the return value (if any) from the delegated-to iterator.

Built-in iterators generally don't have return values, as we covered at the end of the "Iterator Loop" section earlier in this chapter. But if you define your own custom iterator (or generator), you can design it to `return` a value, which `yield *..` would capture:

```
function *foo() {
        yield 1;
        yield 2;
        yield 3;
        return 4;
}

function *bar() {
        var x = yield *foo();
        console.log( "x:", x );
}
```

```
for (var v of bar()) {
        console.log( v );
}
// 1 2 3
// x: 4
```

While the `1`, `2`, and `3` values are `yield`ed out of `*foo()` and then out of `*bar()`, the `4` value returned from `*foo()` is the completion value of the `yield *foo()` expression, which then gets assigned to `x`.

Because `yield *` can call another generator (by way of delegating to its iterator), it can also perform a sort of generator recursion by calling itself:

```
function *foo(x) {
        if (x < 3) {
                x = yield *foo( x + 1 );
        }
        return x * 2;
}

foo( 1 );
```

The result from `foo(1)` and then calling the iterator's `next()` to run it through its recursive steps will be `24`. The first `*foo(..)` run has `x` at value `1`, which is `x < 3`. `x + 1` is passed recursively to `*foo(..)`, so `x` is then `2`. One more recursive call results in `x` of `3`.

Now, because `x < 3` fails, the recursion stops, and `return 3 * 2` gives `6` back to the previous call's `yield *..` expression, which is then assigned to `x`. Another `return 6 * 2` returns `12` back to the previous call's `x`. Finally `12 * 2`, or `24`, is returned from the completed run of the `*foo(..)` generator.

Iterator Control

Earlier, we briefly introduced the concept that generators are controlled by iterators. Let's fully dig into that now.

Recall the recursive `*foo(..)` from the previous section. Here's how we'd run it:

```
function *foo(x) {
        if (x < 3) {
                x = yield *foo( x + 1 );
        }
        return x * 2;
}

var it = foo( 1 );
it.next();                              // { value: 24, done: true }
```

In this case, the generator doesn't really ever pause, as there's no `yield ..` expression. Instead, `yield *` just keeps the current iteration step going via the recursive call. So, just one call to the iterator's `next()` function fully runs the generator.

Now let's consider a generator that will have multiple steps and thus multiple produced values:

```
function *foo() {
        yield 1;
        yield 2;
        yield 3;
}
```

We already know we can consume an iterator, even one attached to a generator like `*foo()`, with a `for..of` loop:

```
for (var v of foo()) {
        console.log( v );
```

```
}
// 1 2 3
```

Note: The `for..of` loop requires an iterable. A generator function reference (like `foo`) by itself is not an iterable; you must execute it with `foo()` to get the iterator (which is also an iterable, as we explained earlier in this chapter). You could theoretically extend the `GeneratorPrototype` (the prototype of all generator functions) with a `Symbol.iterator` function that essentially just does `return this()`. That would make the `foo` reference itself an iterable, which means `for (var v of foo) { .. }` (notice no `()` on `foo`) will work.

Let's instead iterate the generator manually:

```
function *foo() {
        yield 1;
        yield 2;
        yield 3;
}

var it = foo();

it.next();                              // { value: 1, done: false }
it.next();                              // { value: 2, done: false }
it.next();                              // { value: 3, done: false }

it.next();                              // { value: undefined, done: true }
```

If you look closely, there are three `yield` statements and four `next()` calls. That may seem like a strange mismatch. In fact, there will always be one more `next()` call than `yield` expression, assuming all are evaluated and the generator is fully run to completion.

But if you look at it from the opposite perspective (inside-out instead of outside-in), the matching between `yield` and `next()` makes more sense.

Recall that the `yield ..` expression will be completed by the value you resume the generator with. That means the argument you pass to `next(..)` completes whatever `yield ..` expression is currently paused waiting for a completion.

Let's illustrate this perspective this way:

```
function *foo() {
        var x = yield 1;
        var y = yield 2;
        var z = yield 3;
        console.log( x, y, z );
}
```

In this snippet, each `yield ..` is sending a value out (`1`, `2`, `3`), but more directly, it's pausing the generator to wait for a value. In other words, it's almost like asking the question, "What value should I use here? I'll wait to hear back."

Now, here's how we control `*foo()` to start it up:

```
var it = foo();

it.next();                              // { value: 1, done: false }
```

That first `next()` call is starting up the generator from its initial paused state, and running it to the first `yield`. At the moment you call that first `next()`, there's no `yield ..` expression waiting for a completion. If you passed a value to that first `next()` call, it would currently just be thrown away, because no `yield` is waiting to receive such a value.

Note: An early proposal for the "beyond ES6" timeframe *would* let you access a value passed to an initial `next(..)` call via a separate meta property (see Chapter 7) inside the generator.

Now, let's answer the currently pending question, "What value should I assign to `x`?" We'll answer it by sending a value to the *next* `next(..)` call:

```
it.next( "foo" );               // { value: 2, done: false }
```

Now, the `x` will have the value `"foo"`, but we've also asked a new question, "What value should I assign to `y`?" And we answer:

```
it.next( "bar" );               // { value: 3, done: false }
```

Answer given, another question asked. Final answer:

```
it.next( "baz" );               // "foo" "bar" "baz"
                                                // { value: undefined, done: true }
```

Now it should be clearer how each `yield ..` "question" is answered by the *next* `next(..)` call, and so the "extra" `next()` call we observed is always just the initial one that starts everything going.

Let's put all those steps together:

```
var it = foo();

// start up the generator
it.next();                              // { value: 1, done: false }

// answer first question
it.next( "foo" );               // { value: 2, done: false }

// answer second question
it.next( "bar" );               // { value: 3, done: false }

// answer third question
it.next( "baz" );               // "foo" "bar" "baz"
                                                // { value: undefined, done: true }
```

You can think of a generator as a producer of values, in which case each iteration is simply producing a value to be consumed.

But in a more general sense, perhaps it's appropriate to think of generators as controlled, progressive code execution, much like the `tasks` queue example from the earlier "Custom Iterators" section.

Note: That perspective is exactly the motivation for how we'll revisit generators in Chapter 4. Specifically, there's no reason that `next(..)` has to be called right away after the previous `next(..)` finishes. While the generator's inner execution context is paused, the rest of the program continues unblocked, including the ability for asynchronous actions to control when the generator is resumed.

Early Completion

As we covered earlier in this chapter, the iterator attached to a generator supports the optional `return(..)` and `throw(..)` methods. Both of them have the effect of aborting a paused generator immediately.

Consider:

```
function *foo() {
        yield 1;
        yield 2;
        yield 3;
}

var it = foo();
```

```
it.next();                              // { value: 1, done: false }

it.return( 42 );                // { value: 42, done: true }

it.next();                              // { value: undefined, done: true }
```

`return(x)` is kind of like forcing a `return x` to be processed at exactly that moment, such that you get the specified value right back. Once a generator is completed, either normally or early as shown, it no longer processes any code or returns any values.

In addition to `return(..)` being callable manually, it's also called automatically at the end of iteration by any of the ES6 constructs that consume iterators, such as the `for..of` loop and the `...` spread operator.

