# The 80/20 Guide to ES2015 Generators

Valeri Karpov

# Table of Contents

# Chapter 1: Getting Started

Generators are a powerful new feature in ES2015. Generators are far from a new programming construct - they first appeared in 1975 and Python has had them since Python 2.2 in 2001. However, as you'll see, generators are even more powerful in an event-driven language like JavaScript. In JavaScript (assuming Node.js >= 4.0.0), a **generator function** is defined as shown below.

```
const generatorFunction = function*() {
  console.log('Hello, World!');
};
```

However, if you run `generatorFunction`, you'll notice that the return value is an object.

```
$ node
> var generatorFunction = function*() { console.log('Hello, World!'); };
undefined
> generatorFunction()
{}
```

That's because a generator function creates and returns a **generator object**. Typically, the term **generator** refers to a generator object rather than a generator function. A generator object has a single function, `next()`. If you execute the generator object's `next()` function, you'll notice that Node.js printed 'Hello, World!' to the screen.

```
$ node
> var generatorFunction = function*() { console.log('Hello, World!'); };
undefined
> generatorFunction()
{}
> generatorFunction().next()
Hello, World!
{ value: undefined, done: true }
>
```

Notice that `next()` returned an object, `{ value: undefined, done: true }`. The meaning of this object is tied to the `yield` keyword. To introduce you to the `yield` keyword, consider the following generator function.

```
const generatorFunction = function*() {
  yield 'Hello, World!';
};
```

Let's see what happens when you call `next()` on the resulting generator.

```
$ node
> var generatorFunction = function*() { yield 'Hello, World!'; };
undefined
> var generator = generatorFunction();
undefined
> generator.next();
{ value: 'Hello, World!', done: false }
> generator.next();
{ value: undefined, done: true }
>
```

Notice that, the first time you call `generator.next()`, the `value` property is equal to the string your generator function yielded. You can think of `yield` as the generator-specific equivalent of the `return` statement.

You might be wondering why the return value of `generator.next()` has a `done` property. The reason is tied to why `yield` is different from `return`.

## `yield` vs `return`

The `yield` keyword can be thought of as a `return` that allows **re-entry**. In other words, once `return` executes, the currently executing function is done forever. However, when you call `generator.next()`, the JavaScript interpreter executes the generator function until the first `yield` statement. When you call `generator.next()` again, the generator function picks up where it left off. You can think of a generator as a function that can "return" multiple values.

```
const generatorFunction = function*() {
  let message = 'Hello';
  yield message;
  message += ', World!';
  yield message;
};

const generator = generatorFunction();
// { value: 'Hello', done: false };
const v1 = generator.next();
// { value: 'Hello, World!', done: false }
const v2 = generator.next();
// { value: undefined, done: true }
const v3 = generator.next();
```

### Re-entry

The most important detail from the above example is that, when `yield` executes, the generator function stops executing until the next time you call `generator.next()`. You can call `generator.next()` whenever you want, even in a `setTimeout()`. The JavaScript interpreter will re-enter the generator function with the same state that it left off with.

```
const generatorFunction = function*() {
  let i = 0;
  while (i < 3) {
    yield i;
    ++i;
  }
};

const generator = generatorFunction();

let x = generator.next(); // { value: 0, done: false }
setTimeout(() => {
  x = generator.next(); // { value: 1, done: false }
  x = generator.next(); // { value: 2, done: false }
  x = generator.next(); // { value: undefined, done: true }
}, 50);
```

### `yield` vs `return` revisited

You may be wondering what happens when you use `return` instead of `yield` in a generator. As you might expect, `return` behaves similarly to `yield`, except for done is set to true.

```
const generatorFunction = function*() {
  return 'Hello, World!';
};

const generator = generatorFunction();

// { value: 'Hello, World!', done: true }
const v = generator.next();
```

## Case Study: Async Fibonacci

The fact that you can execute `generator.next()` asynchronously hints at why generators are so useful. You can execute `generator.next()` synchronously or asynchronously without changing the implementation of the generator function.

