The 80/20 Guide to ES2015 Generators

Valeri Karpov

Table of Contents

How To Use This Book

This is not just another tech book that sits up on your bookshelf gathering dust. I think of this ebook as something halfway between a blog post and a full book: focused and concise like a blog post, in-depth and rigorous like a pure math textbook. The purpose of this ebook is to take you from a generators novice to someone who would be comfortable discussing co internals in 1 to 2 hours. This ebook is meant to be read in 1-2 sessions (its only 40 pages!), although you may also choose to read one chapter at a time.

What is this ebook focused on?

- 80/20 principle. There's a lot of tooling related to generators out there: transpilers, modules, build systems, etc. This book is focused solely on generators and other features defined in the ES2015 specification. The **only** dependency is Node.js >= 4.0.0 and npm. In particular, this ebook will **not** use webpack, react, babel, gulp, grunt, TypeScript, CoffeeScript, AngularJS, Dart, or any other framework, preprocessor, or hype train.
- The co module and asynchronous coroutines. To better understand how generators work in an asynchronous language like JavaScript, you'll write your own minimal version of co from scratch. Your co implementation will enable you to write asynchronous code without callbacks. For instance,

```
const superagent = require('superagent');
co(function*() {
  // Make an HTTP request to google's home page
  const google = (yield superagent.get('http://google.com')).text;
  const regexp = /google/i;
  // The number of times "Google" appears on google.com
  regexp.match(google).length;
});
```

- Composing asynchronous coroutines. You'll learn about the generator-based server-side web framework koa, and write your own minimal koa implementation.

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 the ability to return promises for asynchronous operations.

Creating your own promises is easy. Promises are a core part of ES2015, so you don't have to include any libraries. Below is an example of how to create an ES2015 promise. The `Promise` constructor takes a single function, called a **resolver**, which takes two function parameters, `resolve` and `reject`. The resolver is responsible for executing the asynchronous operation and calling `resolve()` if the operation succeeded or `reject()` if it failed.

```
// The resolver function takes 2 parameters, a `resolve()` function
// and a `reject()` function.
const resolver = function(resolve, reject) {
  // Call `resolve()` asynchronously with a value
  setTimeout(() => resolve('Hello, World!'), 5);
};
const promise = new Promise(resolver);
promise.then(function(res) {
  // The promise's `onFulfilled` function gets called with
  // the value the resolver passed to `resolve()`. In this
  // case, the string 'Hello, World!'
  res;
});
```

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) {
    async.then(next, (error) => next(error, true));
  }

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

Finally, fo v1 doesn't allow any parallelism. Asynchronous operations must be executed one after the other, there's no way to execute 2 requests in parallel. Suppose you wanted to load Google and Amazon's home pages in parallel as shown below. As you'll see in the "Real Implementation of Co" section, co gives you a convenient mechanism to execute requests in parallel.

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

Real Implementation of Co

The key decision that makes co 4.x overcome all the limitations of fo v1 is that the `co()` function itself returns a promise, and converts all asynchronous operations into promises. If fo() returns a promise, then you can handle the case where the generator yields a generator function by just calling fo() on the yielded generator function. Promises also provide a mechanism for handling errors: the `reject()` function. You can wrap your calls to `generator.next()` and `generator.throw()` in a try/catch, and reject the promise if the generator threw. Below is the implementation of fo v2, which uses promises internally and returns a promise.

```
const fo = function(input) {
  const isGenerator = (v) => typeof v.next === 'function';
  const isGeneratorFunction =
    (v) => v.constructor && v.constructor.name === 'GeneratorFunction';

  let generator;
  if (isGenerator(input)) generator = input;
  if (isGeneratorFunction(input)) generator = input();
  if (!generator) throw `Invalid parameter to fo() ${input}`;

  return new Promise((resolve, reject) => {
    next();

    // Call next() or throw() on the generator as necessary
    function next(v, isError) {
      let res;
      try {
        res = isError ? generator.throw(v) : generator.next(v);
      } catch (error) {
        return reject(error);
      }
      if (res.done) {
        return resolve(res.value);
      }
      toPromise(res.value).then(next, (error) => next(error, true));
    }

    // Convert v to a promise. If invalid, returns a rejected promise
    function toPromise(v) {
      if (isGeneratorFunction(v) || isGenerator(v)) return fo(v);
      if (v.then) return v;
      if (typeof v === 'function') {
        return new Promise((resolve, reject) => {
          v((error, res) => error ? reject(error) : resolve(res));
        });
      }
      if (Array.isArray(v)) return Promise.all(v.map(toPromise));
      return Promise.reject(new Error(`Invalid yield ${v}`));
    }
  });
};
```

Let's take a look at how this new version of fo resolves the major limitations of the first version. First off, let's take a look at error handling. The key idea for error handling in fo is the below code.

```
let res;
try {
  res = isError ? generator.throw(v) : generator.next(v);
} catch (error) {
  return reject(error);
}
```

The `fo()` function may be asynchronous, but every error that can occur in your generator function happens synchronously in the above try block. For instance, suppose you made an HTTP request to a bad URL and didn't wrap your `superagent.get()` call in a try/catch. In that case, fo will reject the promise and trigger your `onRejected` handler.

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

fo(function*() {
  const html = yield superagent.get('http://doesnot.exist.baddomain');
}).then(null, (error) => {
  // Caught the HTTP error!
});
```

ES2015 promises have a handy helper function called `.catch()`. While `catch()` may sound intimidating, it is just a convenient shorthand for `.then()` with no `onFulfilled` handler. In other words using `.catch()` as shown below:

```
promise.catch(errorHandler);
```

is just a convenient shorthand for the below code.