The purpose for this capability is so the generator can be notified if the controlling code is no longer going to iterate over it anymore, so that it can perhaps do any cleanup tasks (freeing up resources, resetting status, etc.). Identical to a normal function cleanup pattern, the main way to accomplish this is to use a `finally` clause:

```
function *foo() {
        try {
                yield 1;
                yield 2;
                yield 3;
        }
        finally {
                console.log( "cleanup!" );
        }
}

for (var v of foo()) {
        console.log( v );
}
// 1 2 3
// cleanup!

var it = foo();

it.next();                              // { value: 1, done: false }
it.return( 42 );                // cleanup!
                                                        // { value: 42, done: true }
```

Warning: Do not put a `yield` statement inside the `finally` clause! It's valid and legal, but it's a really terrible idea. It acts in a sense as deferring the completion of the `return(..)` call you made, as any `yield ..` expressions in the `finally` clause are respected to pause and send messages; you don't immediately get a completed generator as expected. There's basically no good reason to opt in to that crazy *bad part*, so avoid doing so!

In addition to the previous snippet showing how `return(..)` aborts the generator while still triggering the `finally` clause, it also demonstrates that a generator produces a whole new iterator each time it's called. In fact, you can use multiple iterators attached to the same generator concurrently:

```
function *foo() {
        yield 1;
        yield 2;
        yield 3;
}

var it1 = foo();
it1.next();                             // { value: 1, done: false }
it1.next();                             // { value: 2, done: false }

var it2 = foo();
it2.next();                             // { value: 1, done: false }

it1.next();                             // { value: 3, done: false }
```

```
it2.next();                             // { value: 2, done: false }
it2.next();                             // { value: 3, done: false }

it2.next();                             // { value: undefined, done: true }
it1.next();                             // { value: undefined, done: true }
```

Early Abort

Instead of calling `return(..)`, you can call `throw(..)`. Just like `return(x)` is essentially injecting a `return x` into the generator at its current pause point, calling `throw(x)` is essentially like injecting a `throw x` at the pause point.

Other than the exception behavior (we cover what that means to `try` clauses in the next section), `throw(..)` produces the same sort of early completion that aborts the generator's run at its current pause point. For example:

```
function *foo() {
        yield 1;
        yield 2;
        yield 3;
}

var it = foo();

it.next();                              // { value: 1, done: false }

try {
        it.throw( "Oops!" );
}
catch (err) {
        console.log( err );     // Exception: Oops!
}

it.next();                              // { value: undefined, done: true }
```

Because `throw(..)` basically injects a `throw ..` in replacement of the `yield 1` line of the generator, and nothing handles this exception, it immediately propagates back out to the calling code, which handles it with a `try..catch`.

Unlike `return(..)`, the iterator's `throw(..)` method is never called automatically.

Of course, though not shown in the previous snippet, if a `try..finally` clause was waiting inside the generator when you call `throw(..)`, the `finally` clause would be given a chance to complete before the exception is propagated back to the calling code.

Error Handling

As we've already hinted, error handling with generators can be expressed with `try..catch`, which works in both inbound and outbound directions:

```
function *foo() {
        try {
                yield 1;
        }
        catch (err) {
                console.log( err );
        }

        yield 2;

        throw "Hello!";
}

var it = foo();

it.next();                              // { value: 1, done: false }
```

```
try {
        it.throw( "Hi!" );      // Hi!
                                                // { value: 2, done: false }
        it.next();

        console.log( "never gets here" );
}
catch (err) {
        console.log( err );     // Hello!
}
```

Errors can also propagate in both directions through `yield *` delegation:

```
function *foo() {
        try {
                yield 1;
        }
        catch (err) {
                console.log( err );
        }

        yield 2;

        throw "foo: e2";
}

function *bar() {
        try {
                yield *foo();

                console.log( "never gets here" );
        }
        catch (err) {
                console.log( err );
        }
}

var it = bar();

try {
        it.next();                      // { value: 1, done: false }

        it.throw( "e1" );       // e1
                                                // { value: 2, done: false }

        it.next();                      // foo: e2
                                                // { value: undefined, done: true }
}
catch (err) {
        console.log( "never gets here" );
}

it.next();                              // { value: undefined, done: true }
```

When `*foo()` calls `yield 1`, the `1` value passes through `*bar()` untouched, as we've already seen.

But what's most interesting about this snippet is that when `*foo()` calls `throw "foo: e2"`, this error propagates to `*bar()` and is immediately caught by `*bar()`'s `try..catch` block. The error doesn't pass through `*bar()` like the `1` value did.

`*bar()`'s `catch` then does a normal output of `err` (`"foo: e2"`) and then `*bar()` finishes normally, which is why the `{ value: undefined, done: true }` iterator result comes back from `it.next()`.

If `*bar()` didn't have a `try..catch` around the `yield *..` expression, the error would of course propagate all the way out, and on the way through it still would complete (abort) `*bar()`.

Transpiling a Generator

Is it possible to represent a generator's capabilities prior to ES6? It turns out it is, and there are several great tools that do so, including most notably Facebook's Regenerator tool (https://facebook.github.io/regenerator/).

But just to better understand generators, let's try our hand at manually converting. Basically, we're going to create a simple closure-based state machine.

We'll keep our source generator really simple:

```
function *foo() {
	var x = yield 42;
	console.log( x );
}
```

To start, we'll need a function called `foo()` that we can execute, which needs to return an iterator:

```
function foo() {
	// ..

	return {
		next: function(v) {
			// ..
		}

		// we'll skip `return(..)` and `throw(..)`
	};
}
```

Now, we need some inner variable to keep track of where we are in the steps of our "generator"'s logic. We'll call it `state`. There will be three states: `0` initially, `1` while waiting to fulfill the `yield` expression, and `2` once the generator is complete.

Each time `next(..)` is called, we need to process the next step, and then increment `state`. For convenience, we'll put each step into a `case` clause of a `switch` statement, and we'll hold that in an inner function called `nextState(..)` that `next(..)` can call. Also, because `x` is a variable across the overall scope of the "generator," it needs to live outside the `nextState(..)` function.

Here it is all together (obviously somewhat simplified, to keep the conceptual illustration clearer):

```
function foo() {
	function nextState(v) {
		switch (state) {
			case 0:
				state++;

				// the `yield` expression
				return 42;
			case 1:
				state++;

				// `yield` expression fulfilled
				x = v;
				console.log( x );

				// the implicit `return`
				return undefined;

			// no need to handle state `2`
		}
	}

	var state = 0, x;

	return {
```

```
                next: function(v) {
                        var ret = nextState( v );

                        return { value: ret, done: (state == 2) };
                }

                // we'll skip `return(..)` and `throw(..)`
        };
}
```

And finally, let's test our pre-ES6 "generator":

```
var it = foo();

it.next();                              // { value: 42, done: false }

it.next( 10 );                  // 10
                                                // { value: undefined, done: true }
```

Not bad, huh? Hopefully this exercise solidifies in your mind that generators are actually just simple syntax for state machine logic. That makes them widely applicable.

Generator Uses

So, now that we much more deeply understand how generators work, what are they useful for?