For instance, lets say you wrote a generator function that computes the Fibonacci Sequence. Note that generator functions can take parameters like any function.

```
const fibonacciGenerator = function*(n) {
  let back2 = 0;
  let back1 = 1;
  let cur = 1;
  for (let i = 0; i < n - 1; ++i) {
    cur = back2 + back1;
    back2 = back1;
    back1 = cur;
    yield cur;
  }

  return cur;
};
```

You could compute the n-th Fibonacci number synchronously using the code below.

```
const fibonacci = fibonacciGenerator(10);
let it;
for (it = fibonacci.next(); !it.done; it = fibonacci.next()) {}
it.value; // 55, the 10th fibonacci number
```

However, computing the n-th Fibonacci number synchronously is not a hard problem without generators. To make things interesting, let's say you wanted to compute a very large Fibonacci number **without blocking the event loop**. Normally, a JavaScript for loop would block the event loop. In other words, no other JavaScript code can execute until the for loop in the previous example is done. This can get problematic if you want to compute the 100 millionth Fibonacci number in an Express route handler. Without generators, breaking up a long-running calculation can be cumbersome.

However, since you have a generator function that yields after each iteration of the for loop, you can call `generator.next()` in a `setInterval()` function. This will compute the next Fibonacci number with each iteration of the event loop, and so won't prevent Node.js from responding from incoming requests. You can make your Fibonacci calculation asynchronous without changing the generator function!

```
const fibonacci = fibonacciGenerator(10);
// And compute one new Fibonacci number with each iteration
// through the event loop.
const interval = setInterval(() => {
  const res = fibonacci.next();
  if (res.done) {
    clearInterval(interval);
    res.value; // 55, the 10th fibonacci number
  }
}, 0);
```

# For/Of Loops

Remember the for loop you saw for exhausting the Fibonacci generator?

```
for (it = fibonacci.next(); !it.done; it = fibonacci.next()) {}
```

This for loop is a perfectly reasonable way of going through every value of the generator. However, ES2015 introduces a much cleaner mechanism for looping through generators: the `for-of` loop.

```
let fibonacci = fibonacciGenerator(10);
for (const x of fibonacci) {
  x; // 1, 1, 2, 3, 5, ..., 55
}
```

## Iterators and Iterables

For/Of loops aren't just for generators. A generator is actually an instance of a more general ES2015 concept called an iterator. An **iterator** is any JavaScript object that has a `next()` function that returns `{ value: Any, done: Boolean }`. A generator is one example of an iterator. You can also iterate over arrays:

```
for (const x of [1, 2, 3]) {
  x; // 1, 2, 3
}
```

However, For/Of loops don't operate on iterators, they operate on iterables. An **iterable** is an object that has a `Symbol.iterator` property which is a function that returns an iterator. In other words, when you execute a For/Of loop, the JavaScript interpreter looks for a `Symbol.iterator` property on the object you're looping `of`.

```
let iterable = {};
for (const x of iterable) {} // Throws an error

// But once you add a Symbol.iterator property, everything works!
iterable[Symbol.iterator] = function() {
  return fibonacciGenerator(10);
};
for (const x of iterable) {
  x; // 1, 1, 2, 3, 5, ..., 55
}
```

## A Brief Overview of Symbols

**Symbols** are another new feature in ES2015. Since this book is about generators, we won't explore symbols in depth, just enough to understand what the mysterious `iterable[Symbol.iterator]` code in the previous example is about.

You can think of a symbol as a unique identifier for a key on an object. For instance, suppose you wrote your own programming language and defined an iterable as an object that had a property named `iterator`. Now, every object that has a property named `iterator` would be an iterable, which could lead to some unpredictable behavior. For instance, suppose you added a property named `iterator` to an array - now you've accidentally broken for/of loops for that array!

Symbols protect you from the issue of accidental string collision. No string key is equal to `Symbol.iterator`, so you don't have to worry about accidentally breaking an iterable. Furthermore, symbols don't appear in the output of `Object.keys()`.

```
Symbol.iterator; // Symbol(Symbol.iterator)

let iterable = {};
iterable[Symbol.iterator] = function() {
  return fibonacciGenerator(10);
};

iterable.iterator; // undefined
Object.keys(iterable); // Empty array!
```

## Iterables and Generators

The most important detail to note about generators and iterables is that *generator objects* are iterables, not *generator functions*. In order words, you can't run a for/of loop on a generator function.

```
fibonacciGenerator[Symbol.iterable]; // Undefined
fibonacciGenerator(10)[Symbol.iterable]; // Function

for (const x of fibonacciGenerator) {} // Error!
for (const x of fibonacciGenerator(10)) {} // Ok
```

You may find it strange that the generator's `Symbol.iterator` function returns itself given that generator functions are not iterable. One reason for this decision is that a generator function can take parameters. For instance, looping over `fibonacciGenerator(10)` would not give the same results as looping over `fibonacciGenerator(11)`.