```
promise.then(null, errorHandler);
```

Below is the previous example re-written to use `.catch()`.

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

fo(function*() {
  const html = yield superagent.get('http://doesnot.exist.baddomain');
}).catch((error) => {
  // Caught the HTTP error!
});
```

Fo's error handling abilities aren't limited to asynchronous errors. Since you wrapped `generator.next()` in a try/catch, fo reports **any** uncaught errors that occur when executing the generator to your `onRejected` handler. For instance, say you accidentally access a nonexistent property and get the dreaded `TypeError: Cannot read property 'X' of undefined` error. Fo will report that error to `onRejected` as well!

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

fo(function*() {
  const html = yield superagent.get('http://www.google.com');
  // Throws a TypeError, because `html.notARealProperty` is undefined
  const v = html.notARealProperty.test;
}).catch((error) => {
  // Caught the TypeError!
});
```

The new implementation of fo also enables you to yield generators and generator functions. Since `fo()` returns a promise, all you need to do is detect when the generator yields another generator and use `fo()` to convert that generator to a promise. Fo v2 uses the below functions `isGenerator()` and `isGeneratorFunction()` to determine if the yielded value is a generator or generator function, respectively.

Co also has similar helper functions, however, co's checks are more robust. These functions are meant to be examples, don't use them in production!

```
const isGenerator = (v) => typeof v.next === 'function';
const isGeneratorFunction =
  (v) => v.constructor && v.constructor.name === 'GeneratorFunction';
```

Now that you can check if a value is a generator or a generator function, the `toPromise()` function can call `fo()` recursively to convert the generator you yielded into a promise.

```
// Convert v to a promise. If invalid, returns a rejected promise
function toPromise(v) {
  if (isGeneratorFunction(v) || isGenerator(v)) return fo(v);
}
```

And now, you can write helper functions that can yield, as long as the helper function is a generator function too.

```
const get = function*() {
  return (yield superagent.get('http://www.google.com')).text;
};
// fo v2 in action. Note that you're yielding a generator!
fo(function*() {
  // Get the HTML for Google's home page
  const html = yield get();
}).catch((error) => done(error));
```

This new version of fo also enables you to execute requests in parallel. If you yield an array, fo will execute the array elements in parallel and return the results as an array. The magic that makes this work is the `toPromise()` function's array handler.

```
// Convert v to a promise. If invalid, returns a rejected promise
function toPromise(v) {
  if (isGeneratorFunction(v) || isGenerator(v)) return fo(v);
  if (v.then) return v;
  if (typeof v === 'function') {
    return new Promise((resolve, reject) => {
      v((error, res) => error ? reject(error) : resolve(res));
    });
  }
  // Magic array handler
  if (Array.isArray(v)) return Promise.all(v.map(toPromise));
  return Promise.reject(new Error(`Invalid yield ${v}`));
}
```

The `Promise.all()` function is yet another new feature in the ES2015 spec. The `Promise.all()` function takes an array of promises and converts them into a single promise. The `Promise.all()` promise's `onFulfilled` function gets called when every promise in the array is resolved, and its `onRejected` function gets called whenever any of the promises in the array is rejected.

When your generator function yields an array, the `toPromise()` function first uses `.map()` to convert every element in the array to a promise, and then passes that array to `Promise.all()`. This means that the HTTP requests below execute in parallel.

```
const superagent = require('superagent');
fo(function*() {
  // Parallel HTTP requests!
  const res = yield [
        superagent.get('http://www.google.com'),
        superagent.get('http://www.amazon.com')
      ];
});
```

Congratulations, you've successfully implemented your own co knockoff! Co supports all the features of fo, plus a few extra. Go take a look at the source code for co 4.x on GitHub, it should now look familiar. Don't forget the following key points for working with co.

- Error handling. Make sure to use `co().catch()`, this will enable you to catch all errors that occur in your generator function, not just promise rejections.
- Parallelism. Running requests in parallel with co is as simple as yielding an array. Using parallel requests where possible can give you a big performance boost.
- Internal error handling. If you don't want an error to stop execution of your generator function, use try/catch. With co, you can use try/catch to handle asynchronous errors.
- Helper functions. Helper functions can be generator functions as well, just be sure to use yield.

Below is an example of some of the above points in action.

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

co(function*() {
  let res;
  try {
    res = yield superagent.get('http://www.google.com');
  } catch (error) {
    res = getCached();
  }

  // Any errors with this get sent to handleError
  // The `persistToDb()` function is a generator function.
  yield persistToDb(res.text);
}).catch(handleError);
```

Chapter 3: Koa and Middleware

Co and the notion of asynchronous coroutines enable you to write asynchronous code with cleaner syntax. However, co is useful for more than just writing HTTP requests without callbacks. As you saw in the "Real Implementation of Co" section, co supports generators as an asynchronous primitive. You used this idea to write helper functions that were also generators. The idea of yielding generators is useful for composing asynchronous functions, which leads to the idea of middleware.

