You Don't Know JS: Async & Performance

Chapter 4: Generators

In Chapter 2, we identified two key drawbacks to expressing async flow control with callbacks:

- Callback-based async doesn't fit how our brain plans out steps of a task.
- Callbacks aren't trustable or composable because of *inversion of control*.

In Chapter 3, we detailed how Promises uninvert the *inversion of control* of callbacks, restoring trustability/composability.

Now we turn our attention to expressing async flow control in a sequential, synchronous-looking fashion. The "magic" that makes it possible is ES6 **generators**.

Breaking Run-to-Completion

In Chapter 1, we explained an expectation that JS developers almost universally rely on in their code: once a function starts executing, it runs until it completes, and no other code can interrupt and run in between.

As bizarre as it may seem, ES6 introduces a new type of function that does not behave with the run-to-completion behavior. This new type of function is called a "generator."

To understand the implications, let's consider this example:

```
var x = 1;

function foo() {
        x++;
        bar();                          // <-- what about this line?
        console.log( "x:", x );
}

function bar() {
        x++;
}

foo();                                  // x: 3
```

In this example, we know for sure that `bar()` runs in between `x++` and `console.log(x)`. But what if `bar()` wasn't there? Obviously, the result would be `2` instead of `3`.
Now let's twist your brain. What if `bar()` wasn't present, but it could still somehow run between the `x++` and `console.log(x)` statements? How would that be possible?
In **preemptive** multithreaded languages, it would essentially be possible for `bar()` to "interrupt" and run at exactly the right moment between those two statements. But JS is not preemptive, nor is it

(currently) multithreaded. And yet, a **cooperative** form of this "interruption" (concurrency) is possible, if `foo()` itself could somehow indicate a "pause" at that part in the code.
Note: I use the word "cooperative" not only because of the connection to classical concurrency terminology (see Chapter 1), but because as you'll see in the next snippet, the ES6 syntax for indicating a pause point in code is `yield` -- suggesting a politely *cooperative* yielding of control.
Here's the ES6 code to accomplish such cooperative concurrency:

```
var x = 1;

function *foo() {
        x++;
        yield; // pause!
        console.log( "x:", x );
}

function bar() {
        x++;
}
```

Note: You will likely see most other JS documentation/code that will format a generator declaration as `function* foo() { .. }` instead of as I've done here with `function *foo() { .. }` -- the only difference being the stylistic positioning of the `*`. The two forms are functionally/syntactically identical, as is a third `function*foo() { .. }` (no space) form. There are arguments for both styles, but I basically prefer `function *foo..` because it then matches when I reference a generator in writing with `*foo()`. If I said only `foo()`, you wouldn't know as clearly if I was talking about a generator or a regular function. It's purely a stylistic preference.
Now, how can we run the code in that previous snippet such that `bar()` executes at the point of the `yield` inside of `*foo()`?

```
// construct an iterator `it` to control the generator
var it = foo();

// start `foo()` here!
it.next();
x;                                      // 2
bar();
x;                                      // 3
it.next();                          // x: 3
```

OK, there's quite a bit of new and potentially confusing stuff in those two code snippets, so we've got plenty to wade through. But before we explain the different mechanics/syntax with ES6 generators, let's walk through the behavior flow:

1. The `it = foo()` operation does *not* execute the `*foo()` generator yet, but it merely constructs an *iterator* that will control its execution. More on *iterators* in a bit.
2. The first `it.next()` starts the `*foo()` generator, and runs the `x++` on the first line of `*foo()`.
3. `*foo()` pauses at the `yield` statement, at which point that first `it.next()` call finishes. At the moment, `*foo()` is still running and active, but it's in a paused state.
4. We inspect the value of `x`, and it's now `2`.
5. We call `bar()`, which increments `x` again with `x++`.
6. We inspect the value of `x` again, and it's now `3`.

7. The final `it.next()` call resumes the `*foo()` generator from where it was paused, and runs the `console.log(..)` statement, which uses the current value of `x` of `3`.

Clearly, `*foo()` started, but did *not* run-to-completion -- it paused at the `yield`. We resumed `*foo()` later, and let it finish, but that wasn't even required.
So, a generator is a special kind of function that can start and stop one or more times, and doesn't necessarily ever have to finish. While it won't be terribly obvious yet why that's so powerful, as we go throughout the rest of this chapter, that will be one of the fundamental building blocks we use to construct generators-as-async-flow-control as a pattern for our code.

Input and Output

A generator function is a special function with the new processing model we just alluded to. But it's still a function, which means it still has some basic tenets that haven't changed -- namely, that it still accepts arguments (aka "input"), and that it can still return a value (aka "output"):

```
function *foo(x,y) {
        return x * y;
}

var it = foo( 6, 7 );

var res = it.next();

res.value;              // 42
```

We pass in the arguments `6` and `7` to `*foo(..)` as the parameters `x` and `y`, respectively.
And `*foo(..)` returns the value `42` back to the calling code.
We now see a difference with how the generator is invoked compared to a normal function. `foo(6,7)` obviously looks familiar. But subtly, the `*foo(..)` generator hasn't actually run yet as it would have with a function.
Instead, we're just creating an *iterator* object, which we assign to the variable `it`, to control the `*foo(..)` generator. Then we call `it.next()`, which instructs the `*foo(..)` generator to advance from its current location, stopping either at the next `yield` or end of the generator.
The result of that `next(..)` call is an object with a `value` property on it holding whatever value (if anything) was returned from `*foo(..)`. In other words, `yield` caused a value to be sent out from the generator during the middle of its execution, kind of like an intermediate `return`.
Again, it won't be obvious yet why we need this whole indirect *iterator* object to control the generator. We'll get there, I *promise*.

Iteration Messaging

In addition to generators accepting arguments and having return values, there's even more powerful and compelling input/output messaging capability built into them, via `yield` and `next(..)`.
Consider:

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

```
        return y;
}

var it = foo( 6 );

// start `foo(..)`
it.next();

var res = it.next( 7 );

res.value;              // 42
```

First, we pass in `6` as the parameter `x`. Then we call `it.next()`, and it starts up `*foo(..)`.
Inside `*foo(..)`, the `var y = x ..` statement starts to be processed, but then it runs across a `yield` expression. At that point, it pauses `*foo(..)` (in the middle of the assignment statement!), and essentially requests the calling code to provide a result value for the `yield` expression. Next, we call `it.next( 7 )`, which is passing the `7` value back in to *be* that result of the paused `yield` expression.
So, at this point, the assignment statement is essentially `var y = 6 * 7`. Now, `return y` returns that `42` value back as the result of the `it.next( 7 )` call.
Notice something very important but also easily confusing, even to seasoned JS developers: depending on your perspective, there's a mismatch between the `yield` and the `next(..)` call. In general, you're going to have one more `next(..)` call than you have `yield` statements -- the preceding snippet has one `yield` and two `next(..)` calls.
Why the mismatch?

Because the first `next(..)` always starts a generator, and runs to the first `yield`. But it's the second `next(..)` call that fulfills the first paused `yield` expression, and the third `next(..)` would fulfill the second `yield`, and so on.

Tale of Two Questions

Actually, which code you're thinking about primarily will affect whether there's a perceived mismatch or not.

Consider only the generator code:

```
var y = x * (yield);
return y;
```

This **first** `yield` is basically *asking a question*: "What value should I insert here?"
Who's going to answer that question? Well, the **first** `next()` has already run to get the generator up to this point, so obviously *it* can't answer the question. So, the **second** `next(..)` call must answer the question *posed* by the **first** `yield`.
See the mismatch -- second-to-first?

But let's flip our perspective. Let's look at it not from the generator's point of view, but from the iterator's point of view.

To properly illustrate this perspective, we also need to explain that messages can go in both directions -- `yield ..` as an expression can send out messages in response to `next(..)` calls, and `next(..)` can send values to a paused `yield` expression. Consider this slightly adjusted code:

```
function *foo(x) {
        var y = x * (yield "Hello");     // <-- yield a value!
        return y;
}

var it = foo( 6 );

var res = it.next();     // first `next()`, don't pass anything
res.value;                               // "Hello"

res = it.next( 7 );              // pass `7` to waiting `yield`
res.value;                               // 42
```

`yield ..` and `next(..)` pair together as a two-way message passing system **during the execution of the generator**.

So, looking only at the *iterator* code:

```
var res = it.next();     // first `next()`, don't pass anything
res.value;                               // "Hello"

res = it.next( 7 );              // pass `7` to waiting `yield`
res.value;                               // 42
```

Note: We don't pass a value to the first `next()` call, and that's on purpose. Only a paused `yield` could accept such a value passed by a `next(..)`, and at the beginning of the generator when we call the first `next()`, there **is no paused `yield`** to accept such a value. The specification and all compliant browsers just silently **discard** anything passed to the first `next()`. It's still a bad idea to pass a value, as you're just creating silently "failing" code that's confusing. So, always start a generator with an argument-free `next()`.

The first `next()` call (with nothing passed to it) is basically *asking a question*: "What *next* value does the `*foo(..)` generator have to give me?" And who answers this question? The first `yield "hello"` expression.

See? No mismatch there.

Depending on *who* you think about asking the question, there is either a mismatch between the `yield` and `next(..)` calls, or not.

But wait! There's still an extra `next()` compared to the number of `yield` statements. So, that final `it.next(7)` call is again asking the question about what *next* value the generator will produce. But there's no more `yield` statements left to answer, is there? So who answers?

The `return` statement answers the question!

And if there **is no `return`** in your generator -- `return` is certainly not any more required in generators than in regular functions -- there's always an assumed/implicit `return;` (aka `return undefined;`), which serves the purpose of default answering the question *posed* by the final `it.next(7)` call.

These questions and answers -- the two-way message passing with `yield` and `next(..)` -- are quite powerful, but it's not obvious at all how these mechanisms are connected to async flow control. We're getting there!

Multiple Iterators

It may appear from the syntactic usage that when you use an *iterator* to control a generator, you're controlling the declared generator function itself. But there's a subtlety that's easy to miss: each time you construct an *iterator*, you are implicitly constructing an instance of the generator which that *iterator* will control.

