THE MARVELLOUSLY MYSTERIOUS JAVASCRIPT MAYBE MONAD

Written by James Sinclair on the 7th August 2016

You finally made it. You stepped through the looking glass. You learned functional programming. You mastered currying and composition, and followed the path of functional purity. And gradually, you notice a change in the attitude of the other programmers. There's ever-so-slightly less disdain in their voice when you talk to them. Every so often you'll get a little nod when you happen to mention immutable data structures. You've begun to earn their respect. And yet…

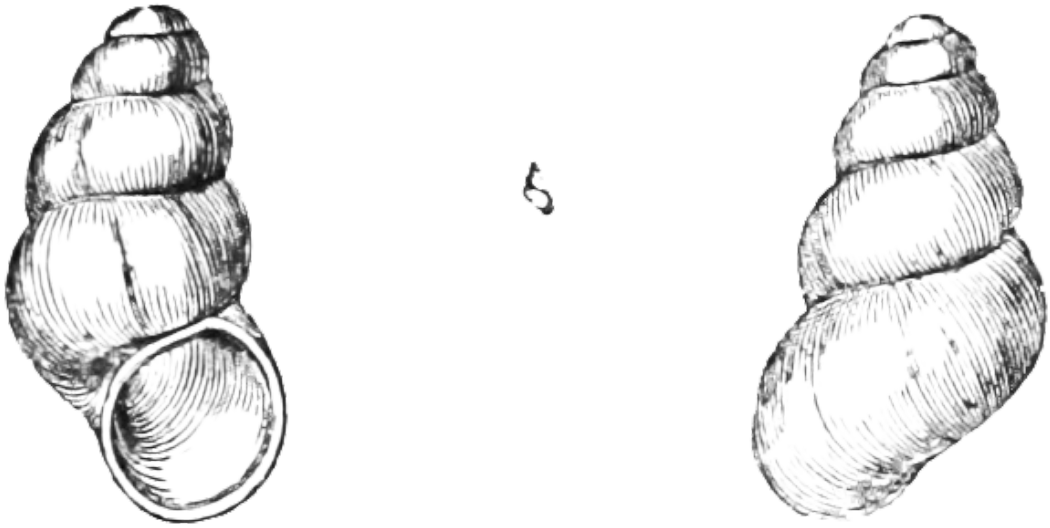
There's something they won't talk about. When they think you're not in earshot, every so often, you'll overhear the word 'monad' discussed in hushed tones. But as soon as they notice you're there, they change the subject. One day, you pluck up the courage to ask someone. "What's this monad thing I keep hearing about?" The other programmer just looks at you. After an awkward silence she simply says "I can't talk about it". So you ask another programmer and she replies "Maybe when you've learned Haskell." She walks away sadly, shaking her head.

Mystified, you start searching for answers on the Internet. And at first there seems to be plenty of people eager to explain the mysterious monads. But, there's a problem. It's as if every single one of them writes in some kind of code. They talk about applicative functors, category theory, algebraic structures and monadic laws. But none of them seem to explain what monads are for. What do they do? Why do they exist? You keep searching and discover article after article trying to come up with some kind of analogy. Monads are like tupperware. Monads are like trees. Monads are like a bucket line. Monads are like hazmat suits. Monads are like burritos. Comparing monads to burritos considered harmful… It starts to drive you mad.

One day, one of the more junior programmers approaches you, a furtive expression on his face. "Look, you've got to stop asking questions about monads, OK? It upsets people. Monads are cursed. It's not that people don't want to tell you about them. They can't." He looks around again and continues in a hushed tone. "Even ol' father Crockford couldn't break the curse. He tried. In a keynote conference talk and everything. But it got him. He couldn't do it. Either you figure monads out or you don't. No one can help you. That's just how it works."

Monads have a bad reputation in the JavaScript community. Douglas Crockford once said that monads are cursed. Once you finally understand monads, you lose the ability to explain monads to others. [1] Even experienced functional programmers treat monads with respect. And some of the explanations out there *are* hard to understand. Especially if they dive straight into category theory. But, if you can understand Promises then you can understand monads.