The second most important detail to note about generators and iterables is that `generator[Symbol.iterator]` is a function that returns the generator itself. This means that you can't loop over the same generator twice. Once a generator is done, subsequent for/of loops will exit immediately.

```
const fibonacci = fibonacciGenerator(10);
fibonacci[Symbol.iterator]() === fibonacci; // true
for (const x of fibonacci) {
  // 1, 1, 2, 3, 5, ..., 55
}
for (const x of fibonacci) {
  // Doesn't run!
}
```

# Error Handling

One detail that has been glossed over so far is how generators handle exceptions. What happens when you divide by zero in a generator? As you might have guessed, the `generator.next()` call throws an error.

```
const generatorFunction = function*() {
  throw new Error('oops!');
};

const generator = generatorFunction();

// throws an error
generator.next();
```

The error's stack trace reflects the fact that `next()` was the function that called the function that threw the error. In particular, if you call `next()` asynchronously, you will lose the original stack trace.

```
const generatorFunction = function*() {
  throw new Error('oops!');
};

const generator = generatorFunction();

setTimeout(() => {
  try {
    generator.next();
  } catch (err) {
    /**
     * Error: oops!
     * at generatorFunction (book.js:2:15)
     * at next (native)
     * at null._onTimeout (book.js:18:21)
     * at Timer.listOnTimeout (timers.js:89:15)
     */
    err.stack;
  }
}, 0);
```

## Re-entry With Error

When you think of generators, you need to think of 2 functions: the generator function itself, and the function that's calling `next()` on the generator. When the generator function calls `yield` or `return`, the calling function regains control. When the calling function calls `next()`, the generator function regains starts running again. There's another way the calling function can give control back to the generator function: the `throw()` function.

The `throw()` function is a way for the calling function to tell the generator function that something went wrong. In the generator function, this will look like the `yield` statement threw an error. You can then use try/catch to handle the error in the generator function. As you'll see in the next section, this pattern is indispensable for working with asynchronous code and generators.

```
const fakeFibonacciGenerator = function*() {
  try {
    yield 3;
  } catch (error) {
    error; // Error: Expected 1, got 3
  }
};
const fibonacci = fakeFibonacciGenerator();

const x = fibonacci.next();
fibonacci.throw(new Error(`Expected 1, got ${x.value}`));
// { value: undefined, done: true }
fibonacci.next();
```

## Case Study: Handling Async Errors

Remember that there are two functions involved in generator functions: the generator function itself, and the function that calls `next()` on the generator object. So far in this book, the function that calls `next()` hasn't done any real work. The most complex example is the async Fibonacci example, which acted as a scheduler for the Fibonacci generator.

One pivotal feature of generators is that the `next()` function can take a parameter. That parameter then becomes the return value of the `yield` statement in the generator function itself!

```
const generatorFunction = function*() {
  const fullName = yield ['John', 'Smith'];
  fullName; // 'John Smith'
};

const generator = generatorFunction();
// Execute up to the first `yield`
const next = generator.next();
// Join ['John', 'Smith'] => 'John Smith' and use it as the
// result of `yield`, then execute the rest of the generator function
generator.next(next.value.join(' '));
```

Once you combine this feature with the `throw()` function, you have everything you need to have the generator function `yield` whenever it needs to do an asynchronous operation. The calling function can then execute the asynchronous operation, `throw()` any errors that occurred, and return the result of the async operation using `next()`.

This means that your generator function doesn't need to worry about callbacks. The calling function can be responsible for running asynchronous operations and reporting any errors back to the generator function. For instance, the below example shows how to run a generator function that yields an asynchronous function without any errors.

```
const async = function(callback) {
  setTimeout(() => callback(null, 'Hello, Async!'), 10);
};

const generatorFunction = function*() {
  const v = yield async;
  v; // 'Hello, Async!'
};

const generator = generatorFunction();
const res = generator.next();
res.value(function(error, res) {
  generator.next(res);
});
```

Now suppose that the `async` function returns an error. The calling function can then call `throw()` on the generator, and now your generator function can handle this asynchronous operation with try/catch! As you'll see in the coroutines chapter, this idea is the basis of the co library.

```
const async = function(callback) {
  setTimeout(() => callback(new Error('Oops!')), 10);
};

const generatorFunction = function*() {
  try {
    yield async;
  } catch (error) {
    error; // Error: Oops!
  }
};

const generator = generatorFunction();
const res = generator.next();
res.value(function(error, res) {
  generator.throw(error);
});
```