We've seen two major patterns:

- *Producing a series of values:* This usage can be simple (e.g., random strings or incremented numbers), or it can represent more structured data access (e.g., iterating over rows returned from a database query).

  Either way, we use the iterator to control a generator so that some logic can be invoked for each call to `next(..)`. Normal iterators on data structures merely pull values without any controlling logic.

- *Queue of tasks to perform serially:* This usage often represents flow control for the steps in an algorithm, where each step requires retrieval of data from some external source. The fulfillment of each piece of data may be immediate, or may be asynchronously delayed.

  From the perspective of the code inside the generator, the details of sync or async at a `yield` point are entirely opaque. Moreover, these details are intentionally abstracted away, such as not to obscure the natural sequential expression of steps with such implementation complications. Abstraction also means the implementations can be swapped/refactored often without touching the code in the generator at all.

When generators are viewed in light of these uses, they become a lot more than just a different or nicer syntax for a manual state machine. They are a powerful abstraction tool for organizing and controlling orderly production and consumption of data.

Modules

I don't think it's an exaggeration to suggest that the single most important code organization pattern in all of JavaScript is, and always has been, the module. For myself, and I think for a large cross-section of the community, the module pattern drives the vast majority of code.

The Old Way

The traditional module pattern is based on an outer function with inner variables and functions, and a returned "public API" with methods that have closure over the inner data and capabilities. It's often expressed like this:

```
function Hello(name) {
        function greeting() {
```

```
                console.log( "Hello " + name + "!" );
        }

        // public API
        return {
                greeting: greeting
        };
}

var me = Hello( "Kyle" );
me.greeting();                  // Hello Kyle!
```

This `Hello(..)` module can produce multiple instances by being called subsequent times. Sometimes, a module is only called for as a singleton (i.e., it just needs one instance), in which case a slight variation on the previous snippet, using an IIFE, is common:

```
var me = (function Hello(name){
        function greeting() {
                console.log( "Hello " + name + "!" );
        }

        // public API
        return {
                greeting: greeting
        };
})( "Kyle" );

me.greeting();                  // Hello Kyle!
```

This pattern is tried and tested. It's also flexible enough to have a wide assortment of variations for a number of different scenarios.

One of the most common is the Asynchronous Module Definition (AMD), and another is the Universal Module Definition (UMD). We won't cover the particulars of these patterns and techniques here, but they're explained extensively in many places online.

Moving Forward

As of ES6, we no longer need to rely on the enclosing function and closure to provide us with module support. ES6 modules have first class syntactic and functional support.

Before we get into the specific syntax, it's important to understand some fairly significant conceptual differences with ES6 modules compared to how you may have dealt with modules in the past:

- ES6 uses file-based modules, meaning one module per file. At this time, there is no standardized way of combining multiple modules into a single file.

  That means that if you are going to load ES6 modules directly into a browser web application, you will be loading them individually, not as a large bundle in a single file as has been common in performance optimization efforts.

  It's expected that the contemporaneous advent of HTTP/2 will significantly mitigate any such performance concerns, as it operates on a persistent socket connection and thus can very efficiently load many smaller files in parallel and interleaved with one another.

- The API of an ES6 module is static. That is, you define statically what all the top-level exports are on your module's public API, and those cannot be amended later.

  Some uses are accustomed to being able to provide dynamic API definitions, where methods can be added/removed/replaced in response to runtime conditions. Either these uses will have to change to fit with ES6 static APIs, or they will have to restrain the dynamic changes to properties/methods of a second-level object.

- ES6 modules are singletons. That is, there's only one instance of the module, which maintains its state. Every time you import that module into another module, you get a reference to the one centralized instance. If you want to be able to

produce multiple module instances, your module will need to provide some sort of factory to do it.

- The properties and methods you expose on a module's public API are not just normal assignments of values or references. They are actual bindings (almost like pointers) to the identifiers in your inner module definition.

  In pre-ES6 modules, if you put a property on your public API that holds a primitive value like a number or string, that property assignment was by value-copy, and any internal update of a corresponding variable would be separate and not affect the public copy on the API object.

  With ES6, exporting a local private variable, even if it currently holds a primitive string/number/etc, exports a binding to the variable. If the module changes the variable's value, the external import binding now resolves to that new value.

- Importing a module is the same thing as statically requesting it to load (if it hasn't already). If you're in a browser, that implies a blocking load over the network. If you're on a server (i.e., Node.js), it's a blocking load from the filesystem.

  However, don't panic about the performance implications. Because ES6 modules have static definitions, the import requirements can be statically scanned, and loads will happen preemptively, even before you've used the module.

  ES6 doesn't actually specify or handle the mechanics of how these load requests work. There's a separate notion of a Module Loader, where each hosting environment (browser, Node.js, etc.) provides a default Loader appropriate to the environment. The importing of a module uses a string value to represent where to get the module (URL, file path, etc.), but this value is opaque in your program and only meaningful to the Loader itself.

  You can define your own custom Loader if you want more fine-grained control than the default Loader affords -- which is basically none, as it's totally hidden from your program's code.

As you can see, ES6 modules will serve the overall use case of organizing code with encapsulation, controlling public APIs, and referencing dependency imports. But they have a very particular way of doing so, and that may or may not fit very closely with how you've already been doing modules for years.

CommonJS

There's a similar, but not fully compatible, module syntax called CommonJS, which is familiar to those in the Node.js ecosystem.

For lack of a more tactful way to say this, in the long run, ES6 modules essentially are bound to supersede all previous formats and standards for modules, even CommonJS, as they are built on syntactic support in the language. This will, in time, inevitably win out as the superior approach, if for no other reason than ubiquity.

We face a fairly long road to get to that point, though. There are literally hundreds of thousands of CommonJS style modules in the server-side JavaScript world, and 10 times that many modules of varying format standards (UMD, AMD, ad hoc) in the browser world. It will take many years for the transitions to make any significant progress.

In the interim, module transpilers/converters will be an absolute necessity. You might as well just get used to that new reality. Whether you author in regular modules, AMD, UMD, CommonJS, or ES6, these tools will have to parse and convert to a format that is suitable for whatever environment your code will run in.

For Node.js, that probably means (for now) that the target is CommonJS. For the browser, it's probably UMD or AMD. Expect lots of flux on this over the next few years as these tools mature and best practices emerge.

From here on out, my best advice on modules is this: whatever format you've been religiously attached to with strong affinity, also develop an appreciation for and understanding of ES6 modules, such as they are, and let your other module tendencies fade. They *are* the future of modules in JS, even if that reality is a bit of a ways off.

The New Way

The two main new keywords that enable ES6 modules are `import` and `export`. There's lots of nuance to the syntax, so let's take a deeper look.

Warning: An important detail that's easy to overlook: both `import` and `export` must always appear in the top-level scope of their respective usage. For example, you cannot put either an `import` or `export` inside an `if` conditional; they must appear

outside of all blocks and functions.