In this article we will look at just one type of monad: The Maybe monad. Focussing on just one will help explain the basic idea without getting too bogged down in theory. Hopefully it will be enough to set you on the path to enlightenment. I'm still new to Monads myself. Perhaps new enough that the curse hasn't fully taken hold. Let's see how it goes…



This article assumes you've some familiarity with *functional programming in JavaScript*. And also that you have some experience working with *JavaScript Promises*.

A QUICK RECAP OF PROMISES

```
Promise.resolve($.getJSON('/path/to/my/api'))
    .then(function(data) {
        // Do something with the data in here.
    });
```

`Promise.resolve()` was necessary because jQuery's version of Promises didn't fully meet the *Promises/A+ standard*. So, clever people would use the `.resolve()` method to make the jQuery version into a real Promise.

Now, if I wanted to, I could rewrite the code above so that it uses a named function instead of an

```
function doSomething(data) {
    // Do something with the data in here.
}

Promise.resolve($.getJSON('/path/to/my/api'))
    .then(doSomething);
```

Same code, just in a different order.

Now, one of the features that makes promises so popular is that you can chain them together. So if

```
Promise.resolve($.getJSON('/path/to/my/api'))
    .then(doSomething)
    .then(doSomethingElse)
    .then(doAnotherThing);
```

```
var p1 = Promise.resolve($.getJSON('/path/to/my/api'));
var p2 = p1.then(doSomething);
var p3 = p2.then(doSomethingElse);
var p4 = p3.then(doAnotherThing);
```

Here we are creating four promises. Each one represents a future value. The intermediate variables aren't necessary, but they make things clearer. Each `.then()` call is returning a new promise object. The key thing is that the functions themselves don't have to know that they're inside a Promise. They just expect regular values as parameters. This is good because it keeps the functions simple and easy to understand.

Now, if you've worked with Promises before, then you may know that `Promise.resolve()` can work with plain values too, not just AJAX calls. So, returning to the example above, we could

```
var data = {foo: 'bar'};
Promise.resolve(data)
    .then(doSomething)
    .then(doSomethingElse)
    .then(doAnotherThing);
```

This creates a promise that resolves straight away with the value of `data`. What's interesting to note here is that for this code to work with a regular value instead of an asynchronous value, *we didn't change a thing*. All the named functions still take regular variables and return whatever they return.

Monads are like Promises in that they allow us to handle tricky things with a consistent approach. 'Tricky things' might include asynchronous data, or null values, or something else entirely. The monad hides away a bunch of the complexity so we don't have to think about it. This lets us concentrate on writing simple, pure functions that are easy to understand.

A PROBLEM TO SOLVE

To show how a monad might be useful, let's consider an example problem. Let's say we're working on some code to personalise a website. We want to change the main banner of the site depending on

```
var user = {
    email: 'james@example.com',
    accountDetails: {
        address: {
            street:   '123 Fake St',
            city:     'Exampleville',
            province: 'NS',
            postcode: '1234'
        }
    },
    preferences: {}
}
```

```
var banners = {
    'AB': '/assets/banners/alberta.jpg',
    'BC': '/assets/banners/british-columbia.jpg',
    'MB': '/assets/banners/manitoba.jpg',
    'NL': '/assets/banners/newfoundland-labrador.jpg',
    'NS': '/assets/banners/nova-scotia.jpg',
    'NT': '/assets/banners/northwest-territories.jpg',
    'ON': '/assets/banners/ontario.jpg',
    'PE': '/assets/banners/prince-edward.jpg',
    'QC': '/assets/banners/quebec.jpg',
    'SK': '/assets/banners/saskatchewan.jpg',
    'YT': '/assets/banners/yukon.jpg',
};
```

So, for the 'ordinary' case, we can write a nice simple function to grab the right banner:

```
function getUserBanner(banners, user) {
    return banners[user.accountDetails.address.province];
}
```

One line. Simple. Easy. Done.