The concept of **middleware** in JavaScript is a sequence of functions where one function is responsible for calling the next function in the sequence. Conventional ES5-style middleware has the below form.

```
const middlewareFn = function(req, res, next) {
  // A middleware function takes in the request and response,
  // transforms them, and calls `next()` to trigger the next
  // function in the middleware chain.
  next();
};
```

However, this function composition paradigm has numerous limitations. Once a middleware function calls `next()`, it defers control to the next function in the sequence. There is no way for the next function to return any values or defer control back to the previous middleware. In other words, what if you wanted to write a middleware function that reported how long the rest of the sequence of functions took to complete?

```
const middlewareFn = (req, res, next) => {
  // next() is async and does not return a promise, nor does it
  // take a callback.
  next();
};
```

The `next()` call fires off the next middleware in the chain, but there is no way for the middleware function above to know when `next()` is done. However, suppose your middleware functions were generator functions rather than regular functions, and suppose `next()` returned a promise. If you wrapped the middleware generator functions in a `co()` call, you would now have the ability to compose asynchronous coroutines: the first coroutine calls the second and so on, and then the first coroutine picks up where it left off after the second is done.

The generator-based function composition library koa-compose does exactly this. The `next` parameter is a generator function that triggers the next generator in the sequence. In the below example, `middleware1` yields next to defer control to `middleware2`, which takes approximately 100ms to run, and picks up where it left off after `middleware2` is done.

```
const co = require('co');
const compose = require('koa-compose');

const middleware1 = function*(next) {
  const start = Date.now();
  yield next;
  // Approximately 100
  const end = Date.now() - start;
};

const middleware2 = function*(next) {
  // Yield a thunk that gets resolved after 100 ms
  yield callback => setTimeout(() => callback(), 100);
  yield next;
};

co(function*() {
  yield compose([middleware1, middleware2])();
});
```

The koa-compose Module

The koa-compose module that you saw in this chapter's introduction is trivial to implement. The key idea is that the `next` generator function that each middleware gets is a generator which, when completed, means that each subsequent middleware has completed. In other words, the `next` parameter to each generator function needs to be a generator derived from the next generator function in the sequence as shown below.

```
const compose = function(middleware) {
  const noop = function*() {};
  // compose returns a generator
  return function*() {
    let i = middleware.length;
    let next = noop;
    while (i--) {
      // Loop over generators from last to first. The `next` passed
      // to the i-th generator function is the (i+1)-th generator.
      next = middleware[i](next);
    }
    yield next;
  };
};
```

The real implementation of koa-compose is almost identical to the above implementation, just with better support for the `this` keyword. The `this` keyword will be important in the koa section. But, before you learn about koa, let's take a look at how error handling works in koa-compose.

Remember that koa-compose doesn't add any special error handling on top of co and generators. If the first middleware in the chain throws an error, the subsequent middleware functions never execute, and the `yield` statement that calls the composed function will throw an error.

```
const co = require('co');
const compose = require('koa-compose');

const middleware1 = function*(next) {
  throw new Error('oops');
};

const middleware2 = function*(next) {
  // never get here
};

co(function*() {
  // This will throw an error
  yield compose([middleware1, middleware2])();
}).catch(error => {
  // Error: oops
});
```

One neat feature of this middleware paradigm is that you can define error handling middleware using try/catch. Since the way you defer control to the next generator function in the sequence is by yielding on a generator, surrounding the `yield next` call in a try/catch will catch **any** errors that the subsequent functions in the sequence throw.

```
let error;
const middleware1 = function*(next) {
  try {
    // The `middleware2` generator function throws...
    yield next;
  } catch (err) {
    // which triggers this try/catch and records the error
    error = err;
  }
};

const middleware2 = function*(next) {
  throw new Error('Will get caught');
};

co(function*() {
  // `res` will be -1 since the final value returned by middleware1
  // is -1
  yield compose([middleware1, middleware2])();
  // error is now 'Error: Will get caught'
});
```

The Koa Web Framework

The most important application of koa-compose and generator-based middleware is a web framework called koa. Koa is a thin layer on top of koa-compose that makes it easier to write web servers using generator-based middleware. The below example creates a koa HTTP server that displays "Hello, World!" when you open `http://localhost:3000` in a browser.

```
const koa = require('koa');
// Create a new koa app
const app = koa();
// Each app has its own sequence of middleware. The `.use()`
// function adds a generator function to the middleware chain.
app.use(function*(next) {
  // `superagent.get('http://localhost:3000');` will now
  // return 'Hello, World!'
  this.body = 'Hello, World!';
});
const server = app.listen(3000);
```

The goal of this section is to write your own minimal take on koa. Koa is just a thin layer on top of koa-compose with some HTTP-specific syntactic sugar. Recall that koa-compose retains the same `this` across all middleware functions. In order to craft a response to an incoming HTTP request, koa requires you to set properties on `this`, also known as the koa **context**.

Before you implement your own minimal koa, there's a couple key concepts you should be aware of. First, generator functions have a context (the value of the `this` keyword) just like regular JavaScript functions. The easiest way to call a generator with a pre-defined value of `this` is the `call()` function defined on every JavaScript function.

```
const generatorFunction = function*() {
  console.log(this);
};

co(function*() {
  // Will print 'Hello, World!'
  yield generatorFunction.call('Hello, World!');
});
```

The other key concept is the Node.js HTTP package. The ability to start an HTTP server is part of core Node.js. Here's how you would start an HTTP server on port 3000 that prints out "Hello, World!" as a response to every request.

```
const http = require('http');
// Create a new server with a request handler. `req` represents
// the request, and `res` represents the response.
const server = http.createServer((req, res) => res.end('Hello, World!'));
// Listen on port 3000
server.listen(3000);
```