`export`ing API Members

The `export` keyword is either put in front of a declaration, or used as an operator (of sorts) with a special list of bindings to export. Consider:

```
export function foo() {
        // ..
}

export var awesome = 42;

var bar = [1,2,3];
export { bar };
```

Another way of expressing the same exports:

```
function foo() {
        // ..
}

var awesome = 42;
var bar = [1,2,3];

export { foo, awesome, bar };
```

These are all called *named exports*, as you are in effect exporting the name bindings of the variables/functions/etc.

Anything you don't *label* with `export` stays private inside the scope of the module. That is, although something like `var bar = ..` looks like it's declaring at the top-level global scope, the top-level scope is actually the module itself; there is no global scope in modules.

Note: Modules *do* still have access to `window` and all the "globals" that hang off it, just not as lexical top-level scope. However, you really should stay away from the globals in your modules if at all possible.

You can also "rename" (aka alias) a module member during named export:

```
function foo() { .. }

export { foo as bar };
```

When this module is imported, only the `bar` member name is available to import; `foo` stays hidden inside the module.

Module exports are not just normal assignments of values or references, as you're accustomed to with the `=` assignment operator. Actually, when you export something, you're exporting a binding (kinda like a pointer) to that thing (variable, etc.).

Within your module, if you change the value of a variable you already exported a binding to, even if it's already been imported (see the next section), the imported binding will resolve to the current (updated) value.

Consider:

```
var awesome = 42;
export { awesome };

// later
awesome = 100;
```

When this module is imported, regardless of whether that's before or after the `awesome = 100` setting, once that assignment has happened, the imported binding resolves to the `100` value, not `42`.

That's because the binding is, in essence, a reference to, or a pointer to, the `awesome` variable itself, rather than a copy of its value. This is a mostly unprecedented concept for JS introduced with ES6 module bindings.

Though you can clearly use `export` multiple times inside a module's definition, ES6 definitely prefers the approach that a module has a single export, which is known as a *default export*. In the words of some members of the TC39 committee, you're "rewarded with simpler `import` syntax" if you follow that pattern, and conversely "penalized" with more verbose syntax if you don't.

A default export sets a particular exported binding to be the default when importing the module. The name of the binding is literally `default` . As you'll see later, when importing module bindings you can also rename them, as you commonly will with a default export.

There can only be one `default` per module definition. We'll cover `import` in the next section, and you'll see how the `import` syntax is more concise if the module has a default export.

There's a subtle nuance to default export syntax that you should pay close attention to. Compare these two snippets:

```
function foo(..) {
        // ..
}

export default foo;
```

And this one:

```
function foo(..) {
        // ..
}

export { foo as default };
```

In the first snippet, you are exporting a binding to the function expression value at that moment, *not* to the identifier `foo` . In other words, `export default ..` takes an expression. If you later assign `foo` to a different value inside your module, the module import still reveals the function originally exported, not the new value.

By the way, the first snippet could also have been written as:

```
export default function foo(..) {
        // ..
}
```

Warning: Although the `function foo..` part here is technically a function expression, for the purposes of the internal scope of the module, it's treated like a function declaration, in that the `foo` name is bound in the module's top-level scope (often called "hoisting"). The same is true for `export default class Foo..` . However, while you *can* do `export var foo = ..` , you currently cannot do `export default var foo = ..` (or `let` or `const`), in a frustrating case of inconsistency. At the time of this writing, there's already discussion of adding that capability in soon, post-ES6, for consistency sake.

Recall the second snippet again:

```
function foo(..) {
        // ..
}

export { foo as default };
```

In this version of the module export, the default export binding is actually to the `foo` identifier rather than its value, so you get the previously described binding behavior (i.e., if you later change `foo` 's value, the value seen on the import side will also be updated).

Be very careful of this subtle gotcha in default export syntax, especially if your logic calls for export values to be updated. If you never plan to update a default export's value, `export default ..` is fine. If you do plan to update the value, you must use `export { .. as default }`. Either way, make sure to comment your code to explain your intent!

Because there can only be one `default` per module, you may be tempted to design your module with one default export of an object with all your API methods on it, such as:

```
export default {
        foo() { .. },
        bar() { .. },
        ..
};
```

That pattern seems to map closely to how a lot of developers have already structured their pre-ES6 modules, so it seems like a natural approach. Unfortunately, it has some downsides and is officially discouraged.

In particular, the JS engine cannot statically analyze the contents of a plain object, which means it cannot do some optimizations for static `import` performance. The advantage of having each member individually and explicitly exported is that the engine *can* do the static analysis and optimization.

If your API has more than one member already, it seems like these principles -- one default export per module, and all API members as named exports -- are in conflict, doesn't it? But you *can* have a single default export as well as other named exports; they are not mutually exclusive.

So, instead of this (discouraged) pattern:

```
export default function foo() { .. }

foo.bar = function() { .. };
foo.baz = function() { .. };
```

You can do:

```
export default function foo() { .. }

export function bar() { .. }
export function baz() { .. }
```

Note: In this previous snippet, I used the name `foo` for the function that `default` labels. That `foo` name, however, is ignored for the purposes of export -- `default` is actually the exported name. When you import this default binding, you can give it whatever name you want, as you'll see in the next section.

Alternatively, some will prefer:

```
function foo() { .. }
function bar() { .. }
function baz() { .. }

export { foo as default, bar, baz, .. };
```

The effects of mixing default and named exports will be more clear when we cover `import` shortly. But essentially it means that the most concise default import form would only retrieve the `foo()` function. The user could additionally manually list `bar` and `baz` as named imports, if they want them.

You can probably imagine how tedious that's going to be for consumers of your module if you have lots of named export bindings. There is a wildcard import form where you import all of a module's exports within a single namespace object, but there's no way to wildcard import to top-level bindings.

Again, the ES6 module mechanism is intentionally designed to discourage modules with lots of exports; relatively speaking, it's desired that such approaches be a little more difficult, as a sort of social engineering to encourage simple module design in favor of large/complex module design.

I would probably recommend you not mix default export with named exports, especially if you have a large API and refactoring to separate modules isn't practical or desired. In that case, just use all named exports, and document that consumers of your module should probably use the `import * as ..` (namespace import, discussed in the next section) approach to bring the whole API in at once on a single namespace.

We mentioned this earlier, but let's come back to it in more detail. Other than the `export default ...` form that exports an expression value binding, all other export forms are exporting bindings to local identifiers. For those bindings, if you change the value of a variable inside a module after exporting, the external imported binding will access the updated value:

```
var foo = 42;
export { foo as default };

export var bar = "hello world";

foo = 10;
bar = "cool";
```

When you import this module, the `default` and `bar` exports will be bound to the local variables `foo` and `bar`, meaning they will reveal the updated `10` and `"cool"` values. The values at time of export are irrelevant. The values at time of import are irrelevant. The bindings are live links, so all that matters is what the current value is when you access the binding.

Warning: Two-way bindings are not allowed. If you import a `foo` from a module, and try to change the value of your imported `foo` variable, an error will be thrown! We'll revisit that in the next section.

You can also re-export another module's exports, such as:

```
export { foo, bar } from "baz";
export { foo as FOO, bar as BAR } from "baz";
export * from "baz";
```

Those forms are similar to just first importing from the `"baz"` module then listing its members explicitly for export from your module. However, in these forms, the members of the `"baz"` module are never imported to your module's local scope; they sort of pass through untouched.