# Chapter 2: Asynchronous Coroutines

In chapter 1, you saw how to `yield` an asynchronous function from a generator. The calling function would then execute the asynchronous function and resume the generator function when the asynchronous function was done. This pattern is an instance of an old (1958) programming concept known as a coroutine. A **coroutine** is a function that can suspend its execution and defer to another function. As you might have guessed, generator functions are coroutines, and the `yield` statement is how a generator function defers control to another function. You can think of a coroutine as two functions running side-by-side, deferring control to each other at predefined points.

So why are coroutines special? In JavaScript, you typically need to specify a callback for asynchronous operations. For instance, if you use the `superagent` HTTP library to make an HTTP request to Google's home page, you would use code similar to what you see below.

```
superagent.get('http://google.com', function(error, res) {
  // Handle error, use res
});
```

By yielding asynchronous operations, you can write asynchronous operations without callbacks. However, remember that a coroutine involves two functions: the generator function, and the function that calls `next()` on the generator. When your generator function yields an asynchronous operation, the calling function needs to handle the asynchronous operation and resume the generator when the asynchronous operation completes.

The most popular library for handling generator functions that yield asynchronous operations is called co. Here's what getting the HTML for Google's home page looks like in co. Looks cool, right? The below code is still asynchronous, but looks like synchronous code. In this chapter, you'll learn about how co works by writing your own co.

```
const co = require('co');
const superagent = require('superagent');

co(function*() {
  const html = (yield superagent.get('http://www.google.com')).text;
  // HTML for Google's home page
  html;
});
```

## Promises and Thunks

The purpose of this chapter is to build your own co. But first, there's one key term that we need to clarify: what sort of asynchronous operations can you yield to co? The examples you've seen so far

in this book have been cherry-picked. For instance, recall the asynchronous function from the asynchronous errors section.

```
const async = function(callback) {
  setTimeout(() => callback(new Error('Oops!')), 10);
};
```

The above function is asynchronous, but not representative of asynchronous functions as a whole. For instance, the `superagent.get` function takes a parameter as well as a callback:

```
superagent.get('http://google.com', function(error, res) {
// Handle error, use res
});
```

The `async` function is an example of a thunk. A **thunk** is an asynchronous function that takes a single parameter, a callback. The `superagent.get()` function is *not* a thunk, because it takes 2 parameters, a url and a callback.

Thunks may seem limited, but with arrow functions you can easily convert any asynchronous function call to a thunk.

```
co(function*() {
  yield (callback) => { superagent.get('http://google.com', callback); };
});
```

There are also libraries that can convert asynchronous functions to thunks for you. The original author of co, TJ Holowaychuk, also wrote a library called thunkify. As the name suggests, thunkify converts a general asynchronous function into a thunk for use with co. The `thunkify` function takes a single parameter, an asynchronous function, and returns a function that returns a thunk. Below is how you would use `thunkify()` with co.

```
const co = require('co');
const superagent = require('superagent');
const thunkify = require('thunkify');

co(function*() {
  const thunk = thunkify(superagent.get)('http://www.google.com');
  // function
  typeof thunk;
  // A function's length property contains the number of parameters
  // In this case, 1
  thunk.length;
  const html = yield thunk;
  // HTML for Google's home page
  html;
}).catch(error => done(error));
```

Thunkify may seem confusing because of the numerous layers of function indirection. Don't worry, thunkify is not that complex, you can implement your own in 9 lines. Below is a simple implementation of thunkify.

```
const co = require('co');
const superagent = require('superagent');

const thunkify = function(fn) {
  // Thunkify returns a function that takes some arguments
  return function() {
    // The function gathers the arguments
    const args = [];
    for (const arg of arguments) {
      args.push(arg);
    }
    // And returns a thunk
    return function(callback) {
      // The thunk calls the original function with your arguments
      // plus the callback
      return fn.apply(null, args.concat([callback]));
    };
  };
};

co(function*() {
  const thunk = thunkify(superagent.get)('http://www.google.com');
  //
  const html = yield thunk;
  // HTML for Google's home page
  html;
});
```

If thunkify makes thunks so easy, why do you ever need anything else? As is often the case with JavaScript, the problem is the `this` keyword. The below example shows that when you call `thunkify()` on a function, that function loses its value of `this`.

```
class Test {
  async(callback) {
    return callback(null, this);
  }
}

co(function*() {
  const test = new Test();
  const res = yield thunkify(test.async)();
  // Woops, res refers to global object rather than the `test` variable
  assert.ok(res !== test);
  done();
});
```

Why does thunkify lose the function's value of `this`? Because the JavaScript language spec treats calling `a.b();` different from `var c = a.b; c();`. When you call a function as a member, like the `a.b();` case, `this` will equal `a` in the function call. However, `var c = a.b; c();` does not call a function as a member, so `this` refers to the global object in `c`. The latter case also applies when you pass a function as a parameter to another function, like you do with `thunkify()`.