The downside of Node.js' vanilla HTTP package is that it doesn't provide an easy way to compose request handlers. You pass `createServer()` a function that takes in a request and a response, and you're responsible for figuring out what that function should be. Koa enables you to use koa-compose to compose generator functions into a route handler that's compatible with Node.js' HTTP server package. With that in mind, below is a minimal implementation of koa that is sufficient to run your basic "Hello, World" example.

```
const http = require('http');
const co = require('co');
const compose = require('koa-compose');

// The minimal koa implementation
const koa = function() {
  // List of middleware
  let middleware = [];

  // The real work is in this function, which creates the
  // actual HTTP server
  const listen = (port) => {
    // First, koa-compose all the middleware together
    const composedMiddleware = compose(middleware);
    // Create a server with a co-based request handler
    const server = http.createServer((req, res) => {
      co(function*() {
        const context = {};
        // Execute all the middleware using the empty context object
        yield composedMiddleware.call(context);
        // Once the middleware is done, send the request body
        res.end(context.body);
      });
    });
    return server.listen(port);
  };

  return {
    // The `use()` function just adds a new generator function to
    // the list of middleware
    use: (middlewareFn) => middleware.push(middlewareFn),
    listen: listen
  };
};

// Create a new koa app
const app = koa();
// Each app has its own sequence of middleware. The `.use()`
// function adds a generator function to the middleware chain.
app.use(function*(next) {
  // `superagent.get('http://localhost:3000');` will now
  // return 'Hello, World!'
  this.body = 'Hello, World!';
});
const server = app.listen(3000);
```

The real implementation of koa includes many more features, but the fundamental idea is similar to the previous implementation. In particular, the request handler koa 1.x passes to the Node.js `http.createServer()` function is just the result of running koa-compose on the app's middleware plus post-processing to translate the `context` object into an HTTP response.

For instance, the real implementation of koa exposes the request and response objects on the request context. In the minimal koa implementation, your middleware doesn't know anything about incoming requests, which makes it impossible to craft a real response. Supporting this feature would look like what you see below.

```
// Create a server with a co-based request handler
const server = http.createServer((req, res) => {
  co(function*() {
    // Expose req and res by default
    const context = { req: req, res: res };
    // Execute all the middleware using the empty context object
    yield composedMiddleware.call(context);
    // Once the middleware is done, send the request body
    res.end(context.body);
  });
});
```

Koa's middleware approach has some neat features. For instance, it is difficult to define a catch-all error handler in web frameworks like Express. However, in koa, defining an error handler is a trivial application of try/catch.

```
const koa = require('koa');
const superagent = require('superagent');
const app = koa();

// The first middleware is an error handler: if any subsequent
// middleware throws an error, this try/catch will handle it.
app.use(function*(next) {
  try {
    yield next;
  } catch (error) {
    this.body = error.toString();
    this.status = 500;
  }
});

// So if google.com is down, this middleware will throw an error,
// and the above middleware will report it as an HTTP 500.
app.use(function*(next) {
  this.body = (yield superagent.get('http://www.google.com')).text;
  yield next;
});

const server = app.listen(3000);
```

Limitations of koa-compose and Koa

Koa and koa-compose are powerful tools, but they are far from perfect. At the time of this writing, koa 2 is in alpha release and completely breaks the koa 1 API that you learned about in the koa section. Koa 1 also supports the composition module as a replacement for koa-compose. What's the issue with koa-compose? One issue with koa-compose is that it doesn't properly support returning values from middleware. This is a minor issue, which is why koa v2 drops support for composition and uses koa-compose.

```
const compose = require('koa-compose');

const middleware1 = function*(next) {
  const res = yield next;
  return res + 'World!';
};

const middleware2 = function*(next) {
  return 'Hello, ';
};

co(function*() {
  const res = yield compose([middleware1, middleware2])();
  // res is undefined!
});
```

Koa 1.x has several limitations. In this section, you'll learn about 2 common complaints about koa 1.x that are resolved in koa 2.x.

The first limitation is that the entire API for structuring your HTTP response is based on generators. As a matter of fact, if you pass anything other than a generator function to `app.use()`, koa will throw an error.

```
const koa = require('koa');
const app = koa();
// Throws 'AssertionError: app.use() requires a generator function'
app.use(() => { this.body = 'Hello, World' });
// Throws 'AssertionError: app.use() requires a generator function'
app.use(function() { this.body = 'Hello, World' });
```

Generators are a powerful feature, but not right for all use cases. For instance, the upcoming ES2016 JavaScript language standard will support the `async` and `await` keywords, which will enable you to write asynchronous code without callbacks in the same way that `co` and `yield` do. Also, since ES5 doesn't support generators, running koa in ES5 environments is difficult.

The way that koa 2.x works around this limitation is similar to the approach you used to overcome the limitations of your minimal v1 implementation of co: just use promises for everything.

In koa 2.x, the `next` parameter to your middleware is not a generator function: it's a plain-old function that returns a promise.

```
app.use(function(ctx, next) {
  next().then(function(res) {
    // Executed after the rest of the middleware is done
    ctx.body = 'Hello, World!';
  });
});
```

In addition to co, promises work with the ES2016 `async` and `await` keywords and are usable in ES5.