`import` ing API Members

To import a module, unsurprisingly you use the `import` statement. Just as `export` has several nuanced variations, so does `import`, so spend plenty of time considering the following issues and experimenting with your options.

If you want to import certain specific named members of a module's API into your top-level scope, you use this syntax:

```
import { foo, bar, baz } from "foo";
```

Warning: The `{ .. }` syntax here may look like an object literal, or even an object destructuring syntax. However, its form is special just for modules, so be careful not to confuse it with other `{ .. }` patterns elsewhere.

The `"foo"` string is called a *module specifier*. Because the whole goal is statically analyzable syntax, the module specifier must be a string literal; it cannot be a variable holding the string value.

From the perspective of your ES6 code and the JS engine itself, the contents of this string literal are completely opaque and meaningless. The module loader will interpret this string as an instruction of where to find the desired module, either as a URL path or a local filesystem path.

The `foo`, `bar`, and `baz` identifiers listed must match named exports on the module's API (static analysis and error assertion apply). They are bound as top-level identifiers in your current scope:

```
import { foo } from "foo";

foo();
```

You can rename the bound identifiers imported, as:

```
import { foo as theFooFunc } from "foo";

theFooFunc();
```

If the module has just a default export that you want to import and bind to an identifier, you can opt to skip the `{ .. }` surrounding syntax for that binding. The `import` in this preferred case gets the nicest and most concise of the `import` syntax forms:

```
import foo from "foo";

// or:
import { default as foo } from "foo";
```

Note: As explained in the previous section, the `default` keyword in a module's `export` specifies a named export where the name is actually `default`, as is illustrated by the second more verbose syntax option. The renaming from `default` to, in this case, `foo`, is explicit in the latter syntax and is identical yet implicit in the former syntax.

You can also import a default export along with other named exports, if the module has such a definition. Recall this module definition from earlier:

```
export default function foo() { .. }

export function bar() { .. }
export function baz() { .. }
```

To import that module's default export and its two named exports:

```
import FOOFN, { bar, baz as BAZ } from "foo";

FOOFN();
bar();
BAZ();
```

The strongly suggested approach from ES6's module philosophy is that you only import the specific bindings from a module that you need. If a module provides 10 API methods, but you only need two of them, some believe it wasteful to bring in the entire set of API bindings.

One benefit, besides code being more explicit, is that narrow imports make static analysis and error detection (accidentally using the wrong binding name, for instance) more robust.

Of course, that's just the standard position influenced by ES6 design philosophy; there's nothing that requires adherence to that approach.

Many developers would be quick to point out that such approaches can be more tedious, requiring you to regularly revisit and update your `import` statement(s) each time you realize you need something else from a module. The trade-off is in exchange for convenience.

In that light, the preference might be to import everything from the module into a single namespace, rather than importing individual members, each directly into the scope. Fortunately, the `import` statement has a syntax variation that can support this style of module consumption, called *namespace import*.

Consider a `"foo"` module exported as:

```
export function bar() { .. }
export var x = 42;
export function baz() { .. }
```

You can import that entire API to a single module namespace binding:

```
import * as foo from "foo";

foo.bar();
foo.x;                  // 42
foo.baz();
```

Note: The `* as ..` clause requires the `*` wildcard. In other words, you cannot do something like `import { bar, x } as foo from "foo"` to bring in only part of the API but still bind to the `foo` namespace. I would have liked something like that, but for ES6 it's all or nothing with the namespace import.

If the module you're importing with `* as ..` has a default export, it is named `default` in the namespace specified. You can additionally name the default import outside of the namespace binding, as a top-level identifier. Consider a `"world"` module exported as:

```
export default function foo() { .. }
export function bar() { .. }
export function baz() { .. }
```

And this `import`:

```
import foofn, * as hello from "world";

foofn();
hello.default();
hello.bar();
hello.baz();
```

While this syntax is valid, it can be rather confusing that one method of the module (the default export) is bound at the top-level of your scope, whereas the rest of the named exports (and one called `default`) are bound as properties on a differently named (`hello`) identifier namespace.

As I mentioned earlier, my suggestion would be to avoid designing your module exports in this way, to reduce the chances that your module's users will suffer these strange quirks.

All imported bindings are immutable and/or read-only. Consider the previous import; all of these subsequent assignment attempts will throw `TypeError`s:

```
import foofn, * as hello from "world";

foofn = 42;                     // (runtime) TypeError!
hello.default = 42;     // (runtime) TypeError!
hello.bar = 42;         // (runtime) TypeError!
hello.baz = 42;         // (runtime) TypeError!
```

Recall earlier in the "`export`ing API Members" section that we talked about how the `bar` and `baz` bindings are bound to the actual identifiers inside the `"world"` module. That means if the module changes those values, `hello.bar` and `hello.baz` now reference the updated values.

But the immutable/read-only nature of your local imported bindings enforces that you cannot change them from the imported bindings, hence the `TypeError`s. That's pretty important, because without those protections, your changes would end up affecting all other consumers of the module (remember: singleton), which could create some very surprising side effects!

Moreover, though a module *can* change its API members from the inside, you should be very cautious of intentionally designing your modules in that fashion. ES6 modules are *intended* to be static, so deviations from that principle should be rare and should be carefully and verbosely documented.

Warning: There are module design philosophies where you actually intend to let a consumer change the value of a property on your API, or module APIs are designed to be "extended" by having other "plug-ins" add to the API namespace. As we just asserted, ES6 module APIs should be thought of and designed as static and unchangeable, which strongly restricts and discourages these alternative module design patterns. You can get around these limitations by exporting a plain object, which of course can then be changed at will. But be careful and think twice before going down that road.

Declarations that occur as a result of an `import` are "hoisted" (see the *Scope & Closures* title of this series). Consider:

```
foo();

import { foo } from "foo";
```

`foo()` can run because not only did the static resolution of the `import ..` statement figure out what `foo` is during compilation, but it also "hoisted" the declaration to the top of the module's scope, thus making it available throughout the module.

Finally, the most basic form of the `import` looks like this:

```
import "foo";
```

This form does not actually import any of the module's bindings into your scope. It loads (if not already loaded), compiles (if not already compiled), and evaluates (if not already run) the `"foo"` module.

In general, that sort of import is probably not going to be terribly useful. There may be niche cases where a module's definition has side effects (such as assigning things to the `window` /global object). You could also envision using `import "foo"` as a sort of preload for a module that may be needed later.

Circular Module Dependency

A imports B. B imports A. How does this actually work?

I'll state off the bat that designing systems with intentional circular dependency is generally something I try to avoid. That having been said, I recognize there are reasons people do this and it can solve some sticky design situations.

Let's consider how ES6 handles this. First, module `"A"`:

```
import bar from "B";

export default function foo(x) {
        if (x > 10) return bar( x - 1 );
        return x * 2;
}
```

Now, module `"B"`:

```
import foo from "A";

export default function bar(y) {
        if (y > 5) return foo( y / 2 );
        return y * 3;
}
```

These two functions, `foo(..)` and `bar(..)`, would work as standard function declarations if they were in the same scope, because the declarations are "hoisted" to the whole scope and thus available to each other regardless of authoring order.