You can have multiple instances of the same generator running at the same time, and they can even interact:

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

var z = 1;

var it1 = foo();
var it2 = foo();

var val1 = it1.next().value;                    // 2 <-- yield 2
var val2 = it2.next().value;                    // 2 <-- yield 2

val1 = it1.next( val2 * 10 ).value;             // 40  <-- x:20,  z:2
val2 = it2.next( val1 * 5 ).value;              // 600 <-- x:200, z:3

it1.next( val2 / 2 );                                   // y:300

        // 20 300 3
it2.next( val1 / 4 );                                   // y:10

        // 200 10 3
```

Warning: The most common usage of multiple instances of the same generator running concurrently is not such interactions, but when the generator is producing its own values without input, perhaps from some independently connected resource. We'll talk more about value production in the next section.

Let's briefly walk through the processing:

1. Both instances of `*foo()` are started at the same time, and both `next()` calls reveal a `value` of `2` from the `yield 2` statements, respectively.
2. `val2 * 10` is `2 * 10`, which is sent into the first generator instance `it1`, so that `x` gets value `20`. `z` is incremented from `1` to `2`, and then `20 * 2` is `yield`ed out, setting `val1` to `40`.
3. `val1 * 5` is `40 * 5`, which is sent into the second generator instance `it2`, so that `x` gets value `200`. `z` is incremented again, from `2` to `3`, and then `200 * 3` is `yield`ed out, setting `val2` to `600`.
4. `val2 / 2` is `600 / 2`, which is sent into the first generator instance `it1`, so that `y` gets value `300`, then printing out `20 300 3` for its `x y z` values, respectively.

5. `val1 / 4` is `40 / 4`, which is sent into the second generator instance `it2`, so that `y` gets value `10`, then printing out `200 10 3` for its `x y z` values, respectively.

That's a "fun" example to run through in your mind. Did you keep it straight?

Interleaving

Recall this scenario from the "Run-to-completion" section of Chapter 1:

```
var a = 1;
var b = 2;

function foo() {
        a++;
        b = b * a;
        a = b + 3;
}

function bar() {
        b--;
        a = 8 + b;
        b = a * 2;
}
```

With normal JS functions, of course either `foo()` can run completely first, or `bar()` can run completely first, but `foo()` cannot interleave its individual statements with `bar()`. So, there are only two possible outcomes to the preceding program.
However, with generators, clearly interleaving (even in the middle of statements!) is possible:

```
var a = 1;
var b = 2;

function *foo() {
        a++;
        yield;
        b = b * a;
        a = (yield b) + 3;
}

function *bar() {
        b--;
        yield;
        a = (yield 8) + b;
        b = a * (yield 2);
}
```

Depending on what respective order the *iterators* controlling `*foo()` and `*bar()` are called, the preceding program could produce several different results. In other words, we can actually illustrate (in a sort of fake-ish way) the theoretical "threaded race conditions" circumstances discussed in Chapter 1, by interleaving the two generator iterations over the same shared variables.
First, let's make a helper called `step(..)` that controls an *iterator*:

```
function step(gen) {
```

```
        var it = gen();
        var last;

        return function() {
                // whatever is `yield`ed out, just
                // send it right back in the next time!
                last = it.next( last ).value;
        };
}
```

`step(..)` initializes a generator to create its `it` *iterator*, then returns a function which, when called, advances the *iterator* by one step. Additionally, the previously `yield`ded out value is sent right back in at the *next* step. So, `yield 8` will just become `8` and `yield b` will just be `b` (whatever it was at the time of `yield`).

Now, just for fun, let's experiment to see the effects of interleaving these different chunks of `*foo()` and `*bar()`. We'll start with the boring base case, making sure `*foo()` totally finishes before `*bar()` (just like we did in Chapter 1):

```
// make sure to reset `a` and `b`
a = 1;
b = 2;

var s1 = step( foo );
var s2 = step( bar );

// run `*foo()` completely first
s1();
s1();
s1();

// now run `*bar()`
s2();
s2();
s2();
s2();

console.log( a, b );     // 11 22
```

The end result is `11` and `22`, just as it was in the Chapter 1 version. Now let's mix up the interleaving ordering and see how it changes the final values of `a` and `b`:

```
// make sure to reset `a` and `b`
a = 1;
b = 2;

var s1 = step( foo );
var s2 = step( bar );

s2();           // b--;
s2();           // yield 8
s1();           // a++;
s2();           // a = 8 + b;
                        // yield 2
s1();           // b = b * a;
                        // yield b
s1();           // a = b + 3;
```

```
s2();            // b = a * 2;
```

Before I tell you the results, can you figure out what `a` and `b` are after the preceding program? No cheating!

```
console.log( a, b );     // 12 18
```

Note: As an exercise for the reader, try to see how many other combinations of results you can get back rearranging the order of the `s1()` and `s2()` calls. Don't forget you'll always need three `s1()` calls and four `s2()` calls. Recall the discussion earlier about matching `next()` with `yield` for the reasons why.

You almost certainly won't want to intentionally create *this* level of interleaving confusion, as it creates incredibly difficult to understand code. But the exercise is interesting and instructive to understand more about how multiple generators can run concurrently in the same shared scope, because there will be places where this capability is quite useful.

We'll discuss generator concurrency in more detail at the end of this chapter.

Generator'ing Values

In the previous section, we mentioned an interesting use for generators, as a way to produce values. This is **not** the main focus in this chapter, but we'd be remiss if we didn't cover the basics, especially because this use case is essentially the origin of the name: generators.

We're going to take a slight diversion into the topic of *iterators* for a bit, but we'll circle back to how they relate to generators and using a generator to *generate* values.

Producers and Iterators

Imagine you're producing a series of values where each value has a definable relationship to the previous value. To do this, you're going to need a stateful producer that remembers the last value it gave out.

You can implement something like that straightforwardly using a function closure (see the *Scope & Closures* title of this series):

```
var gimmeSomething = (function(){
        var nextVal;

        return function(){
                if (nextVal === undefined) {
                        nextVal = 1;
                }
                else {
                        nextVal = (3 * nextVal) + 6;
                }

                return nextVal;
        };
})();
```

```
gimmeSomething();               // 1
gimmeSomething();               // 9
gimmeSomething();               // 33
gimmeSomething();               // 105
```

Note: The `nextVal` computation logic here could have been simplified, but conceptually, we don't want to calculate the *next value* (aka `nextVal`) until the *next* `gimmeSomething()` call happens, because in general that could be a resource-leaky design for producers of more persistent or resource-limited values than simple `numbers`.
Generating an arbitrary number series isn't a terribly realistic example. But what if you were generating records from a data source? You could imagine much the same code.

In fact, this task is a very common design pattern, usually solved by iterators. An *iterator* is a well-defined interface for stepping through a series of values from a producer. The JS interface for iterators, as it is in most languages, is to call `next()` each time you want the next value from the producer.
We could implement the standard *iterator* interface for our number series producer:

```
var something = (function(){
        var nextVal;

        return {
                // needed for `for..of` loops
                [Symbol.iterator]: function(){ return this; },

                // standard iterator interface method
                next: function(){
                        if (nextVal === undefined) {
                                nextVal = 1;
                        }
                        else {
                                nextVal = (3 * nextVal) + 6;
                        }

                        return { done:false, value:nextVal };
                }
        };
})();

something.next().value;         // 1
something.next().value;         // 9
something.next().value;         // 33
something.next().value;         // 105
```

Note: We'll explain why we need the `[Symbol.iterator]: ..` part of this code snippet in the "Iterables" section. Syntactically though, two ES6 features are at play. First, the `[ .. ]` syntax is called a *computed property name* (see the *this & Object Prototypes* title of this series). It's a way in an object literal definition to specify an expression and use the result of that expression as the name for the property. Next, `Symbol.iterator` is one of ES6's predefined special `Symbol` values (see the *ES6 & Beyond* title of this book series).

The `next()` call returns an object with two properties: `done` is a `boolean` value signaling the *iterator's* complete status; `value` holds the iteration value.

ES6 also adds the `for..of` loop, which means that a standard *iterator* can automatically be consumed with native loop syntax:

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

	// don't let the loop run forever!
	if (v > 500) {
		break;
	}
}
// 1 9 33 105 321 969
```

Note: Because our `something` *iterator* always returns `done:false`, this `for..of` loop would run forever, which is why we put the `break` conditional in. It's totally OK for iterators to be never-ending, but there are also cases where the *iterator* will run over a finite set of values and eventually return a `done:true`.

The `for..of` loop automatically calls `next()` for each iteration -- it doesn't pass any values in to the `next()` -- and it will automatically terminate on receiving a `done:true`. It's quite handy for looping over a set of data.

Of course, you could manually loop over iterators, calling `next()` and checking for the `done:true` condition to know when to stop:

```
for (
	var ret;
	(ret = something.next()) && !ret.done;
) {
	console.log( ret.value );

	// don't let the loop run forever!
	if (ret.value > 500) {
		break;
	}
}
// 1 9 33 105 321 969
```

Note: This manual `for` approach is certainly uglier than the ES6 `for..of` loop syntax, but its advantage is that it affords you the opportunity to pass in values to the `next(..)` calls if necessary.

In addition to making your own *iterators*, many built-in data structures in JS (as of ES6), like `arrays`, also have default *iterators*:

```
var a = [1,3,5,7,9];

for (var v of a) {
	console.log( v );
}
// 1 3 5 7 9
```

The `for..of` loop asks `a` for its *iterator*, and automatically uses it to iterate over `a`'s values.

Note: It may seem a strange omission by ES6, but regular `objects` intentionally do not come with a default *iterator* the way `arrays` do. The reasons go deeper than we will cover here. If all you want is to iterate over the properties of an object (with no particular guarantee of ordering), `Object.keys(..)` returns an `array`, which can then be used like `for (var k of`

`Object.keys(obj)) { ...` Such a `for..of` loop over an object's keys would be similar to a `for..in` loop, except that `Object.keys(..)` does not include properties from the `[[Prototype]]` chain while `for..in` does (see the *this & Object Prototypes* title of this series).

Iterables

The `something` object in our running example is called an *iterator*, as it has the `next()` method on its interface. But a closely related term is *iterable*, which is an `object` that **contains** an *iterator* that can iterate over its values.

As of ES6, the way to retrieve an *iterator* from an *iterable* is that the *iterable* must have a function on it, with the name being the special ES6 symbol value `Symbol.iterator`. When this function is called, it returns an *iterator*. Though not required, generally each call should return a fresh new *iterator*.