The above example also hints at the second limitation of koa 1.x: relying on the `this` keyword. The `this` keyword is notorious for making JavaScript beginners' lives difficult. Instead of using `this`, koa 2.x passes a 'context' parameter (commonly abbreviated `ctx`) as the first parameter to your middleware function. The koa 2.x `ctx` parameter is analogous to the koa 1.x `this` keyword.

```
const app = koa();
// In koa 2.x, you can use arrow functions as middleware
app.use((ctx) => {
  ctx.body = 'Hello, World!';
});
const server = app.listen(3000);
```

Chapter 4: Transpiling

The last step on your journey to mastering generators is to learn how to transpile generators into ES5. A **transpiler** compiles one JavaScript dialect into another JavaScript dialect. In this chapter, you'll write a rudimentary transpiler that converts generator functions into plain-old JavaScript functions.

Parsing JavaScript is complex, however, learning how generators work in terms of ES5 JavaScript will test the limits of your knowledge of generator fundamentals. In order to get some insight into how to compile generators into ES5, you'll first learn about regenerator, Facebook's open-source transpiler for transforming generator functions into ES5.

```
const regenerator = require('regenerator');

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

// Given the above simple generator function, regenerator will produce
// the below code.
assert.equal(code, `
  var generatorFunction = regeneratorRuntime.mark(function callee$0$0() {
    return regeneratorRuntime.wrap(function callee$0$0$(context$1$0) {
      while (1) switch (context$1$0.prev = context$1$0.next) {
      case 0:
        context$1$0.next = 2;
        return 'Hello, World!';
      case 2:
      case "end":
        return context$1$0.stop();
      }
    }, callee$0$0, this);
  });`);
```

Introducing Regenerator

Regenerator is a transpiler: it takes some JavaScript code as a string, and produces some equivalent JavaScript code as a string. When writing a transpiler, the two key questions are:

1. What code do you want to transform?
2. What do you want to transform the code into?

In regenerator's case, the first question is simple: you want to transform every generator function `function*() {}` and every `yield` statement within a generator function. The second question is more subtle.

The ES2015 spec defines a generator solely in terms of which properties it has. In other words, any JavaScript object can be a generator, not just the return value of a generator function. When it comes to JavaScript language APIs, the letter of the law is vastly more important than the spirit of the law. Any JavaScript object with a `next()` function and a `throw()` function is a generator as far as co is concerned. For example, if you have an object with a `next()` function that returns a promise like you see below, co will still recognize it as a generator. Note that there are no `function*()` definitions.

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

// `plainFunction` is **not** a generator function because
// it isn't declared with `function*() {}`. However, it returns
// an object that qualifies as a generator.
var plainFunction = function() {
  return {
    // `next()` and `throw()` are the only properties necessary for
    // an object to qualify as a generator.
    next: () => {
      return {
        // Note that this generator's `next()` returns a promise
        value: superagent.get('http://www.google.com'),
        done: true
      };
    },
    // `throw()` doesn't get used in this example
    throw: (error) => {
      throw error;
    }
  };
};

co(function*() {
  // You can use the fake generator returned by `plainFunction`
  // with co, since it "yields" a promise.
  const res = yield plainFunction();
  // res.text now contains Google's home page HTML!
});
```

The general idea of regenerator is simple: convert a generator function into a regular function that returns an object that fulfills the ES2015 generator API. Of course, the fake generator needs to return the correct values for `next()`: every `yield` statement needs to cause a `next()` function call to return. The above example is not sophisticated, it only allows you to return a single value. You'll learn about how you can replace yield statements with ES5 code in the "Faking a Generator Function" section.

Below is an example of using regenerator output with co. Note that, even though it isn't defined using `function*`, co still accepts the `generatorFunction` as a valid generator function. Like the fake version of co you saw in the "Asynchronous Coroutines" chapter, co defines a generator function as any object whose `constructor.name` property is equal to 'Generator Function'. Regenerator is smart enough to change this property for you.

```
const co = require('co');
const regenerator = require('regenerator');
const superagent = require('superagent');
// Necessary to include the `regeneratorRuntime` variable
// that you use in the below generated code.
regenerator.runtime();

// The below code is regenerator's output when it transpiles
//
// const generatorFunction = function*() {
//   return superagent.get('http://www.google.com');
// };
var generatorFunction = regeneratorRuntime.mark(function callee$0$0() {
  return regeneratorRuntime.wrap(function callee$0$0$(context$1$0) {
    while (1) switch (context$1$0.prev = context$1$0.next) {
      case 0:
        return context$1$0.abrupt("return",
          superagent.get('http://www.google.com'));
      case 1:
      case "end":
        return context$1$0.stop();
    }
  }, callee$0$0, this);
});

// 'GeneratorFunction'
generatorFunction.constructor.name;

co(generatorFunction).then((res) => {
  // res.text contains google's home page!
});
```

Faking a Generator Function

The first key idea for writing generator functions in ES5 is that a generator can be thought of as a series of functions calls that return values. The difference between `yield` and `return` is that you can't resume a function after `return` has been called. Thus, in order to build a generator function out of normal functions, you need multiple function calls.

Regenerator handles this by creating a function that gets a parameter which defines which "step" the generator is on. A step ends with a `return` statement, or a `yield` statement, which is transformed into a `return`. You can think of a generator as being on the x-th step if there have been x yield statements thus far.

Let's take a look at an example. Suppose you have the below generator function.