With modules, you have declarations in entirely different scopes, so ES6 has to do extra work to help make these circular references work.

In a rough conceptual sense, this is how circular `import` dependencies are validated and resolved:

- If the `"A"` module is loaded first, the first step is to scan the file and analyze all the exports, so it can register all those bindings available for import. Then it processes the `import .. from "B"`, which signals that it needs to go fetch `"B"`.
- Once the engine loads `"B"`, it does the same analysis of its export bindings. When it sees the `import .. from "A"`, it knows the API of `"A"` already, so it can verify the `import` is valid. Now that it knows the `"B"` API, it can also validate the `import .. from "B"` in the waiting `"A"` module.

In essence, the mutual imports, along with the static verification that's done to validate both `import` statements, virtually composes the two separate module scopes (via the bindings), such that `foo(..)` can call `bar(..)` and vice versa. This is symmetric to if they had originally been declared in the same scope.

Now let's try using the two modules together. First, we'll try `foo(..)`:

```
import foo from "foo";
foo( 25 );                              // 11
```

Or we can try `bar(..)`:

```
import bar from "bar";
bar( 25 );                              // 11.5
```

By the time either the `foo(25)` or `bar(25)` calls are executed, all the analysis/compilation of all modules has completed. That means `foo(..)` internally knows directly about `bar(..)` and `bar(..)` internally knows directly about `foo(..)`.

If all we need is to interact with `foo(..)`, then we only need to import the `"foo"` module. Likewise with `bar(..)` and the `"bar"` module.

Of course, we *can* import and use both of them if we want to:

```
import foo from "foo";
import bar from "bar";

foo( 25 );                              // 11
bar( 25 );                              // 11.5
```

The static loading semantics of the `import` statement mean that a `"foo"` and `"bar"` that mutually depend on each other via `import` will ensure that both are loaded, parsed, and compiled before either of them runs. So their circular dependency is statically resolved and this works as you'd expect.

Module Loading

We asserted at the beginning of this "Modules" section that the `import` statement uses a separate mechanism, provided by the hosting environment (browser, Node.js, etc.), to actually resolve the module specifier string into some useful instruction for finding and loading the desired module. That mechanism is the system *Module Loader*.

The default module loader provided by the environment will interpret a module specifier as a URL if in the browser, and (generally) as a local filesystem path if on a server such as Node.js. The default behavior is to assume the loaded file is authored in the ES6 standard module format.

Moreover, you will be able to load a module into the browser via an HTML tag, similar to how current script programs are loaded. At the time of this writing, it's not fully clear if this tag will be `<script type="module">` or `<module>`. ES6 doesn't control that decision, but discussions in the appropriate standards bodies are already well along in parallel of ES6.

Whatever the tag looks like, you can be sure that under the covers it will use the default loader (or a customized one you've pre-specified, as we'll discuss in the next section).

Just like the tag you'll use in markup, the module loader itself is not specified by ES6. It is a separate, parallel standard (http://whatwg.github.io/loader/) controlled currently by the WHATWG browser standards group.

At the time of this writing, the following discussions reflect an early pass at the API design, and things are likely to change.

Loading Modules Outside of Modules

One use for interacting directly with the module loader is if a non-module needs to load a module. Consider:

```
// normal script loaded in browser via `<script>`,
// `import` is illegal here

Reflect.Loader.import( "foo" ) // returns a promise for `"foo"`
.then( function(foo){
        foo.bar();
} );
```

The `Reflect.Loader.import(..)` utility imports the entire module onto the named parameter (as a namespace), just like the `import * as foo ..` namespace import we discussed earlier.

Note: The `Reflect.Loader.import(..)` utility returns a promise that is fulfilled once the module is ready. To import multiple modules, you can compose promises from multiple `Reflect.Loader.import(..)` calls using `Promise.all([ .. ])`. For more information about Promises, see "Promises" in Chapter 4.

You can also use `Reflect.Loader.import(..)` in a real module to dynamically/conditionally load a module, where `import` itself would not work. You might, for instance, choose to load a module containing a polyfill for some ES7+ feature if a feature test reveals it's not defined by the current engine.

For performance reasons, you'll want to avoid dynamic loading whenever possible, as it hampers the ability of the JS engine to fire off early fetches from its static analysis.

Customized Loading

Another use for directly interacting with the module loader is if you want to customize its behavior through configuration or even redefinition.

At the time of this writing, there's a polyfill for the module loader API being developed (https://github.com/ModuleLoader/es6-module-loader). While details are scarce and highly subject to change, we can explore what possibilities may eventually land.

The `Reflect.Loader.import(..)` call may support a second argument for specifying various options to customize the import/load task. For example:

```
Reflect.Loader.import( "foo", { address: "/path/to/foo.js" } )
.then( function(foo){
        // ..
} )
```

It's also expected that a customization will be provided (through some means) for hooking into the process of loading a module, where a translation/transpilation could occur after load but before the engine compiles the module.

For example, you could load something that's not already an ES6-compliant module format (e.g., CoffeeScript, TypeScript, CommonJS, AMD). Your translation step could then convert it to an ES6-compliant module for the engine to then process.

Classes

From nearly the beginning of JavaScript, syntax and development patterns have all strived (read: struggled) to put on a facade of supporting class-oriented development. With things like `new` and `instanceof` and a `.constructor` property, who couldn't help but be teased that JS had classes hidden somewhere inside its prototype system?

Of course, JS "classes" aren't nearly the same as classical classes. The differences are well documented, so I won't belabor that point any further here.

Note: To learn more about the patterns used in JS to fake "classes," and an alternative view of prototypes called "delegation," see the second half of the *this & Object Prototypes* title of this series.

`class`

Although JS's prototype mechanism doesn't work like traditional classes, that doesn't stop the strong tide of demand on the language to extend the syntactic sugar so that expressing "classes" looks more like real classes. Enter the ES6 `class` keyword and its associated mechanism.

This feature is the result of a highly contentious and drawn-out debate, and represents a smaller subset compromise from several strongly opposed views on how to approach JS classes. Most developers who want full classes in JS will find parts of the new syntax quite inviting, but will find important bits still missing. Don't worry, though. TC39 is already working on additional features to augment classes in the post-ES6 timeframe.

At the heart of the new ES6 class mechanism is the `class` keyword, which identifies a *block* where the contents define the members of a function's prototype. Consider:

```
class Foo {
	constructor(a,b) {
		this.x = a;
		this.y = b;
	}

	gimmeXY() {
		return this.x * this.y;
	}
}
```

Some things to note:

- `class Foo` implies creating a (special) function of the name `Foo`, much like you did pre-ES6.
- `constructor(..)` identifies the signature of that `Foo(..)` function, as well as its body contents.
- Class methods use the same "concise method" syntax available to object literals, as discussed in Chapter 2. This also includes the concise generator form as discussed earlier in this chapter, as well as the ES5 getter/setter syntax. However, class methods are non-enumerable whereas object methods are by default enumerable.
- Unlike object literals, there are no commas separating members in a `class` body! In fact, they're not even allowed.