`a` in the previous snippet is an *iterable*. The `for..of` loop automatically calls its `Symbol.iterator` function to construct an *iterator*. But we could of course call the function manually, and use the *iterator* it returns:

```
var a = [1,3,5,7,9];

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

it.next().value; // 1
it.next().value; // 3
it.next().value; // 5
..
```

In the previous code listing that defined `something`, you may have noticed this line:

```
[Symbol.iterator]: function(){ return this; }
```

That little bit of confusing code is making the `something` value -- the interface of the `something` *iterator* -- also an *iterable*; it's now both an *iterable* and an *iterator*. Then, we pass `something` to the `for..of` loop:

```
for (var v of something) {
        ..
}
```

The `for..of` loop expects `something` to be an *iterable*, so it looks for and calls its `Symbol.iterator` function. We defined that function to simply `return this`, so it just gives itself back, and the `for..of` loop is none the wiser.

Generator Iterator

Let's turn our attention back to generators, in the context of *iterators*. A generator can be treated as a producer of values that we extract one at a time through an *iterator* interface's `next()` calls.

So, a generator itself is not technically an *iterable*, though it's very similar -- when you execute the generator, you get an *iterator* back:

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

var it = foo();
```

We can implement the `something` infinite number series producer from earlier with a generator, like this:

```
function *something() {
	var nextVal;

	while (true) {
		if (nextVal === undefined) {
			nextVal = 1;
		}
		else {
			nextVal = (3 * nextVal) + 6;
		}

		yield nextVal;
	}
}
```

Note: A `while..true` loop would normally be a very bad thing to include in a real JS program, at least if it doesn't have a `break` or `return` in it, as it would likely run forever, synchronously, and block/lock-up the browser UI. However, in a generator, such a loop is generally totally OK if it has a `yield` in it, as the generator will pause at each iteration, `yield`ing back to the main program and/or to the event loop queue. To put it glibly, "generators put the `while..true` back in JS programming!"

That's a fair bit cleaner and simpler, right? Because the generator pauses at each `yield`, the state (scope) of the function `*something()` is kept around, meaning there's no need for the closure boilerplate to preserve variable state across calls.

Not only is it simpler code -- we don't have to make our own *iterator* interface -- it actually is more reason-able code, because it more clearly expresses the intent. For example, the `while..true` loop tells us the generator is intended to run forever -- to keep *generating* values as long as we keep asking for them.

And now we can use our shiny new `*something()` generator with a `for..of` loop, and you'll see it works basically identically:

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

	// don't let the loop run forever!
	if (v > 500) {
		break;
	}
}
// 1 9 33 105 321 969
```

But don't skip over `for (var v of something()) ..`! We didn't just reference `something` as a value like in earlier examples, but instead called the `*something()` generator to get its *iterator* for the `for..of` loop to use.

If you're paying close attention, two questions may arise from this interaction between the generator and the loop:

- Why couldn't we say `for (var v of something) ..`? Because `something` here is a generator, which is not an *iterable*. We have to call `something()` to construct a producer for the `for..of` loop to iterate over.
- The `something()` call produces an *iterator*, but the `for..of` loop wants an *iterable*, right? Yep. The generator's *iterator* also has a `Symbol.iterator` function on it, which basically does a `return this`, just like the `something` *iterable* we defined earlier. In other words, a generator's *iterator* is also an *iterable*!

Stopping the Generator

In the previous example, it would appear the *iterator* instance for the `*something()` generator was basically left in a suspended state forever after the `break` in the loop was called.
But there's a hidden behavior that takes care of that for you. "Abnormal completion" (i.e., "early termination") of the `for..of` loop -- generally caused by a `break`, `return`, or an uncaught exception -- sends a signal to the generator's *iterator* for it to terminate.
Note: Technically, the `for..of` loop also sends this signal to the *iterator* at the normal completion of the loop. For a generator, that's essentially a moot operation, as the generator's *iterator* had to complete first so the `for..of` loop completed. However, custom *iterators* might desire to receive this additional signal from `for..of` loop consumers.
While a `for..of` loop will automatically send this signal, you may wish to send the signal manually to an *iterator*; you do this by calling `return(..)`.
If you specify a `try..finally` clause inside the generator, it will always be run even when the generator is externally completed. This is useful if you need to clean up resources (database connections, etc.):

```
function *something() {
        try {
                var nextVal;

                while (true) {
                        if (nextVal === undefined) {
                                nextVal = 1;
                        }
                        else {
                                nextVal = (3 * nextVal) + 6;
                        }

                        yield nextVal;
                }
        }
        // cleanup clause
        finally {
                console.log( "cleaning up!" );
        }
}
```

The earlier example with `break` in the `for..of` loop will trigger the `finally` clause. But you could instead manually terminate the generator's *iterator* instance from the outside with `return(..)`:

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

```
        // don't let the loop run forever!
        if (v > 500) {
                console.log(
                        // complete the generator's iterator
                        it.return( "Hello World" ).value
                );
                // no `break` needed here
        }
}
// 1 9 33 105 321 969
// cleaning up!
// Hello World
```

When we call `it.return(..)`, it immediately terminates the generator, which of course runs the `finally` clause. Also, it sets the returned `value` to whatever you passed in to `return(..)`, which is how `"Hello World"` comes right back out. We also don't need to include a `break` now because the generator's *iterator* is set to `done:true`, so the `for..of` loop will terminate on its next iteration. Generators owe their namesake mostly to this *consuming produced values* use. But again, that's just one of the uses for generators, and frankly not even the main one we're concerned with in the context of this book.

But now that we more fully understand some of the mechanics of how they work, we can *next* turn our attention to how generators apply to async concurrency.

Iterating Generators Asynchronously

What do generators have to do with async coding patterns, fixing problems with callbacks, and the like? Let's get to answering that important question.

We should revisit one of our scenarios from Chapter 3. Let's recall the callback approach:

```
function foo(x,y,cb) {
        ajax(
                "http://some.url.1/?x=" + x + "&y=" + y,
                cb
        );
}

foo( 11, 31, function(err,text) {
        if (err) {
                console.error( err );
        }
        else {
                console.log( text );
        }
} );
```

If we wanted to express this same task flow control with a generator, we could do:

```
function foo(x,y) {
```

```
        ajax(
                "http://some.url.1/?x=" + x + "&y=" + y,
                function(err,data){
                        if (err) {
                                // throw an error into `*main()`
                                it.throw( err );
                        }
                        else {
                                // resume `*main()` with received `data`
                                it.next( data );
                        }
                }
        );
}

function *main() {
        try {
                var text = yield foo( 11, 31 );
                console.log( text );
        }
        catch (err) {
                console.error( err );
        }
}

var it = main();

// start it all up!
it.next();
```

At first glance, this snippet is longer, and perhaps a little more complex looking, than the callback snippet before it. But don't let that impression get you off track. The generator snippet is actually **much** better! But there's a lot going on for us to explain.

First, let's look at this part of the code, which is the most important:

```
var text = yield foo( 11, 31 );
console.log( text );
```

Think about how that code works for a moment. We're calling a normal function `foo(..)` and we're apparently able to get back the `text` from the Ajax call, even though it's asynchronous.
How is that possible? If you recall the beginning of Chapter 1, we had almost identical code:

```
var data = ajax( "..url 1.." );
console.log( data );
```

And that code didn't work! Can you spot the difference? It's the `yield` used in a generator.
That's the magic! That's what allows us to have what appears to be blocking, synchronous code, but it doesn't actually block the whole program; it only pauses/blocks the code in the generator itself.

In `yield foo(11,31)`, first the `foo(11,31)` call is made, which returns nothing (aka `undefined`), so we're making a call to request data, but we're actually then doing `yield undefined`. That's OK,

because the code is not currently relying on a `yield`ed value to do anything interesting. We'll revisit this point later in the chapter.
We're not using `yield` in a message passing sense here, only in a flow control sense to pause/block. Actually, it will have message passing, but only in one direction, after the generator is resumed.
So, the generator pauses at the `yield`, essentially asking the question, "what value should I return to assign to the variable `text`?" Who's going to answer that question?
Look at `foo(..)`. If the Ajax request is successful, we call:

```
it.next( data );
```

That's resuming the generator with the response data, which means that our paused `yield` expression receives that value directly, and then as it restarts the generator code, that value gets assigned to the local variable `text`.
Pretty cool, huh?

Take a step back and consider the implications. We have totally synchronous-looking code inside the generator (other than the `yield` keyword itself), but hidden behind the scenes, inside of `foo(..)`, the operations can complete asynchronously.
That's huge! That's a nearly perfect solution to our previously stated problem with callbacks not being able to express asynchrony in a sequential, synchronous fashion that our brains can relate to.

In essence, we are abstracting the asynchrony away as an implementation detail, so that we can reason synchronously/sequentially about our flow control: "Make an Ajax request, and when it finishes print out the response." And of course, we just expressed two steps in the flow control, but this same capability extends without bounds, to let us express however many steps we need to.

Tip: This is such an important realization, just go back and read the last three paragraphs again to let it sink in!

Synchronous Error Handling

But the preceding generator code has even more goodness to *yield* to us. Let's turn our attention to the `try..catch` inside the generator:

```
try {
        var text = yield foo( 11, 31 );
        console.log( text );
}
catch (err) {
        console.error( err );
}
```

How does this work? The `foo(..)` call is asynchronously completing, and doesn't `try..catch` fail to catch asynchronous errors, as we looked at in Chapter 3?
We already saw how the `yield` lets the assignment statement pause to wait for `foo(..)` to finish, so that the completed response can be assigned to `text`. The awesome part is that this `yield` pausing *also* allows the generator to `catch` an error. We throw that error into the generator with this part of the earlier code listing:

```
if (err) {
        // throw an error into `*main()`
        it.throw( err );
```

```
}
```

The `yield`-pause nature of generators means that not only do we get synchronous-looking `return` values from async function calls, but we can also synchronously `catch` errors from those async function calls!