```
const generatorFunction = function*() {
  return (yield superagent.get('http://www.google.com')).text;
};
```

The above generator function has 2 steps:

1. `superagent.get('http://www.google.com')`
2. `return` the `text` property from the value that `generator.next()` gives you.

In the below function, the `generatorLogic()` function executes these two steps.

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

// Behaves like the below generator function:
// const generatorFunction = function*() {
//   return (yield superagent.get('http://www.google.com')).text;
// };
const fakeGeneratorFunction = function() {
  let step = 0;
  const numSteps = 2;
  // This function takes the value that was passed to `next()` and
  // the 'step' that the generator is on.
  const generatorLogic = function(v, step) {
    if (step === 0) return superagent.get('http://www.google.com');
    if (step === 1) return v.text;
  }

  return {
    next: (v) => {
      // For the first n-1 functions, return `done: false`
      if (step < numSteps - 1) {
        return {
          done: false,
          value: generatorLogic(v, step++)
        };
      }
      // For last function, return `done: true`
      // (like a `return` in a generator)
      return {
        done: true,
        value: generatorLogic(v, step++)
      };
    },
    throw: (error) => {
      throw error;
    }
  }
};
```

There are two key points the previous example glossed over. The first point is what happens with variable assignments? Suppose you had a generator function that assigned to a variable.

```
const generatorFunction = function*() {
  let res = yield superagent.get('http://www.google.com');
  return res.text;
};
```

The `generatorLogic()` function needs to get called multiple times, and after each `return` the function stack gets wiped out. There's no way to persist local variables between calls to `generatorLogic()`. Right about now you're probably starting to miss generators. The workaround is to store variables in an object outside of the `generatorLogic()` function.

```
const variables = {};

const fakeGeneratorFunction = function(v, step) {
  if (step === 0) return superagent.get('http://www.google.com');
  if (step === 1) {
    variables['res'] = v;
    return variables['res'].text;
  }
};
```

The second and more tricky point is how to handle errors and try/catch. Remember that generators have a `throw()` method that lets you trigger an error in the generator function that you can try/catch. For instance:

```
const generatorFunction = function*() {
  let res;
  try {
    // You'll get an error here (so the `catch` block will execute)
    // because co will call `throw()` when this `superagent.get()`
    // call fails.
    res = yield superagent.get('http://notvalid.baddomain');
  } catch(err) {
    // When the error is thrown, this will return
    return 'Failed';
  }
  return res.text;
};
```

The general idea is that your `generatorLogic()` function needs to take an `error` parameter, and you need to have a separate case for if you're on step 1 and `generator.throw()` was called. In other words, the catch block above needs to be in a separate `if()` statement.

Below is an example of how you can handle errors in your fake generator function.

```
// Behaves like the below generator function:
// const generatorFunction = function*() {
//   let res;
//   try {
//     res = yield superagent.get('http://notvalid.baddomain');
//   } catch(err) {
//     return 'Failed';
//   }
//   return res.text;
// };
const fakeGeneratorFunction = function() {
  let step = 0;
  let variables = {};
  const result = (value, done) => {
    return {
      value: value,
      done: done
    };
  };
  const url = 'http://nota.baddomain';

  const generatorLogic = function(v, step, error) {
    if (step === 0) return result(superagent.get(url), false);
    // This is the catch block
    if (step === 1 && error) return result('Failed', true);
    if (step === 1) {
      variables['res'] = v;
      return result(variables['res'].text, true);
    }
    if (error) throw error;
  }

  return {
    next: (v) => generatorLogic(v, step++),
    // `throw()` needs to set the error parameter to `next()`
    throw: (error) => generatorLogic(null, step++, error)
  }
};
```

In this section, your transpiler will **not** account for try/catch blocks and variable assignments. Implementing these details would make the transpiler too complex to serve as an digestible example. However, the previous examples sketch the general idea of how you would implement variable assignments and try/catch blocks.

There's one more detail to account for: defining an API for the generator runtime that your transpiled generator functions will use. Your transpiler will convert generator function code like what you see below:

```
const generatorFunction = function*() {
  yield superagent.get('http://www.google.com');
};
```

The converted ES5 code will look like what you see below.

```
const generatorFunction = GeneratorFunction(function(v, step) {
  if (step === 0) return generatorResult(
    superagent.get('http://www.google.com'), false);
  return generatorResult(undefined, true);
});
```

The above code uses 2 functions, `GeneratorFunction()` and `generatorResult()`. These functions are the API for your generator runtime. Don't worry, they're only cosmetically different from the logic in `fakeGeneratorFunction()` in previous examples.

```
// We'll use 2 functions to convert a regular function into a
// generator. `generatorResult()` returns an object with the same
// format that `generator.next()` does.
const generatorResult = (value, done) => {
  return {
    value: value,
    done: done
  };
};

// `GeneratorFunction()` converts a fake generator function that
// takes the `v` and `step` parameters into something you can pass
// to `co()`
const GeneratorFunction = function(fakeGeneratorFunction) {
  let res = () => {
    let step = 0;

    return {
      next: (v) => fakeGeneratorFunction(v, step++),
      throw: (error) => {
        throw error;
      }
    }
  };
  // Make the result look like a generator function
  Object.defineProperty(res.constructor, 'name', {
    value: 'GeneratorFunction'
  });

  // Note that this example returns an actual faked generator
  // function rather than a fake generator like previous examples.
  return res;
};

// Here's an example of passing a basic fake generator function
// to `GeneratorFunction()`
co(GeneratorFunction(function(v, step) {
  if (step === 0) return generatorResult(
    superagent.get('http://www.google.com'), false);
  return generatorResult(undefined, true);
}))
```

Parsing Generators With Esprima

In order to write a transpiler, you first need to learn to use a JavaScript parser. Esprima is one of the most well-adopted JavaScript parsers. In this section, you'll inspect the syntax trees esprima produces for generator functions in preparation for writing your own rudimentary transpiler.