There are ways to make thunkify work better. For instance, in the previous example, you could use `.bind()`.

```
const res = yield thunkify(test.async.bind(test))();
```

However, `bind()` gets to be very confusing when you have *chained* function calls. A chained function call takes the form `a.b().c().d()`, and the `b()`, `c()`, and `d()` function calls are "chained" together. This API pattern is often used in JavaScript for building up complex objects, like HTTP requests or MongoDB queries. For instance, superagent has a chainable API for building up HTTP requests.

```
// Create an arbitrary complex HTTP request to show how superagent's
// request builder works.
superagent.
  get('http://google.com').
  // Set the HTTP Authorization header
  set('Authorization', 'MY_TOKEN_HERE').
  // Only allow 5 HTTP redirects before failing
  redirects(5).
  // Add `?color=blue` to the URL
  query({ color: blue }).
  // Send the request
  end(function(error, res) {});
```

Let's say you wanted to thunkify the above code. Where would you need to use `.bind()` and what would you need to `bind()` to? The answer is not obvious unless you read superagent's code. You need to `bind()` to the return value of `superagent.get()`.

```
co(function*() {
  const req = superagent.get('http://google.com');
  const res = yield thunkify(req.query({ color: blue }).end.bind(req));
});
```

Thunkify and thunks in general are an excellent fallback, but co supports a better asynchronous primitive: promises. A **promise** is an object that has a `.then()` function that takes two functions as parameters.

- `onFulfilled`: called if the asynchronous operation succeeds.
- `onRejected`: called if the asynchronous operation failed.

You can think of promises as an object wrapper around a single asynchronous operation. Once you call `.then()`, the asynchronous operation starts. Once the asynchronous operation completes, the promise then calls either `onFulfilled` or `onRejected`.

For example, each function call in the superagent HTTP request builder returns a promise that you can `yield`.

```
co(function*() {
  // `superagent.get()` returns a promise, because the `.then` property
  // is a function.
  superagent.get('http://www.google.com').then;
  co(function*() {
    // Works because co is smart enough to look for a `.then()` function
    const res = yield superagent.
      get('http://www.google.com').
      query({ color: 'blue' });
  });
});
```

Much easier than using thunkify! More importantly, you don't have to worry about messing up the value of `this` because you aren't passing a function as a parameter. The downside of promises, though, is that you rely on the function itself to return a promise. When you use thunkify, you make no assumptions about the return value of the function you're calling. However, many popular Node.js libraries, like superagent, the redis driver, and the MongoDB driver, all have mechanisms for promise-based APIs.

Promises are a deep subject and what you've seen thus far is just the tip of the iceberg. To use co, all you need to know is that a promise is an object with a `.then()` function that takes 2 function parameters: `onFulfilled` and `onRejected`. For instance, below is an example of a minimal promise that's compatible with co.

```
const promise = {
  then: function(onFulfilled, onRejected) {
    setTimeout(() => onFulfilled('Hello, World!'), 0);
  }
};

co(function*() {
  const str = yield promise;
  assert.equal(str, 'Hello, World!');
});
```

## Write Your Own Co

Now that you've seen how thunks and promises work, it's time to apply the fundamentals of generators to write your own minimal implementation of co. To avoid confusion, your version will be called "fo" (pronounced like "faux").