So we've seen we can throw errors *into* a generator, but what about throwing errors *out of* a generator? Exactly as you'd expect:

```
function *main() {
        var x = yield "Hello World";

        yield x.toLowerCase();   // cause an exception!
}

var it = main();

it.next().value;                 // Hello World

try {
        it.next( 42 );
}
catch (err) {
        console.error( err );    // TypeError
}
```

Of course, we could have manually thrown an error with `throw ..` instead of causing an exception. We can even `catch` the same error that we `throw(..)` into the generator, essentially giving the generator a chance to handle it but if it doesn't, the *iterator* code must handle it:

```
function *main() {
        var x = yield "Hello World";

        // never gets here
        console.log( x );
}

var it = main();

it.next();

try {
        // will `*main()` handle this error? we'll see!
        it.throw( "Oops" );
}
catch (err) {
        // nope, didn't handle it!
        console.error( err );                    // Oops
}
```

Synchronous-looking error handling (via `try..catch`) with async code is a huge win for readability and reason-ability.

Generators + Promises

In our previous discussion, we showed how generators can be iterated asynchronously, which is a huge step forward in sequential reason-ability over the spaghetti mess of callbacks. But we lost something very important: the trustability and composability of Promises (see Chapter 3)!

Don't worry -- we can get that back. The best of all worlds in ES6 is to combine generators (synchronous-looking async code) with Promises (trustable and composable).

But how?

Recall from Chapter 3 the Promise-based approach to our running Ajax example:

```
function foo(x,y) {
	return request(
		"http://some.url.1/?x=" + x + "&y=" + y
	);
}

foo( 11, 31 )
.then(
	function(text){
		console.log( text );
	},
	function(err){
		console.error( err );
	}
);
```

In our earlier generator code for the running Ajax example, `foo(..)` returned nothing (`undefined`), and our *iterator* control code didn't care about that `yield`ed value.
But here the Promise-aware `foo(..)` returns a promise after making the Ajax call. That suggests that we could construct a promise with `foo(..)` and then `yield` it from the generator, and then the *iterator* control code would receive that promise.
But what should the *iterator* do with the promise?

It should listen for the promise to resolve (fulfillment or rejection), and then either resume the generator with the fulfillment message or throw an error into the generator with the rejection reason.

Let me repeat that, because it's so important. The natural way to get the most out of Promises and generators is **to `yield` a Promise**, and wire that Promise to control the generator's *iterator*.
Let's give it a try! First, we'll put the Promise-aware `foo(..)` together with the generator `*main()`:

```
function foo(x,y) {
	return request(
		"http://some.url.1/?x=" + x + "&y=" + y
	);
}

function *main() {
	try {
		var text = yield foo( 11, 31 );
		console.log( text );
	}
	catch (err) {
```

```
                console.error( err );
        }
}
```

The most powerful revelation in this refactor is that the code inside `*main()` **did not have to change at all!** Inside the generator, whatever values are `yield`ed out is just an opaque implementation detail, so we're not even aware it's happening, nor do we need to worry about it.
But how are we going to run `*main()` now? We still have some of the implementation plumbing work to do, to receive and wire up the `yield`ed promise so that it resumes the generator upon resolution. We'll start by trying that manually:

```
var it = main();

var p = it.next().value;

// wait for the `p` promise to resolve
p.then(
        function(text){
                it.next( text );
        },
        function(err){
                it.throw( err );
        }
);
```

Actually, that wasn't so painful at all, was it?

This snippet should look very similar to what we did earlier with the manually wired generator controlled by the error-first callback. Instead of an `if (err) { it.throw..`, the promise already splits fulfillment (success) and rejection (failure) for us, but otherwise the *iterator* control is identical. Now, we've glossed over some important details.

Most importantly, we took advantage of the fact that we knew that `*main()` only had one Promise-aware step in it. What if we wanted to be able to Promise-drive a generator no matter how many steps it has? We certainly don't want to manually write out the Promise chain differently for each generator! What would be much nicer is if there was a way to repeat (aka "loop" over) the iteration control, and each time a Promise comes out, wait on its resolution before continuing.
Also, what if the generator throws out an error (intentionally or accidentally) during the `it.next(..)` call? Should we quit, or should we `catch` it and send it right back in? Similarly, what if we `it.throw(..)` a Promise rejection into the generator, but it's not handled, and comes right back out?

Promise-Aware Generator Runner

The more you start to explore this path, the more you realize, "wow, it'd be great if there was just some utility to do it for me." And you're absolutely correct. This is such an important pattern, and you don't want to get it wrong (or exhaust yourself repeating it over and over), so your best bet is to use a utility that is specifically designed to *run* Promise-`yield`ing generators in the manner we've illustrated.

Several Promise abstraction libraries provide just such a utility, including my *asynquence* library and its `runner(..)`, which will be discussed in Appendix A of this book.

But for the sake of learning and illustration, let's just define our own standalone utility that we'll call `run(..)`:

```
// thanks to Benjamin Gruenbaum (@benjamingr on GitHub) for
// big improvements here!
function run(gen) {
	var args = [].slice.call( arguments, 1), it;

	// initialize the generator in the current context
	it = gen.apply( this, args );

	// return a promise for the generator completing
	return Promise.resolve()
		.then( function handleNext(value){
			// run to the next yielded value
			var next = it.next( value );

			return (function handleResult(next){
				// generator has completed running?
				if (next.done) {
					return next.value;
				}
				// otherwise keep going
				else {
					return Promise.resolve( next.value )
						.then(
							// resume the async loop
on
							// success, sending the
resolved
							// value back into the
generator
							handleNext,

							// if `value` is a
rejected
							// promise, propagate
error back
							// into the generator for
its own
							// error handling
							function handleErr(err) {
								return
Promise.resolve(
									it.throw(
err )
								)
								.then(
handleResult );
							}
						);
				}
			})(next);
```

```
                } );
}
```

As you can see, it's a quite a bit more complex than you'd probably want to author yourself, and you especially wouldn't want to repeat this code for each generator you use. So, a utility/library helper is definitely the way to go. Nevertheless, I encourage you to spend a few minutes studying that code listing to get a better sense of how to manage the generator+Promise negotiation.

How would you use `run(..)` with `*main()` in our *running* Ajax example?

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

run( main );
```

That's it! The way we wired `run(..)`, it will automatically advance the generator you pass to it, asynchronously until completion.

Note: The `run(..)` we defined returns a promise which is wired to resolve once the generator is complete, or receive an uncaught exception if the generator doesn't handle it. We don't show that capability here, but we'll come back to it later in the chapter.

ES7: `async` and `await`?

The preceding pattern -- generators yielding Promises that then control the generator's *iterator* to advance it to completion -- is such a powerful and useful approach, it would be nicer if we could do it without the clutter of the library utility helper (aka `run(..)`).

There's probably good news on that front. At the time of this writing, there's early but strong support for a proposal for more syntactic addition in this realm for the post-ES6, ES7-ish timeframe. Obviously, it's too early to guarantee the details, but there's a pretty decent chance it will shake out similar to the following:

```
function foo(x,y) {
        return request(
                "http://some.url.1/?x=" + x + "&y=" + y
        );
}

async function main() {
        try {
                var text = await foo( 11, 31 );
                console.log( text );
        }
        catch (err) {
                console.error( err );
        }
}

main();
```

As you can see, there's no `run(..)` call (meaning no need for a library utility!) to invoke and drive `main()` -- it's just called as a normal function. Also, `main()` isn't declared as a generator function anymore; it's a new kind of function: `async function`. And finally, instead of `yield`ing a Promise, we `await` for it to resolve.

The `async function` automatically knows what to do if you `await` a Promise -- it will pause the function (just like with generators) until the Promise resolves. We didn't illustrate it in this snippet, but calling an async function like `main()` automatically returns a promise that's resolved whenever the function finishes completely.

Tip: The `async` / `await` syntax should look very familiar to readers with experience in C#, because it's basically identical.

The proposal essentially codifies support for the pattern we've already derived, into a syntactic mechanism: combining Promises with sync-looking flow control code. That's the best of both worlds combined, to effectively address practically all of the major concerns we outlined with callbacks.

The mere fact that such a ES7-ish proposal already exists and has early support and enthusiasm is a major vote of confidence in the future importance of this async pattern.

Promise Concurrency in Generators

So far, all we've demonstrated is a single-step async flow with Promises+generators. But real-world code will often have many async steps.

If you're not careful, the sync-looking style of generators may lull you into complacency with how you structure your async concurrency, leading to suboptimal performance patterns. So we want to spend a little time exploring the options.

Imagine a scenario where you need to fetch data from two different sources, then combine those responses to make a third request, and finally print out the last response. We explored a similar scenario with Promises in Chapter 3, but let's reconsider it in the context of generators.

Your first instinct might be something like:

```
function *foo() {
        var r1 = yield request( "http://some.url.1" );
        var r2 = yield request( "http://some.url.2" );

        var r3 = yield request(
                "http://some.url.3/?v=" + r1 + "," + r2
        );

        console.log( r3 );
}

// use previously defined `run(..)` utility
run( foo );
```

This code will work, but in the specifics of our scenario, it's not optimal. Can you spot why?

Because the `r1` and `r2` requests can -- and for performance reasons, *should* -- run concurrently, but in this code they will run sequentially; the `"http://some.url.2"` URL isn't Ajax fetched until after the `"http://some.url.1"` request is finished. These two requests are independent, so the better performance approach would likely be to have them run at the same time.

But how exactly would you do that with a generator and `yield`? We know that `yield` is only a single pause point in the code, so you can't really do two pauses at the same time.
The most natural and effective answer is to base the async flow on Promises, specifically on their capability to manage state in a time-independent fashion (see "Future Value" in Chapter 3).

The simplest approach:

```
function *foo() {
	// make both requests "in parallel"
	var p1 = request( "http://some.url.1" );
	var p2 = request( "http://some.url.2" );

	// wait until both promises resolve
	var r1 = yield p1;
	var r2 = yield p2;

	var r3 = yield request(
		"http://some.url.3/?v=" + r1 + "," + r2
	);

	console.log( r3 );
}

// use previously defined `run(..)` utility
run( foo );
```

Why is this different from the previous snippet? Look at where the `yield` is and is not. `p1` and `p2` are promises for Ajax requests made concurrently (aka "in parallel"). It doesn't matter which one finishes first, because promises will hold onto their resolved state for as long as necessary.
Then we use two subsequent `yield` statements to wait for and retrieve the resolutions from the promises (into `r1` and `r2`, respectively). If `p1` resolves first, the `yield p1` resumes first then waits on the `yield p2` to resume. If `p2` resolves first, it will just patiently hold onto that resolution value until asked, but the `yield p1` will hold on first, until `p1` resolves.
Either way, both `p1` and `p2` will run concurrently, and both have to finish, in either order, before the `r3 = yield request..` Ajax request will be made.
If that flow control processing model sounds familiar, it's basically the same as what we identified in Chapter 3 as the "gate" pattern, enabled by the `Promise.all([ .. ])` utility. So, we could also express the flow control like this:

```
function *foo() {
	// make both requests "in parallel," and
	// wait until both promises resolve
	var results = yield Promise.all( [
		request( "http://some.url.1" ),
		request( "http://some.url.2" )
	] );

	var r1 = results[0];
	var r2 = results[1];

	var r3 = yield request(
		"http://some.url.3/?v=" + r1 + "," + r2
	);
```

```
        console.log( r3 );
}

// use previously defined `run(..)` utility
run( foo );
```

Note: As we discussed in Chapter 3, we can even use ES6 destructuring assignment to simplify the `var r1 = .. var r2 = ..` assignments, with `var [r1,r2] = results`.
In other words, all of the concurrency capabilities of Promises are available to us in the generator+Promise approach. So in any place where you need more than sequential this-then-that async flow control steps, Promises are likely your best bet.

Promises, Hidden

As a word of stylistic caution, be careful about how much Promise logic you include **inside your generators**. The whole point of using generators for asynchrony in the way we've described is to create simple, sequential, sync-looking code, and to hide as much of the details of asynchrony away from that code as possible.

For example, this might be a cleaner approach:

```
// note: normal function, not generator
function bar(url1,url2) {
        return Promise.all( [
                request( url1 ),
                request( url2 )
        ] );
}

function *foo() {
        // hide the Promise-based concurrency details
        // inside `bar(..)`
        var results = yield bar(
                "http://some.url.1",
                "http://some.url.2"
        );

        var r1 = results[0];
        var r2 = results[1];

        var r3 = yield request(
                "http://some.url.3/?v=" + r1 + "," + r2
        );

        console.log( r3 );
}

// use previously defined `run(..)` utility
run( foo );
```

Inside `*foo()`, it's cleaner and clearer that all we're doing is just asking `bar(..)` to get us some `results`, and we'll `yield`-wait on that to happen. We don't have to care that under the covers a `Promise.all([ .. ])` Promise composition will be used to make that happen.
We treat asynchrony, and indeed Promises, as an implementation detail.

Hiding your Promise logic inside a function that you merely call from your generator is especially useful if you're going to do a sophisticated series flow-control. For example:

```
function bar() {
        return  Promise.all( [
                  baz( .. )
                  .then( .. ),
                  Promise.race( [ .. ] )
                ] )
                .then( .. )
}
```

That kind of logic is sometimes required, and if you dump it directly inside your generator(s), you've defeated most of the reason why you would want to use generators in the first place.
We *should* intentionally abstract such details away from our generator code so that they don't clutter up the higher level task expression.

Beyond creating code that is both functional and performant, you should also strive to make code that is as reason-able and maintainable as possible.

Note: Abstraction is not *always* a healthy thing for programming -- many times it can increase complexity in exchange for terseness. But in this case, I believe it's much healthier for your generator+Promise async code than the alternatives. As with all such advice, though, pay attention to your specific situations and make proper decisions for you and your team.

Generator Delegation

In the previous section, we showed calling regular functions from inside a generator, and how that remains a useful technique for abstracting away implementation details (like async Promise flow). But the main drawback of using a normal function for this task is that it has to behave by the normal function rules, which means it cannot pause itself with `yield` like a generator can.
It may then occur to you that you might try to call one generator from another generator, using our `run(..)` helper, such as:

```
function *foo() {
        var r2 = yield request( "http://some.url.2" );
        var r3 = yield request( "http://some.url.3/?v=" + r2 );

        return r3;
}

function *bar() {
        var r1 = yield request( "http://some.url.1" );

        // "delegating" to `*foo()` via `run(..)`
```

```
        var r3 = yield run( foo );

        console.log( r3 );
}

run( bar );
```

We run `*foo()` inside of `*bar()` by using our `run(..)` utility again. We take advantage here of the fact that the `run(..)` we defined earlier returns a promise which is resolved when its generator is run to completion (or errors out), so if we `yield` out to a `run(..)` instance the promise from another `run(..)` call, it automatically pauses `*bar()` until `*foo()` finishes.

But there's an even better way to integrate calling `*foo()` into `*bar()`, and it's called `yield`-delegation. The special syntax for `yield`-delegation is: `yield * __` (notice the extra `*`). Before we see it work in our previous example, let's look at a simpler scenario:

```
function *foo() {
        console.log( "`*foo()` starting" );
        yield 3;
        yield 4;
        console.log( "`*foo()` finished" );
}

function *bar() {
        yield 1;
        yield 2;
        yield *foo();    // `yield`-delegation!
        yield 5;
}

var it = bar();

it.next().value; // 1
it.next().value; // 2
it.next().value; // `*foo()` starting
                                        // 3
it.next().value; // 4
it.next().value; // `*foo()` finished
                                        // 5
```

Note: Similar to a note earlier in the chapter where I explained why I prefer `function *foo() ..` instead of `function* foo() ..`, I also prefer -- differing from most other documentation on the topic -- to say `yield *foo()` instead of `yield* foo()`. The placement of the `*` is purely stylistic and up to your best judgment. But I find the consistency of styling attractive.

How does the `yield *foo()` delegation work?

First, calling `foo()` creates an *iterator* exactly as we've already seen. Then, `yield *` delegates/transfers the *iterator* instance control (of the present `*bar()` generator) over to this other `*foo()` *iterator*.

So, the first two `it.next()` calls are controlling `*bar()`, but when we make the third `it.next()` call, now `*foo()` starts up, and now we're controlling `*foo()` instead of `*bar()`. That's why it's called delegation -- `*bar()` delegated its iteration control to `*foo()`.

As soon as the `it` *iterator* control exhausts the entire `*foo()` *iterator*, it automatically returns to controlling `*bar()`.

So now back to the previous example with the three sequential Ajax requests:

```
function *foo() {
        var r2 = yield request( "http://some.url.2" );
        var r3 = yield request( "http://some.url.3/?v=" + r2 );

        return r3;
}

function *bar() {
        var r1 = yield request( "http://some.url.1" );

        // "delegating" to `*foo()` via `yield*`
        var r3 = yield *foo();

        console.log( r3 );
}

run( bar );
```

The only difference between this snippet and the version used earlier is the use of `yield *foo()` instead of the previous `yield run(foo)`.
Note: `yield *` yields iteration control, not generator control; when you invoke the `*foo()` generator, you're now `yield`-delegating to its *iterator*. But you can actually `yield`-delegate to any *iterable*; `yield *[1,2,3]` would consume the default *iterator* for the `[1,2,3]` array value.

Why Delegation?

The purpose of `yield`-delegation is mostly code organization, and in that way is symmetrical with normal function calling.
Imagine two modules that respectively provide methods `foo()` and `bar()`, where `bar()` calls `foo()`. The reason the two are separate is generally because the proper organization of code for the program calls for them to be in separate functions. For example, there may be cases where `foo()` is called standalone, and other places where `bar()` calls `foo()`.
For all these exact same reasons, keeping generators separate aids in program readability, maintenance, and debuggability. In that respect, `yield *` is a syntactic shortcut for manually iterating over the steps of `*foo()` while inside of `*bar()`.
Such manual approach would be especially complex if the steps in `*foo()` were asynchronous, which is why you'd probably need to use that `run(..)` utility to do it. And as we've shown, `yield *foo()` eliminates the need for a sub-instance of the `run(..)` utility (like `run(foo)`).

Delegating Messages

You may wonder how this `yield`-delegation works not just with *iterator* control but with the two-way message passing. Carefully follow the flow of messages in and out, through the `yield`-delegation:

```
function *foo() {
        console.log( "inside `*foo()`:", yield "B" );

        console.log( "inside `*foo()`:", yield "C" );
```

```
        return "D";
}

function *bar() {
        console.log( "inside `*bar()`:", yield "A" );

        // `yield`-delegation!
        console.log( "inside `*bar()`:", yield *foo() );

        console.log( "inside `*bar()`:", yield "E" );

        return "F";
}

var it = bar();

console.log( "outside:", it.next().value );
// outside: A

console.log( "outside:", it.next( 1 ).value );
// inside `*bar()`: 1
// outside: B

console.log( "outside:", it.next( 2 ).value );
// inside `*foo()`: 2
// outside: C

console.log( "outside:", it.next( 3 ).value );
// inside `*foo()`: 3
// inside `*bar()`: D
// outside: E

console.log( "outside:", it.next( 4 ).value );
// inside `*bar()`: 4
// outside: F
```

Pay particular attention to the processing steps after the `it.next(3)` call:

1. The `3` value is passed (through the `yield`-delegation in `*bar()`) into the waiting `yield "C"` expression inside of `*foo()`.
2. `*foo()` then calls `return "D"`, but this value doesn't get returned all the way back to the outside `it.next(3)` call.
3. Instead, the `"D"` value is sent as the result of the waiting `yield *foo()` expression inside of `*bar()` -- this `yield`-delegation expression has essentially been paused while all of `*foo()` was exhausted. So `"D"` ends up inside of `*bar()` for it to print out.
4. `yield "E"` is called inside of `*bar()`, and the `"E"` value is yielded to the outside as the result of the `it.next(3)` call.

From the perspective of the external *iterator* (`it`), it doesn't appear any differently between controlling the initial generator or a delegated one.

In fact, `yield`-delegation doesn't even have to be directed to another generator; it can just be directed to a non-generator, general *iterable*. For example:

```
function *bar() {
	console.log( "inside `*bar()`:", yield "A" );

	// `yield`-delegation to a non-generator!
	console.log( "inside `*bar()`:", yield *[ "B", "C", "D" ] );

	console.log( "inside `*bar()`:", yield "E" );

	return "F";
}

var it = bar();

console.log( "outside:", it.next().value );
// outside: A

console.log( "outside:", it.next( 1 ).value );
// inside `*bar()`: 1
// outside: B

console.log( "outside:", it.next( 2 ).value );
// outside: C

console.log( "outside:", it.next( 3 ).value );
// outside: D

console.log( "outside:", it.next( 4 ).value );
// inside `*bar()`: undefined
// outside: E

console.log( "outside:", it.next( 5 ).value );
// inside `*bar()`: 5
// outside: F
```

Notice the differences in where the messages were received/reported between this example and the one previous.

Most strikingly, the default `array` *iterator* doesn't care about any messages sent in via `next(..)` calls, so the values `2`, `3`, and `4` are essentially ignored. Also, because that *iterator* has no explicit `return` value (unlike the previously used `*foo()`), the `yield *` expression gets an `undefined` when it finishes.

Exceptions Delegated, Too!

In the same way that `yield`-delegation transparently passes messages through in both directions, errors/exceptions also pass in both directions:

```
function *foo() {
	try {
		yield "B";
	}
```

```
        catch (err) {
                console.log( "error caught inside `*foo()`:", err );
        }

        yield "C";

        throw "D";
}

function *bar() {
        yield "A";

        try {
                yield *foo();
        }
        catch (err) {
                console.log( "error caught inside `*bar()`:", err );
        }

        yield "E";

        yield *baz();

        // note: can't get here!
        yield "G";
}

function *baz() {
        throw "F";
}

var it = bar();

console.log( "outside:", it.next().value );
// outside: A

console.log( "outside:", it.next( 1 ).value );
// outside: B

console.log( "outside:", it.throw( 2 ).value );
// error caught inside `*foo()`: 2
// outside: C

console.log( "outside:", it.next( 3 ).value );
// error caught inside `*bar()`: D
// outside: E

try {
        console.log( "outside:", it.next( 4 ).value );
}
catch (err) {
        console.log( "error caught outside:", err );
}
// error caught outside: F
```

Some things to note from this snippet:

1. When we call `it.throw(2)`, it sends the error message `2` into `*bar()`, which delegates that to `*foo()`, which then `catch`es it and handles it gracefully. Then, the `yield "C"` sends `"C"` back out as the return `value` from the `it.throw(2)` call.
2. The `"D"` value that's next `thrown` from inside `*foo()` propagates out to `*bar()`, which `catch`es it and handles it gracefully. Then the `yield "E"` sends `"E"` back out as the return `value` from the `it.next(3)` call.
3. Next, the exception `thrown` from `*baz()` isn't caught in `*bar()` -- though we did `catch` it outside -- so both `*baz()` and `*bar()` are set to a completed state. After this snippet, you would not be able to get the `"G"` value out with any subsequent `next(..)` call(s) -- they will just return `undefined` for `value`.

Delegating Asynchrony

Let's finally get back to our earlier `yield`-delegation example with the multiple sequential Ajax requests:

```
function *foo() {
	var r2 = yield request( "http://some.url.2" );
	var r3 = yield request( "http://some.url.3/?v=" + r2 );

	return r3;
}

function *bar() {
	var r1 = yield request( "http://some.url.1" );

	var r3 = yield *foo();

	console.log( r3 );
}

run( bar );
```

Instead of calling `yield run(foo)` inside of `*bar()`, we just call `yield *foo()`.
In the previous version of this example, the Promise mechanism (controlled by `run(..)`) was used to transport the value from `return r3` in `*foo()` to the local variable `r3` inside `*bar()`. Now, that value is just returned back directly via the `yield *` mechanics.
Otherwise, the behavior is pretty much identical.

Delegating "Recursion"

Of course, `yield`-delegation can keep following as many delegation steps as you wire up. You could even use `yield`-delegation for async-capable generator "recursion" -- a generator `yield`-delegating to itself:

```
function *foo(val) {
	if (val > 1) {
		// generator recursion
		val = yield *foo( val - 1 );
```

```
        }

        return yield request( "http://some.url/?v=" + val );
}

function *bar() {
        var r1 = yield *foo( 3 );
        console.log( r1 );
}

run( bar );
```

Note: Our `run(..)` utility could have been called with `run( foo, 3 )`, because it supports additional parameters being passed along to the initialization of the generator. However, we used a parameter-free `*bar()` here to highlight the flexibility of `yield *`.
What processing steps follow from that code? Hang on, this is going to be quite intricate to describe in detail:

1. `run(bar)` starts up the `*bar()` generator.
2. `foo(3)` creates an *iterator* for `*foo(..)` and passes `3` as its `val` parameter.
3. Because `3 > 1`, `foo(2)` creates another *iterator* and passes in `2` as its `val` parameter.
4. Because `2 > 1`, `foo(1)` creates yet another *iterator* and passes in `1` as its `val` parameter.
5. `1 > 1` is `false`, so we next call `request(..)` with the `1` value, and get a promise back for that first Ajax call.
6. That promise is `yield`ed out, which comes back to the `*foo(2)` generator instance.
7. The `yield *` passes that promise back out to the `*foo(3)` generator instance. Another `yield *` passes the promise out to the `*bar()` generator instance. And yet again another `yield *` passes the promise out to the `run(..)` utility, which will wait on that promise (for the first Ajax request) to proceed.
8. When the promise resolves, its fulfillment message is sent to resume `*bar()`, which passes through the `yield *` into the `*foo(3)` instance, which then passes through the `yield *` to the `*foo(2)` generator instance, which then passes through the `yield *` to the normal `yield` that's waiting in the `*foo(3)` generator instance.
9. That first call's Ajax response is now immediately `return`ed from the `*foo(3)` generator instance, which sends that value back as the result of the `yield *` expression in the `*foo(2)` instance, and assigned to its local `val` variable.
10. Inside `*foo(2)`, a second Ajax request is made with `request(..)`, whose promise is `yield`ed back to the `*foo(1)` instance, and then `yield *` propagates all the way out to `run(..)` (step 7 again). When the promise resolves, the second Ajax response propagates all the way back into the `*foo(2)` generator instance, and is assigned to its local `val` variable.
11. Finally, the third Ajax request is made with `request(..)`, its promise goes out to `run(..)`, and then its resolution value comes all the way back, which is then `return`ed so that it comes back to the waiting `yield *` expression in `*bar()`.

Phew! A lot of crazy mental juggling, huh? You might want to read through that a few more times, and then go grab a snack to clear your head!

Generator Concurrency

As we discussed in both Chapter 1 and earlier in this chapter, two simultaneously running "processes" can cooperatively interleave their operations, and many times this can *yield* (pun intended) very powerful asynchrony expressions.

Frankly, our earlier examples of concurrency interleaving of multiple generators showed how to make it really confusing. But we hinted that there's places where this capability is quite useful.

Recall a scenario we looked at in Chapter 1, where two different simultaneous Ajax response handlers needed to coordinate with each other to make sure that the data communication was not a race condition. We slotted the responses into the `res` array like this:

```
function response(data) {
        if (data.url == "http://some.url.1") {
                res[0] = data;
        }
        else if (data.url == "http://some.url.2") {
                res[1] = data;
        }
}
```

But how can we use multiple generators concurrently for this scenario?

```
// `request(..)` is a Promise-aware Ajax utility

var res = [];

function *reqData(url) {
        res.push(
                yield request( url )
        );
}
```

Note: We're going to use two instances of the `*reqData(..)` generator here, but there's no difference to running a single instance of two different generators; both approaches are reasoned about identically. We'll see two different generators coordinating in just a bit.

Instead of having to manually sort out `res[0]` and `res[1]` assignments, we'll use coordinated ordering so that `res.push(..)` properly slots the values in the expected and predictable order. The expressed logic thus should feel a bit cleaner.

But how will we actually orchestrate this interaction? First, let's just do it manually, with Promises:

```
var it1 = reqData( "http://some.url.1" );
var it2 = reqData( "http://some.url.2" );

var p1 = it1.next().value;
var p2 = it2.next().value;

p1
.then( function(data){
        it1.next( data );
        return p2;
```

```
} )
.then( function(data){
        it2.next( data );
} );
```

`*reqData(..)`'s two instances are both started to make their Ajax requests, then paused with `yield`. Then we choose to resume the first instance when `p1` resolves, and then `p2`'s resolution will restart the second instance. In this way, we use Promise orchestration to ensure that `res[0]` will have the first response and `res[1]` will have the second response.
But frankly, this is awfully manual, and it doesn't really let the generators orchestrate themselves, which is where the true power can lie. Let's try it a different way:

```
// `request(..)` is a Promise-aware Ajax utility

var res = [];

function *reqData(url) {
        var data = yield request( url );

        // transfer control
        yield;

        res.push( data );
}

var it1 = reqData( "http://some.url.1" );
var it2 = reqData( "http://some.url.2" );

var p1 = it1.next().value;
var p2 = it2.next().value;

p1.then( function(data){
        it1.next( data );
} );

p2.then( function(data){
        it2.next( data );
} );

Promise.all( [p1,p2] )
.then( function(){
        it1.next();
        it2.next();
} );
```

OK, this is a bit better (though still manual!), because now the two instances of `*reqData(..)` run truly concurrently, and (at least for the first part) independently.
In the previous snippet, the second instance was not given its data until after the first instance was totally finished. But here, both instances receive their data as soon as their respective responses come back, and then each instance does another `yield` for control transfer purposes. We then choose what order to resume them in the `Promise.all([ .. ])` handler.
What may not be as obvious is that this approach hints at an easier form for a reusable utility, because of the symmetry. We can do even better. Let's imagine using a utility called `runAll(..)`:

```
// `request(..)` is a Promise-aware Ajax utility

var res = [];

runAll(
        function*(){
                var p1 = request( "http://some.url.1" );

                // transfer control
                yield;

                res.push( yield p1 );
        },
        function*(){
                var p2 = request( "http://some.url.2" );

                // transfer control
                yield;

                res.push( yield p2 );
        }
);
```

Note: We're not including a code listing for `runAll(..)` as it is not only long enough to bog down the text, but is an extension of the logic we've already implemented in `run(..)` earlier. So, as a good supplementary exercise for the reader, try your hand at evolving the code from `run(..)` to work like the imagined `runAll(..)`. Also, my *asynquence* library provides a previously mentioned `runner(..)` utility with this kind of capability already built in, and will be discussed in Appendix A of this book.

Here's how the processing inside `runAll(..)` would operate:

1. The first generator gets a promise for the first Ajax response from `"http://some.url.1"`, then `yield`s control back to the `runAll(..)` utility.
2. The second generator runs and does the same for `"http://some.url.2"`, `yield`ing control back to the `runAll(..)` utility.
3. The first generator resumes, and then `yield`s out its promise `p1`. The `runAll(..)` utility does the same in this case as our previous `run(..)`, in that it waits on that promise to resolve, then resumes the same generator (no control transfer!).
   When `p1` resolves, `runAll(..)` resumes the first generator again with that resolution value, and then `res[0]` is given its value. When the first generator then finishes, that's an implicit transfer of control.
4. The second generator resumes, `yield`s out its promise `p2`, and waits for it to resolve. Once it does, `runAll(..)` resumes the second generator with that value, and `res[1]` is set.

In this running example, we use an outer variable called `res` to store the results of the two different Ajax responses -- that's our concurrency coordination making that possible.

But it might be quite helpful to further extend `runAll(..)` to provide an inner variable space for the multiple generator instances to *share*, such as an empty object we'll call `data` below. Also, it could take non-Promise values that are `yield`ed and hand them off to the next generator.

Consider:

```
// `request(..)` is a Promise-aware Ajax utility

runAll(
	function*(data){
		data.res = [];

		// transfer control (and message pass)
		var url1 = yield "http://some.url.2";

		var p1 = request( url1 ); // "http://some.url.1"

		// transfer control
		yield;

		data.res.push( yield p1 );
	},
	function*(data){
		// transfer control (and message pass)
		var url2 = yield "http://some.url.1";

		var p2 = request( url2 ); // "http://some.url.2"

		// transfer control
		yield;

		data.res.push( yield p2 );
	}
);
```

In this formulation, the two generators are not just coordinating control transfer, but actually communicating with each other, both through `data.res` and the `yield`ed messages that trade `url1` and `url2` values. That's incredibly powerful!
Such realization also serves as a conceptual base for a more sophisticated asynchrony technique called CSP (Communicating Sequential Processes), which we will cover in Appendix B of this book.

Thunks

So far, we've made the assumption that `yield`ing a Promise from a generator -- and having that Promise resume the generator via a helper utility like `run(..)` -- was the best possible way to manage asynchrony with generators. To be clear, it is.
But we skipped over another pattern that has some mildly widespread adoption, so in the interest of completeness we'll take a brief look at it.

In general computer science, there's an old pre-JS concept called a "thunk." Without getting bogged down in the historical nature, a narrow expression of a thunk in JS is a function that -- without any parameters -- is wired to call another function.

In other words, you wrap a function definition around function call -- with any parameters it needs -- to *defer* the execution of that call, and that wrapping function is a thunk. When you later execute the thunk, you end up calling the original function.

For example:

```
function foo(x,y) {
	return x + y;
}

function fooThunk() {
	return foo( 3, 4 );
}

// later

console.log( fooThunk() );	// 7
```

So, a synchronous thunk is pretty straightforward. But what about an async thunk? We can essentially extend the narrow thunk definition to include it receiving a callback.

Consider:

```
function foo(x,y,cb) {
	setTimeout( function(){
		cb( x + y );
	}, 1000 );
}

function fooThunk(cb) {
	foo( 3, 4, cb );
}

// later

fooThunk( function(sum){
	console.log( sum );		// 7
} );
```

As you can see, `fooThunk(..)` only expects a `cb(..)` parameter, as it already has values `3` and `4` (for `x` and `y`, respectively) pre-specified and ready to pass to `foo(..)`. A thunk is just waiting around patiently for the last piece it needs to do its job: the callback.
You don't want to make thunks manually, though. So, let's invent a utility that does this wrapping for us.

Consider:

```
function thunkify(fn) {
	var args = [].slice.call( arguments, 1 );
	return function(cb) {
		args.push( cb );
		return fn.apply( null, args );
	};
}

var fooThunk = thunkify( foo, 3, 4 );
```

```
// later

fooThunk( function(sum) {
        console.log( sum );             // 7
} );
```

Tip: Here we assume that the original (`foo(..)`) function signature expects its callback in the last position, with any other parameters coming before it. This is a pretty ubiquitous "standard" for async JS function standards. You might call it "callback-last style." If for some reason you had a need to handle "callback-first style" signatures, you would just make a utility that used `args.unshift(..)` instead of `args.push(..)`.

The preceding formulation of `thunkify(..)` takes both the `foo(..)` function reference, and any parameters it needs, and returns back the thunk itself (`fooThunk(..)`). However, that's not the typical approach you'll find to thunks in JS.

Instead of `thunkify(..)` making the thunk itself, typically -- if not perplexingly -- the `thunkify(..)` utility would produce a function that produces thunks.

Uhhhh... yeah.

Consider:

```
function thunkify(fn) {
        return function() {
                var args = [].slice.call( arguments );
                return function(cb) {
                        args.push( cb );
                        return fn.apply( null, args );
                };
        };
}
```

The main difference here is the extra `return function() { .. }` layer. Here's how its usage differs:

```
var whatIsThis = thunkify( foo );

var fooThunk = whatIsThis( 3, 4 );

// later

fooThunk( function(sum) {
        console.log( sum );             // 7
} );
```

Obviously, the big question this snippet implies is what is `whatIsThis` properly called? It's not the thunk, it's the thing that will produce thunks from `foo(..)` calls. It's kind of like a "factory" for "thunks." There doesn't seem to be any kind of standard agreement for naming such a thing.

So, my proposal is "thunkory" ("thunk" + "factory"). So, `thunkify(..)` produces a thunkory, and a thunkory produces thunks. That reasoning is symmetric to my proposal for "promisory" in Chapter 3:

```
var fooThunkory = thunkify( foo );

var fooThunk1 = fooThunkory( 3, 4 );
var fooThunk2 = fooThunkory( 5, 6 );

// later
```

```
fooThunk1( function(sum) {
        console.log( sum );             // 7
} );

fooThunk2( function(sum) {
        console.log( sum );             // 11
} );
```

Note: The running `foo(..)` example expects a style of callback that's not "error-first style." Of course, "error-first style" is much more common. If `foo(..)` had some sort of legitimate error-producing expectation, we could change it to expect and use an error-first callback. None of the subsequent `thunkify(..)` machinery cares what style of callback is assumed. The only difference in usage would be `fooThunk1(function(err,sum){...`.
Exposing the thunkory method -- instead of how the earlier `thunkify(..)` hides this intermediary step -- may seem like unnecessary complication. But in general, it's quite useful to make thunkories at the beginning of your program to wrap existing API methods, and then be able to pass around and call those thunkories when you need thunks. The two distinct steps preserve a cleaner separation of capability.
To illustrate:

```
// cleaner:
var fooThunkory = thunkify( foo );

var fooThunk1 = fooThunkory( 3, 4 );
var fooThunk2 = fooThunkory( 5, 6 );

// instead of:
var fooThunk1 = thunkify( foo, 3, 4 );
var fooThunk2 = thunkify( foo, 5, 6 );
```

Regardless of whether you like to deal with the thunkories explicitly or not, the usage of thunks `fooThunk1(..)` and `fooThunk2(..)` remains the same.

s/promise/thunk/

So what's all this thunk stuff have to do with generators?

Comparing thunks to promises generally: they're not directly interchangable as they're not equivalent in behavior. Promises are vastly more capable and trustable than bare thunks.

But in another sense, they both can be seen as a request for a value, which may be async in its answering.

Recall from Chapter 3 we defined a utility for promisifying a function, which we called `Promise.wrap(..)` -- we could have called it `promisify(..)`, too! This Promise-wrapping utility doesn't produce Promises; it produces promisories that in turn produce Promises. This is completely symmetric to the thunkories and thunks presently being discussed.
To illustrate the symmetry, let's first alter the running `foo(..)` example from earlier to assume an "error-first style" callback:

```
function foo(x,y,cb) {
	setTimeout( function(){
		// assume `cb(..)` as "error-first style"
		cb( null, x + y );
	}, 1000 );
}
```

Now, we'll compare using `thunkify(..)` and `promisify(..)` (aka `Promise.wrap(..)` from Chapter 3):

```
// symmetrical: constructing the question asker
var fooThunkory = thunkify( foo );
var fooPromisory = promisify( foo );

// symmetrical: asking the question
var fooThunk = fooThunkory( 3, 4 );
var fooPromise = fooPromisory( 3, 4 );

// get the thunk answer
fooThunk( function(err,sum){
	if (err) {
		console.error( err );
	}
	else {
		console.log( sum );		// 7
	}
} );

// get the promise answer
fooPromise
.then(
	function(sum){
		console.log( sum );		// 7
	},
	function(err){
		console.error( err );
	}
);
```

Both the thunkory and the promisory are essentially asking a question (for a value), and respectively the thunk `fooThunk` and promise `fooPromise` represent the future answers to that question. Presented in that light, the symmetry is clear.

With that perspective in mind, we can see that generators which `yield` Promises for asynchrony could instead `yield` thunks for asynchrony. All we'd need is a smarter `run(..)` utility (like from before) that can not only look for and wire up to a `yield`ed Promise but also to provide a callback to a `yield`ed thunk.

Consider:

```
function *foo() {
	var val = yield request( "http://some.url.1" );
	console.log( val );
}

run( foo );
```

In this example, `request(..)` could either be a promisory that returns a promise, or a thunkory that returns a thunk. From the perspective of what's going on inside the generator code logic, we don't care about that implementation detail, which is quite powerful!
So, `request(..)` could be either:

```
// promisory `request(..)` (see Chapter 3)
var request = Promise.wrap( ajax );

// vs.