Esprima exposes a `parse()` function that takes in some JavaScript code and outputs a **syntax tree**. A syntax tree is a tree that represents the structure of the code - a syntax tree makes it much easier to transform code than if you just tried to manipulate a string.

Below is an example syntax tree for a simple generator function. In particular, note that the generator function expression is parsed onto a node that has a `type` property equal to "FunctionExpression" and a `generator` property that's set to true. It also has a `body` property that contains a `body` array property which contains a `return` statement.

```
const esprima = require('esprima');

const parsed = esprima.parse(`
  const generatorFunction = function*() {
    return 'Hello, World';
  };`).body;

/* `parsed` is an array that looks like what you see below
[
  {
    "type": "VariableDeclaration",
    "declarations": [
      {
        "type": "VariableDeclarator",
        "id": { "type": "Identifier", "name": "generatorFunction" },
        "init": {
          "type": "FunctionExpression",
          "params": [],
          "body": {
            "type": "BlockStatement",
            "body": [
              {
                "type": "ReturnStatement",
                "argument": {
                  "type": "Literal",
                  "value": "Hello, World",
                  "raw": "'Hello, World'"
                }
              }
            ]
          },
          "generator": true
        }
      }
    ],
    "kind": "const"
  }
] */
```

Syntax trees can be confusing at first, but the key idea in this example is what generator functions look like in the syntax tree.

Let's take a look at a simple problem: count the number of generator functions in a piece of code. You'd need to visit each node in the tree and check if it has a `type` property equal to 'FunctionExpression' and a `generator` property equal to 'true'. The hard part is how to visit each node in the tree. To make that easier, you'll use the estraverse module. This module lets you execute two functions for each node in the tree, an `enter()` function that executes before the traversal visits any child nodes, and a `leave()` function that executes after the traversal visits all child nodes.

```
const esprima = require('esprima');
const estraverse = require('estraverse');

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

let numGenerators = 0;
estraverse.traverse(parsed, {
  enter: (node, parent) => {
    if (node.type === 'FunctionExpression' && node.generator) {
      ++numGenerators;
    }
  },
  leave: () => {}
});
assert.equal(numGenerators, 2);
```

Let's take a look at a more challenging problem that will be more useful for your rudimentary transpiler: count the number of `yield` statements in each generator function. For instance, in the previous example, you had 2 generator functions, each with 1 yield statement.

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

For this example, the correct output would be `[1, 1]`, because both the first and the second generator functions have 1 yield statement. What about a trickier case?

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

The correct output is `[2, 1]` because the first generator function has 2 yield statements. However, estraverse will visit the 2nd yield statement after the 2nd generator function.

To properly handle the case where you `yield` a generator and then `yield` another value, you're going to use a stack and the estraverse `leave()` function as shown below.

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

let res = [];
let stack = [];
estraverse.traverse(parsed, {
  enter: (node, parent) => {
    if (node.type === 'FunctionExpression' && node.generator) {
      // We've found a new generator function, so add a 0 to the
      // result array and push the index of this generator function's
      // count in the result array onto the stack
      stack.push(res.length);
      res.push(0);
    } else if (node.type === 'YieldExpression') {
      // We've found a yield statement! Increment the current
      // generator function's count.
      ++res[stack[stack.length - 1]];
    }
  },
  leave: (node, parent) => {
    if (node.type === 'FunctionExpression' && node.generator) {
      // We've visited everything within a generator function, so
      // pop its index off the stack
      stack.pop();
    }
  }
});
assert.deepEqual(res, [2, 1]);
```

Write Your Own Transpiler

Now that you've seen how the syntax tree for generator functions looks, it's time to implement a rudimentary transpiler for generator functions using your runtime API from the "Faking a Generator Function" section. Your transpiler will be far from a fully fledged regenerator replacement, but it will be able to transpile some basic generator functions.

The transpiler needs to perform 3 distinct tasks:

- Convert generator functions into calls to `GeneratorFunction()`.

```
// Before
const f = function*() {};
```

```
// After
const f = GeneratorFunction(function(v, step)) {}
```

- Convert `yield` and `return` expressions to returns that use `generatorResult()`.

```
// Before
yield 'Hello';
return 'World';
```

```
// After
return generatorResult('Hello', false);
return generatorResult('World', true);
```

- Break the function body up into steps based on `yield` and `return` statements.

```
// Before
const variables = {};
const generatorFunction = function*() {
  variables['res'] = yield superagent.get('http://www.google.com');
  return variables['res'].text;
};
```

```
// After
const generatorFunction = GeneratorFunction(function(v, step) {
  if (step === 0) {
    return generatorResult(
      superagent.get('http://www.google.com'), false);
  }
  if (step === 1) {
    // Note that we need to assign to the value passed to `next()`
    variables['res'] = v;
    return generatorResult(variables['res'], true);
  }
});
```

You're going to implement all these steps separately, because one block of code that does all of these steps at once is too complex. The first step, converting `function*` into a call to `GeneratorFunction()`, is the most straightforward. You need to take each 'FunctionExpression' node that has a `generator` flag set and convert it into a non-generator within a 'CallExpression' node as shown below.