The `class` syntax definition in the previous snippet can be roughly thought of as this pre-ES6 equivalent, which probably will look fairly familiar to those who've done prototype-style coding before:

```
function Foo(a,b) {
	this.x = a;
	this.y = b;
}

Foo.prototype.gimmeXY = function() {
	return this.x * this.y;
}
```

In either the pre-ES6 form or the new ES6 `class` form, this "class" can now be instantiated and used just as you'd expect:

```
var f = new Foo( 5, 15 );

f.x;                                    // 5
f.y;                                    // 15
f.gimmeXY();                    // 75
```

Caution! Though `class Foo` seems much like `function Foo()`, there are important differences:

- A `Foo(..)` call of `class Foo` *must* be made with `new`, as the pre-ES6 option of `Foo.call( obj )` will *not* work.
- While `function Foo` is "hoisted" (see the *Scope & Closures* title of this series), `class Foo` is not; the `extends ..` clause specifies an expression that cannot be "hoisted." So, you must declare a `class` before you can instantiate it.
- `class Foo` in the top global scope creates a lexical `Foo` identifier in that scope, but unlike `function Foo` does not create a global object property of that name.

The established `instanceof` operator still works with ES6 classes, because `class` just creates a constructor function of the same name. However, ES6 introduces a way to customize how `instanceof` works, using `Symbol.hasInstance` (see "Well-Known Symbols" in Chapter 7).

Another way of thinking about `class`, which I find more convenient, is as a *macro* that is used to automatically populate a `prototype` object. Optionally, it also wires up the `[[Prototype]]` relationship if using `extends` (see the next section).

An ES6 `class` isn't really an entity itself, but a meta concept that wraps around other concrete entities, such as functions and properties, and ties them together.

Tip: In addition to the declaration form, a `class` can also be an expression, as in: `var x = class Y { .. }`. This is primarily useful for passing a class definition (technically, the constructor itself) as a function argument or assigning it to an object property.

`extends` and `super`

ES6 classes also have syntactic sugar for establishing the `[[Prototype]]` delegation link between two function prototypes -- commonly mislabeled "inheritance" or confusingly labeled "prototype inheritance" -- using the class-oriented familiar terminology `extends`:

```
class Bar extends Foo {
        constructor(a,b,c) {
                super( a, b );
                this.z = c;
        }

        gimmeXYZ() {
                return super.gimmeXY() * this.z;
        }
}

var b = new Bar( 5, 15, 25 );

b.x;                                            // 5
b.y;                                            // 15
b.z;                                            // 25
b.gimmeXYZ();                           // 1875
```

A significant new addition is `super`, which is actually something not directly possible pre-ES6 (without some unfortunate hack trade-offs). In the constructor, `super` automatically refers to the "parent constructor," which in the previous example is `Foo(..)`. In a method, it refers to the "parent object," such that you can then make a property/method access off it, such as `super.gimmeXY()`.

`Bar extends Foo` of course means to link the `[[Prototype]]` of `Bar.prototype` to `Foo.prototype`. So, `super` in a method like `gimmeXYZ()` specifically means `Foo.prototype`, whereas `super` means `Foo` when used in the `Bar` constructor.

Note: `super` is not limited to `class` declarations. It also works in object literals, in much the same way we're discussing here. See "Object `super`" in Chapter 2 for more information.

There Be `super` Dragons

It is not insignificant to note that `super` behaves differently depending on where it appears. In fairness, most of the time, that won't be a problem. But surprises await if you deviate from a narrow norm.

There may be cases where in the constructor you would want to reference the `Foo.prototype`, such as to directly access one of its properties/methods. However, `super` in the constructor cannot be used in that way; `super.prototype` will not work. `super(..)` means roughly to call `new Foo(..)`, but isn't actually a usable reference to `Foo` itself.

Symmetrically, you may want to reference the `Foo(..)` function from inside a non-constructor method. `super.constructor` will point at `Foo(..)` the function, but beware that this function can *only* be invoked with `new`. `new super.constructor(..)` would be valid, but it wouldn't be terribly useful in most cases, because you can't make that call use or reference the current `this` object context, which is likely what you'd want.

Also, `super` looks like it might be driven by a function's context just like `this` -- that is, that they'd both be dynamically bound. However, `super` is not dynamic like `this` is. When a constructor or method makes a `super` reference inside it at declaration time (in the `class` body), that `super` is statically bound to that specific class hierarchy, and cannot be overridden (at least in ES6).

What does that mean? It means that if you're in the habit of taking a method from one "class" and "borrowing" it for another class by overriding its `this`, say with `call(..)` or `apply(..)`, that may very well create surprises if the method you're borrowing has a `super` in it. Consider this class hierarchy:

```
class ParentA {
        constructor() { this.id = "a"; }
        foo() { console.log( "ParentA:", this.id ); }
}

class ParentB {
        constructor() { this.id = "b"; }
        foo() { console.log( "ParentB:", this.id ); }
}

class ChildA extends ParentA {
        foo() {
                super.foo();
                console.log( "ChildA:", this.id );
        }
}

class ChildB extends ParentB {
        foo() {
                super.foo();
                console.log( "ChildB:", this.id );
        }
}

var a = new ChildA();
a.foo();                                        // ParentA: a
                                                        // ChildA: a
var b = new ChildB();           // ParentB: b
b.foo();                                        // ChildB: b
```

All seems fairly natural and expected in this previous snippet. However, if you try to borrow `b.foo()` and use it in the context of `a` -- by virtue of dynamic `this` binding, such borrowing is quite common and used in many different ways, including mixins most notably -- you may find this result an ugly surprise:

```
// borrow `b.foo()` to use in `a` context
b.foo.call( a );                        // ParentB: a
                                                // ChildB: a
```

As you can see, the `this.id` reference was dynamically rebound so that `: a` is reported in both cases instead of `: b`. But `b.foo()`'s `super.foo()` reference wasn't dynamically rebound, so it still reported `ParentB` instead of the expected `ParentA`.

Because `b.foo()` references `super`, it is statically bound to the `ChildB` / `ParentB` hierarchy and cannot be used against the `ChildA` / `ParentA` hierarchy. There is no ES6 solution to this limitation.

`super` seems to work intuitively if you have a static class hierarchy with no cross-pollination. But in all fairness, one of the main benefits of doing `this`-aware coding is exactly that sort of flexibility. Simply, `class` + `super` requires you to avoid such techniques.

The choice boils down to narrowing your object design to these static hierarchies -- `class`, `extends`, and `super` will be quite nice -- or dropping all attempts to "fake" classes and instead embrace dynamic and flexible, classless objects and `[[Prototype]]` delegation (see the *this & Object Prototypes* title of this series).

Subclass Constructor

Constructors are not required for classes or subclasses; a default constructor is substituted in both cases if omitted. However, the default substituted constructor is different for a direct class versus an extended class.

Specifically, the default subclass constructor automatically calls the parent constructor, and passes along any arguments. In other words, you could think of the default subclass constructor sort of like this:

```
constructor(...args) {
        super(...args);
}
```

This is an important detail to note. Not all class languages have the subclass constructor automatically call the parent constructor. C++ does, but Java does not. But more importantly, in pre-ES6 classes, such automatic "parent constructor" calling does not happen. Be careful when converting to ES6 `class` if you've been relying on such calls *not* happening.