The v1 implementation of fo is short, but will utilize all the concepts that you've learned thus far. You'll use generator functions, `generator.next()`, `generator.throw()`, thunks, and promises. Ready? The implementation is...

```
const fo = function(generatorFunction) {
  const generator = generatorFunction();
  next();

  // Call next() or throw() on the generator as necessary
  function next(v, isError) {
    const res = isError ? generator.throw(v) : generator.next(v);
    if (res.done) {
      return;
    }
    handleAsync(res.value);
  }

  // Handle the result the generator yielded
  function handleAsync(async) {
    if (async && async.then) {
      handlePromise(async);
    } else if (typeof async === 'function') {
      handleThunk(async);
    } else {
      next(new Error(`Invalid yield ${async}`), true);
    }
  }

  // If the generator yielded a promise, call `.then()`
  function handlePromise(async) {
    const onRejected = function(error) {
      next(error, true);
    };
    async.then(next, onRejected);
  }

  // If the generator yielded a thunk, call it
  function handleThunk(async) {
    async((error, v) => {
      if (error) {
        next(error, true);
        return;
      }
      next(v);
    });
  }
};

// fo in action
fo(function*() {
  const html = (yield superagent.get('http://www.google.com')).text;
});
```

Let's take a closer look at how the `fo()` function works. The first step is to create a generator from the provided generator function. Once you have a generator, you need to use the `next()` function to kick off the generator's execution. The `next()` function calls `generator.next()` to start off the generator. Any time the generator yields, the `fo()` function calls `handleAsync()`, which is responsible for handling the asynchronous operations the generator yields. In particular, `handlePromise()` handles any promises that the generator yields, and `handleThunk()` handles any thunks.

Let's see how `fo()` works with a simple error. In the below example, you make an HTTP request to a nonexistent URL. Superagent will fail, and so the promise calls its `onRejected()` function. The `fo()` function will then call `next()` with `isError` set to true. The internal `next()` function then calls `generator.throw()` to trigger an error in the generator, which you can then try/catch.

```
fo(function*() {
  try {
    // First iteration of `next()` stops here,
    // calls `.then()` on the promise
    const res = yield superagent.get('http://doesnot.exist.baddomain');
  } catch (error) {
    // The promise was rejected, so fo calls `generator.throw()` and
    // you end up here.
  }

  // Second iteration of `next()` stops here, `.then()` on the promise
  const res = yield superagent.get('http://www.google.com');
  res.text;
  // Third iteration of `next()` stops here because generator is done
});
```

One neat pattern that illustrates the power of asynchronous coroutines is retrying failed HTTP requests. If the server you're trying to reach is unreliable, you may want to retry requests a fixed number of times before giving up. Without generators, retrying requests involves a lot of recursion and careful design decisions. With generators and `fo()` (or co), all you need to retry requests is a for loop as shown below.

```
fo(function*() {
  const url = 'http://doesnot.exist.baddomain';
  const NUM_RETRIES = 3;
  let res;
  let i;
  for (i = 0; i < 3; ++i) {
    try {
      // Going to yield 3 times, and `fo()` will call `generator.throw()`
      // 3 times because superagent will fail every time
      res = yield superagent.get(url);
    } catch (error) { /* retry */ }
  }

  // res is undefined - retried 3 times with no results
});
```

# Limitations

The v1 implementation of fo is simple, clean, and gets you 80% of the way to writing your own co. However, co has several subtle features that become indispensable when you try to write a real application.

One particular edge case that we glossed over in the "Write Your Own Co" section is what happens when there's an uncaught error in the generator. In the v1 implementation of fo, the uncaught error will crash the process. Crashing the process isn't the worst possible behavior, but, as you'll see in the "Real Implementation of Co" section, co provides a neat way to catch any uncaught errors in the generator function.

```
try {
  fo(function*() {
    // This will throw an uncaught asynchronous exception
    // and crash the process!
    yield superagent.get('http://doesnot.exist.baddomain');
  });
} catch (error) {
  // This try/catch won't catch the error within the `fo()` call!
}
```

Another limitation is the ability to use helper functions that yield. For instance, suppose you wanted to write a `retry()` helper function that retried an asynchronous operation a fixed number of times for you. You might try implementing retry as a plain old function (as opposed to a generator function) and then quickly realize that you can't yield from a normal function. You might then try implementing the `retry()` function as shown below. But alas! Your `fo()` v1 doesn't supporting yielding generators!

```
// Needs to be a generator function so you can `yield` in it.
const retry = function*(fn, numRetries) {
  for (let i = 0; i < numRetries; ++i) {
    try {
      const res = yield fn();
      return res;
    } catch (error) {}
  }
  throw new Error(`Retried ${numRetries} times`);
};

fo(function*() {
  const url = 'http://www.google.com';
  // Fo's `handleAsync` function will throw because you're
  // yielding a generator function!
  const res = yield retry(() => superagent.get(url), 3);
});
```