```
const esprima = require('esprima');
const estraverse = require('estraverse');
// escodegen exposes a `generate()` function that takes
// an esprima syntax tree and outputs code as a string.
const escodegen = require('escodegen');

const parsed = esprima.parse(`
  const variables = [];
  const generatorFunction = function*() {
    variables['res'] = yield superagent.get('http://www.google.com');
    return variables['res'];
  };`);

estraverse.replace(parsed, {
  enter: (node, parent) => {
    if (node.type === 'FunctionExpression' && node.generator) {
      node.generator = false;
      node.params.push({
        type: 'Identifier',
        name: 'v'
      });
      node.params.push({
        type: 'Identifier',
        name: 'step'
      });
      // This property will identify this node as a former
      // generator function
      node._steps = [[]];

      return {
        type: 'CallExpression',
        callee: {
          type: 'Identifier',
          name: 'GeneratorFunction'
        },
        arguments: [node]
      }
    }
  }
});

assert.equal(escodegen.generate(parsed), `
const variables = [];
const generatorFunction = GeneratorFunction(function (v, step) {
    variables['res'] = yield superagent.get('http://www.google.com');
    return variables['res'];
});`.trim());
```

The second step for the transpiler is to convert `yield` and `return` statements to `return` statements that use the `generatorResult()` function. This ensures the resulting generator's `next()` returns properly formated results. To do this, every 'YieldExpression' and 'ReturnStatement' node needs to be transformed to a 'ReturnStatement' that returns a call to `generatorResult()`, as shown below.

```
const parsed = esprima.parse(`
  const variables = [];
  const generatorFunction = function*() {
    variables['res'] = yield superagent.get('http://www.google.com');
    return variables['res'];
  };`);
const FunctionCall = (name, args) => {
  return {
    type: 'CallExpression',
    callee: {
      type: 'Identifier',
      name: name
    },
    arguments: args
  };
};
const Literal = (value) => {
  return {
    type: 'Literal',
    value: value,
    raw: value.toString()
  };
};

estraverse.replace(parsed, {
  enter: (node, parent) => {},
  leave: (node, parent) => {
    const type = node.type;
    if (type === 'YieldExpression' || type === 'ReturnStatement') {
      if (type === 'ReturnStatement') {
        const args = [node.argument, Literal(true)];
        node.argument = FunctionCall('generatorResult', args);
      } else if (type === 'YieldExpression') {
        node.type = 'ReturnStatement';
        node._wasYield = true;
        const args = [node.argument, Literal(false)];
        node.argument = FunctionCall('generatorResult', args);
      }
    }
  }
});

const yieldStatement = `superagent.get('http://www.google.com')`;
assert.equal(escodegen.generate(parsed), `
const variables = [];
const generatorFunction = GeneratorFunction(function (v, step) {
    variables['res'] = return generatorResult(${yieldStatement}, false);;
    return generatorResult(variables['res'], true);
});`.trim());
```

The third and final step is the most complex. This step has 2 primary objectives. First, you need to break up the code into steps: all the code between `return` statements needs to be wrapped in an `if` statement that checks the current `step`. To do this, you'll add a 2d array `_steps` to every generator function node. You'll add each top-level expression to the last element of the `_steps` array, and add a new empty array to the `_steps` array every time you see a return statement.

Secondly, you need to transform any assignments that have a `yield` statement as the right-hand side. To do this, you'll add a new 'AssignmentExpression' to the start of the next step that assigns to the parameter `v`, which is the value passed to `generator.next()`.

```
const clone = v => JSON.parse(JSON.stringify(v));
const Identifier = name => ({ type: 'Identifier', name: name });
const BlockStatement = body => ({ type: 'BlockStatement', body: body });
const ReturnStatement = v => ({ type: 'ReturnStatement', argument: v });
const If = (t, c) => ({ type: 'IfStatement', test: t, consequent: c });
const Test = (l, r) => ({ type: 'BinaryExpression', operator: '===',
  left: l, right: r });
let stack = [];
let lastGen = () => stack.length && stack[stack.length - 1];
estraverse.replace(parsed, {
  enter: (node, parent) => {
    // Keep track of nested generator functions
    if (node._steps) stack.push(node);
    // Put top-level expressions into the `_steps` array
    else if (lastGen() && lastGen().body === parent)
      lastGen()._steps[lastGen()._steps.length - 1].push(node);
  },
  leave: (node, parent) => {
    if (node.type === 'ExpressionStatement' &&
        node.expression.type === 'AssignmentExpression' &&
        node.expression.right._wasYield) {
      // Handle assigning to result of `yield`
      const newNode = clone(node);
      newNode.expression.right = Identifier('v');
      lastGen()._steps[lastGen()._steps.length - 1].push(newNode);
      node.type = 'ReturnStatement';
      node.argument = node.expression.right.argument;
    } else if (node.type === 'FunctionExpression' && node._steps) {
      // Merge all of the steps into if statements
      const newBody = [];
      for (let i = 0; i < node._steps.length - 1; ++i) {
        // Wrap each step in an if statement
        const test = Test(Identifier('step'), Literal(i));
        newBody.push(If(test, BlockStatement(clone(node._steps[i]))));
      }
      const r = [Identifier(undefined), Literal(true)];
      newBody.push(ReturnStatement(FunctionCall('generatorResult', r)));
      node.body.body = newBody;
    } else if (node.type === 'ReturnStatement') {
      // If there's a return, create a new step
      lastGen()._steps.push([]);
    } else if (node._steps) stack.pop();
  }
});
```