Another perhaps surprising deviation/limitation of ES6 subclass constructors: in a constructor of a subclass, you cannot access `this` until `super(..)` has been called. The reason is nuanced and complicated, but it boils down to the fact that the parent constructor is actually the one creating/initializing your instance's `this`. Pre-ES6, it works oppositely; the `this` object is created by the "subclass constructor," and then you call a "parent constructor" with the context of the "subclass" `this`.

Let's illustrate. This works pre-ES6:

```
function Foo() {
        this.a = 1;
}

function Bar() {
        this.b = 2;
        Foo.call( this );
}

// `Bar` "extends" `Foo`
Bar.prototype = Object.create( Foo.prototype );
```

But this ES6 equivalent is not allowed:

```
class Foo {
        constructor() { this.a = 1; }
}

class Bar extends Foo {
        constructor() {
                this.b = 2;                     // not allowed before `super()`
                super();                        // to fix swap these two statements
        }
}
```

In this case, the fix is simple. Just swap the two statements in the subclass `Bar` constructor. However, if you've been relying pre-ES6 on being able to skip calling the "parent constructor," beware because that won't be allowed anymore.

`extend`ing Natives

One of the most heralded benefits to the new `class` and `extend` design is the ability to (finally!) subclass the built-in natives, like `Array`. Consider:

```
class MyCoolArray extends Array {
        first() { return this[0]; }
        last() { return this[this.length - 1]; }
}

var a = new MyCoolArray( 1, 2, 3 );

a.length;                                       // 3
a;                                                      // [1,2,3]

a.first();                                      // 1
a.last();                                       // 3
```

Prior to ES6, a fake "subclass" of `Array` using manual object creation and linking to `Array.prototype` only partially worked. It missed out on the special behaviors of a real array, such as the automatically updating `length` property. ES6 subclasses should fully work with "inherited" and augmented behaviors as expected!

Another common pre-ES6 "subclass" limitation is with the `Error` object, in creating custom error "subclasses." When genuine `Error` objects are created, they automatically capture special `stack` information, including the line number and file where the error is created. Pre-ES6 custom error "subclasses" have no such special behavior, which severely limits their usefulness.

ES6 to the rescue:

```
class Oops extends Error {
        constructor(reason) {
                super(reason);
                this.oops = reason;
        }
}

// later:
var ouch = new Oops( "I messed up!" );
throw ouch;
```

The `ouch` custom error object in this previous snippet will behave like any other genuine error object, including capturing `stack`. That's a big improvement!

`new.target`

ES6 introduces a new concept called a *meta property* (see Chapter 7), in the form of `new.target`.

If that looks strange, it is; pairing a keyword with a `.` and a property name is definitely an out-of-the-ordinary pattern for JS.

`new.target` is a new "magical" value available in all functions, though in normal functions it will always be `undefined`. In any constructor, `new.target` always points at the constructor that `new` actually directly invoked, even if the constructor is in a parent class and was delegated to by a `super(..)` call from a child constructor. Consider:

```
class Foo {
        constructor() {
                console.log( "Foo: ", new.target.name );
        }
}

class Bar extends Foo {
        constructor() {
                super();
                console.log( "Bar: ", new.target.name );
        }
        baz() {
                console.log( "baz: ", new.target );
```

```
        }
}

var a = new Foo();
// Foo: Foo

var b = new Bar();
// Foo: Bar   <-- respects the `new` call-site
// Bar: Bar

b.baz();
// baz: undefined
```

The `new.target` meta property doesn't have much purpose in class constructors, except accessing a static property/method (see the next section).

If `new.target` is `undefined`, you know the function was not called with `new`. You can then force a `new` invocation if that's necessary.

`static`

When a subclass `Bar` extends a parent class `Foo`, we already observed that `Bar.prototype` is `[[Prototype]]`-linked to `Foo.prototype`. But additionally, `Bar()` is `[[Prototype]]`-linked to `Foo()`. That part may not have such an obvious reasoning.

However, it's quite useful in the case where you declare `static` methods (not just properties) for a class, as these are added directly to that class's function object, not to the function object's `prototype` object. Consider:

```
class Foo {
        static cool() { console.log( "cool" ); }
        wow() { console.log( "wow" ); }
}

class Bar extends Foo {
        static awesome() {
                super.cool();
                console.log( "awesome" );
        }
        neat() {
                super.wow();
                console.log( "neat" );
        }
}

Foo.cool();                                     // "cool"
Bar.cool();                                     // "cool"
Bar.awesome();                          // "cool"
                                                        // "awesome"

var b = new Bar();
b.neat();                                       // "wow"
                                                        // "neat"

b.awesome;                                      // undefined
b.cool;                                         // undefined
```

Be careful not to get confused that `static` members are on the class's prototype chain. They're actually on the dual/parallel chain between the function constructors.

`Symbol.species` Constructor Getter

One place where `static` can be useful is in setting the `Symbol.species` getter (known internally in the specification as `@@species`) for a derived (child) class. This capability allows a child class to signal to a parent class what constructor should be used -- when not intending the child class's constructor itself -- if any parent class method needs to vend a new instance.

For example, many methods on `Array` create and return a new `Array` instance. If you define a derived class from `Array`, but you want those methods to continue to vend actual `Array` instances instead of from your derived class, this works:

```
class MyCoolArray extends Array {
	// force `species` to be parent constructor
	static get [Symbol.species]() { return Array; }
}

var a = new MyCoolArray( 1, 2, 3 ),
	b = a.map( function(v){ return v * 2; } );

b instanceof MyCoolArray;	// false
b instanceof Array;			// true
```

To illustrate how a parent class method can use a child's species declaration somewhat like `Array#map(..)` is doing, consider:

```
class Foo {
	// defer `species` to derived constructor
	static get [Symbol.species]() { return this; }
	spawn() {
		return new this.constructor[Symbol.species]();
	}
}

class Bar extends Foo {
	// force `species` to be parent constructor
	static get [Symbol.species]() { return Foo; }
}

var a = new Foo();
var b = a.spawn();
b instanceof Foo;					// true

var x = new Bar();
var y = x.spawn();
y instanceof Bar;					// false
y instanceof Foo;					// true
```

The parent class `Symbol.species` does `return this` to defer to any derived class, as you'd normally expect. `Bar` then overrides to manually declare `Foo` to be used for such instance creation. Of course, a derived class can still vend instances of itself using `new this.constructor(..)`.

Review

ES6 introduces several new features that aid in code organization:

- Iterators provide sequential access to data or operations. They can be consumed by new language features like `for..of` and `...`.
- Generators are locally pause/resume capable functions controlled by an iterator. They can be used to programmatically (and interactively, through `yield` / `next(..)` message passing) *generate* values to be consumed via iteration.
- Modules allow private encapsulation of implementation details with a publicly exported API. Module definitions are file-based, singleton instances, and statically resolved at compile time.
- Classes provide cleaner syntax around prototype-based coding. The addition of `super` also solves tricky issues with relative references in the `[[Prototype]]` chain.

These new tools should be your first stop when trying to improve the architecture of your JS projects by embracing ES6.