// thunkory `request(..)`
var request = thunkify( ajax );
```

Finally, as a thunk-aware patch to our earlier `run(..)` utility, we would need logic like this:

```
// ..
// did we receive a thunk back?
else if (typeof next.value == "function") {
        return new Promise( function(resolve,reject){
                // call the thunk with an error-first callback
                next.value( function(err,msg) {
                        if (err) {
                                reject( err );
                        }
                        else {
                                resolve( msg );
                        }
                } );
        } )
        .then(
                handleNext,
                function handleErr(err) {
                        return Promise.resolve(
                                it.throw( err )
                        )
                        .then( handleResult );
                }
        );
}
```

Now, our generators can either call promisories to `yield` Promises, or call thunkories to `yield` thunks, and in either case, `run(..)` would handle that value and use it to wait for the completion to resume the generator.
Symmetry wise, these two approaches look identical. However, we should point out that's true only from the perspective of Promises or thunks representing the future value continuation of a generator.

From the larger perspective, thunks do not in and of themselves have hardly any of the trustability or composability guarantees that Promises are designed with. Using a thunk as a stand-in for a Promise in this particular generator asynchrony pattern is workable but should be seen as less than ideal when compared to all the benefits that Promises offer (see Chapter 3).

If you have the option, prefer `yield pr` rather than `yield th`. But there's nothing wrong with having a `run(..)` utility which can handle both value types.

Note: The `runner(..)` utility in my *asynquence* library, which will be discussed in Appendix A, handles `yield`s of Promises, thunks and *asynquence* sequences.

Pre-ES6 Generators

You're hopefully convinced now that generators are a very important addition to the async programming toolbox. But it's a new syntax in ES6, which means you can't just polyfill generators like you can Promises (which are just a new API). So what can we do to bring generators to our browser JS if we don't have the luxury of ignoring pre-ES6 browsers?

For all new syntax extensions in ES6, there are tools -- the most common term for them is transpilers, for trans-compilers -- which can take your ES6 syntax and transform it into equivalent (but obviously uglier!) pre-ES6 code. So, generators can be transpiled into code that will have the same behavior but work in ES5 and below.

But how? The "magic" of `yield` doesn't obviously sound like code that's easy to transpile. We actually hinted at a solution in our earlier discussion of closure-based *iterators*.

Manual Transformation

Before we discuss the transpilers, let's derive how manual transpilation would work in the case of generators. This isn't just an academic exercise, because doing so will actually help further reinforce how they work.

Consider:

```
// `request(..)` is a Promise-aware Ajax utility

function *foo(url) {
	try {
		console.log( "requesting:", url );
		var val = yield request( url );
		console.log( val );
	}
	catch (err) {
		console.log( "Oops:", err );
		return false;
	}
}

var it = foo( "http://some.url.1" );
```

The first thing to observe is that we'll still need a normal `foo()` function that can be called, and it will still need to return an *iterator*. So, let's sketch out the non-generator transformation:

```
function foo(url) {

	// ..

	// make and return an iterator
```

```
        return {
                next: function(v) {
                        // ..
                },
                throw: function(e) {
                        // ..
                }
        };
}

var it = foo( "http://some.url.1" );
```

The next thing to observe is that a generator does its "magic" by suspending its scope/state, but we can emulate that with function closure (see the *Scope & Closures* title of this series). To understand how to write such code, we'll first annotate different parts of our generator with state values:

```
// `request(..)` is a Promise-aware Ajax utility

function *foo(url) {
        // STATE *1*

        try {
                console.log( "requesting:", url );
                var TMP1 = request( url );

                // STATE *2*
                var val = yield TMP1;
                console.log( val );
        }
        catch (err) {
                // STATE *3*
                console.log( "Oops:", err );
                return false;
        }
}
```

Note: For more accurate illustration, we split up the `val = yield request..` statement into two parts, using the temporary `TMP1` variable. `request(..)` happens in state `*1*`, and the assignment of its completion value to `val` happens in state `*2*`. We'll get rid of that intermediate `TMP1` when we convert the code to its non-generator equivalent.

In other words, `*1*` is the beginning state, `*2*` is the state if the `request(..)` succeeds, and `*3*` is the state if the `request(..)` fails. You can probably imagine how any extra `yield` steps would just be encoded as extra states.

Back to our transpiled generator, let's define a variable `state` in the closure we can use to keep track of the state:

```
function foo(url) {
        // manage generator state
        var state;

        // ..
}
```

Now, let's define an inner function called `process(..)` inside the closure which handles each state, using a `switch` statement:

```
// `request(..)` is a Promise-aware Ajax utility

function foo(url) {
	// manage generator state
	var state;

	// generator-wide variable declarations
	var val;

	function process(v) {
		switch (state) {
			case 1:
				console.log( "requesting:", url );
				return request( url );
			case 2:
				val = v;
				console.log( val );
				return;
			case 3:
				var err = v;
				console.log( "Oops:", err );
				return false;
		}
	}

	// ..
}
```

Each state in our generator is represented by its own `case` in the `switch` statement. `process(..)` will be called each time we need to process a new state. We'll come back to how that works in just a moment.

For any generator-wide variable declarations (`val`), we move those to a `var` declaration outside of `process(..)` so they can survive multiple calls to `process(..)`. But the "block scoped" `err` variable is only needed for the `*3*` state, so we leave it in place.

In state `*1*`, instead of `yield request(..)`, we did `return request(..)`. In terminal state `*2*`, there was no explicit `return`, so we just do a `return;` which is the same as `return undefined`. In terminal state `*3*`, there was a `return false`, so we preserve that.

Now we need to define the code in the *iterator* functions so they call `process(..)` appropriately:

```
function foo(url) {
	// manage generator state
	var state;

	// generator-wide variable declarations
	var val;

	function process(v) {
		switch (state) {
			case 1:
				console.log( "requesting:", url );
				return request( url );
			case 2:
```

```
                        val = v;
                        console.log( val );
                        return;
                case 3:
                        var err = v;
                        console.log( "Oops:", err );
                        return false;
                }
        }

        // make and return an iterator
        return {
                next: function(v) {
                        // initial state
                        if (!state) {
                                state = 1;
                                return {
                                        done: false,
                                        value: process()
                                };
                        }
                        // yield resumed successfully
                        else if (state == 1) {
                                state = 2;
                                return {
                                        done: true,
                                        value: process( v )
                                };
                        }
                        // generator already completed
                        else {
                                return {
                                        done: true,
                                        value: undefined
                                };
                        }
                },
                "throw": function(e) {
                        // the only explicit error handling is in
                        // state *1*
                        if (state == 1) {
                                state = 3;
                                return {
                                        done: true,
                                        value: process( e )
                                };
                        }
                        // otherwise, an error won't be handled,
                        // so just throw it right back out
                        else {
                                throw e;
                        }
                }
        };
}
```

How does this code work?

1. The first call to the *iterator*'s `next()` call would move the generator from the uninitialized state to state `1`, and then call `process()` to handle that state. The return value from `request(..)`, which is the promise for the Ajax response, is returned back as the `value` property from the `next()` call.
2. If the Ajax request succeeds, the second call to `next(..)` should send in the Ajax response value, which moves our state to `2`. `process(..)` is again called (this time with the passed in Ajax response value), and the `value` property returned from `next(..)` will be `undefined`.
3. However, if the Ajax request fails, `throw(..)` should be called with the error, which would move the state from `1` to `3` (instead of `2`). Again `process(..)` is called, this time with the error value. That `case` returns `false`, which is set as the `value` property returned from the `throw(..)` call.

From the outside -- that is, interacting only with the *iterator* -- this `foo(..)` normal function works pretty much the same as the `*foo(..)` generator would have worked. So we've effectively "transpiled" our ES6 generator to pre-ES6 compatibility!
We could then manually instantiate our generator and control its iterator -- calling `var it = foo("..")` and `it.next(..)` and such -- or better, we could pass it to our previously defined `run(..)` utility as `run(foo,"..")`.

Automatic Transpilation

The preceding exercise of manually deriving a transformation of our ES6 generator to pre-ES6 equivalent teaches us how generators work conceptually. But that transformation was really intricate and very non-portable to other generators in our code. It would be quite impractical to do this work by hand, and would completely obviate all the benefit of generators.

But luckily, several tools already exist that can automatically convert ES6 generators to things like what we derived in the previous section. Not only do they do the heavy lifting work for us, but they also handle several complications that we glossed over.

One such tool is regenerator (https://facebook.github.io/regenerator/), from the smart folks at Facebook.

If we use regenerator to transpile our previous generator, here's the code produced (at the time of this writing):

```
// `request(..)` is a Promise-aware Ajax utility

var foo = regeneratorRuntime.mark(function foo(url) {
    var val;

    return regeneratorRuntime.wrap(function foo$(context$1$0) {
        while (1) switch (context$1$0.prev = context$1$0.next) {
        case 0:
            context$1$0.prev = 0;
            console.log( "requesting:", url );
```

```
            context$1$0.next = 4;
            return request( url );
        case 4:
            val = context$1$0.sent;
            console.log( val );
            context$1$0.next = 12;
            break;
        case 8:
            context$1$0.prev = 8;
            context$1$0.t0 = context$1$0.catch(0);
            console.log("Oops:", context$1$0.t0);
            return context$1$0.abrupt("return", false);
        case 12:
        case "end":
            return context$1$0.stop();
        }
    }, foo, this, [[0, 8]]);
});
```

There's some obvious similarities here to our manual derivation, such as the `switch` / `case` statements, and we even see `val` pulled out of the closure just as we did. Of course, one trade-off is that regenerator's transpilation requires a helper library `regeneratorRuntime` that holds all the reusable logic for managing a general generator / *iterator*. A lot of that boilerplate looks different than our version, but even then, the concepts can be seen, like with `context$1$0.next = 4` keeping track of the next state for the generator.
The main takeaway is that generators are not restricted to only being useful in ES6+ environments. Once you understand the concepts, you can employ them throughout your code, and use tools to transform the code to be compatible with older environments.

This is more work than just using a `Promise` API polyfill for pre-ES6 Promises, but the effort is totally worth it, because generators are so much better at expressing async flow control in a reason-able, sensible, synchronous-looking, sequential fashion.
Once you get hooked on generators, you'll never want to go back to the hell of async spaghetti callbacks!

Review

Generators are a new ES6 function type that does not run-to-completion like normal functions. Instead, the generator can be paused in mid-completion (entirely preserving its state), and it can later be resumed from where it left off.

This pause/resume interchange is cooperative rather than preemptive, which means that the generator has the sole capability to pause itself, using the `yield` keyword, and yet the *iterator* that controls the generator has the sole capability (via `next(..)`) to resume the generator.
The `yield` / `next(..)` duality is not just a control mechanism, it's actually a two-way message passing mechanism. A `yield ..` expression essentially pauses waiting for a value, and the next `next(..)` call passes a value (or implicit `undefined`) back to that paused `yield` expression.
The key benefit of generators related to async flow control is that the code inside a generator expresses a sequence of steps for the task in a naturally sync/sequential fashion. The trick is that we

essentially hide potential asynchrony behind the `yield` keyword -- moving the asynchrony to the code where the generator's *iterator* is controlled.

In other words, generators preserve a sequential, synchronous, blocking code pattern for async code, which lets our brains reason about the code much more naturally, addressing one of the two key drawbacks of callback-based async.