And because we're badass functional programmers, we could even write this

```
var R       = require('ramada'),
    compose = R.compose,
    prop    = R.prop,
    path    = R.path;

var getUserBanner = compose(
    prop(R.__, banners),
    path(['accountDetails', 'address', 'province'])
);
```

Except...

```
var user = {};
```

So, to handle that case, we abandon pointfree style, and add a check to see if

```
function getUserBanner(banners, user) {
    if (typeof user.accountDetails !== 'undefined') {
        return banners[user.accountDetails.address.province];
    }
}
```

```
var user = null;
```

```
function getUserBanner(banners, user) {
    if (user !== null) {
        if (user.accountDetails !== undefined) {
            return banners[user.accountDetails.address.province];
        }
    }
}
```

But there's also the case where the user has signed in, but has never filled out their address details. In

```
var user = {
    email:          'james@example.com',
    accountDetails: {}
};
```

```
function getUserBanner(banners, user) {
    if (user !== null) {
        if (user.accountDetails !== undefined) {
            if (user.accountDetails.address !== undefined) {
                return banners[user.accountDetails.address.province];
            }
```

```
        }
    }
}
```

This is starting to look like a pyramid of doom. To make it slightly better, could merge it all into one

```
function getUserBanner(banners, user) {
    if ((user !== null)
        && (user.accountDetails !== undefined)
        && (user.accountDetails.address !== undefined)) {
        return banners[user.accountDetails.address.province];
    }
}
```

But this isn't a great improvement on the pyramid of doom. What was an easy one-line function has transformed into a messy bunch of conditionals. It's hard to read and makes the purpose of the function less clear. Fortunately, the Maybe monad can help us.

THE MAYBE MONAD

In essence, a monad is simply a wrapper around a value. We can create that with an object that holds

```
var Maybe = function(val) {
    this.__value = val;
};

var maybeOne = new Maybe(1);
```

Typing that `new` keyword everywhere is a pain though (and has other problems). It would be nice to

```
Maybe.of = function(val) {
    return new Maybe(val);
};

var maybeOne = Maybe.of(1);
```

Because the point of our Maybe monad is to protect us from empty values (like `null` and `undefined`),

```
Maybe.prototype.isNothing = function() {
    return (this.__value === null || this.__value === undefined);
};
```

So far, our Maybe wrapper doesn't do anything for us. If anything, it makes life harder. We want to be able to do things with the value. So, we write a method that will let us get the value and do something with it. But we'll also put a guard on it, to protect us from those pesky `null` and `undefined`values.

```
Maybe.prototype.map = function(f) {
    if (this.isNothing()) {
        return Maybe.of(null);
    }
    return Maybe.of(f(this.__value));
};
```

This is already enough to be useful. We can rewrite our `getUserBanner()` function so that it uses a

```
function getUserBanner(banners, user) {
    return Maybe.of(user)
        .map(prop('accountDetails'))
        .map(prop('address'))
        .map(prop('province'))
        .map(prop(R.__, banners));
}
```

If any of those `prop` calls returns `undefined` then Maybe just skips over it. We don't have to catch or throw any errors. Maybe just quietly takes care of it.

This looks a lot like our Promise pattern. We have something that creates the monad, `Maybe.of()`, rather like `Promise.resolve()`. And then we have a chain of `.map()` methods that do something with the value, rather like `.then()`. A Promise lets us write code without worrying about whether data is asynchronous or not. The Maybe monad lets us write code without worrying whether data is empty or not.

MAYBE OF A MAYBE? MAYBE NOT.

Now, what if we got excited about this whole Maybe thing, and decided to write a function to grab the

```
var getProvinceBanner = function(province) {
    return Maybe.of(banners[province]);
};
```

```
function getUserBanner(user) {
    return Maybe.of(user)
        .map(prop('accountDetails'))
        .map(prop('address'))
        .map(prop('province'))
        .map(getProvinceBanner);
}
```

But now we have a problem. Instead of returning a Maybe with a string inside it, we get back a Maybe

```
getUserBanner(user)
    .map(function(m) {
        m.map(function(banner) {
            // I now have the banner,
            // but this is too many maps
        }
    })
```

We're back to another pyramid of doom. We need a way of flattening nested Maybes back down —*join* them together, you might say. So we create a `.join()` method that will unwrap an outer Maybe

```
Maybe.prototype.join = function() {
    return this.__value;
};
```

```
function getUserBanner(user) {
    return Maybe.of(user)
        .map(prop('accountDetails'))
```

```
        .map(prop('address'))
        .map(prop('province'))
        .map(getProvinceBanner)
        .join();
}
```

That gets us back to one layer of Maybe. So we can work with functions that pass back Maybes. But, if we're mapping and joining a lot, we might as well combine them into a single method. It allows us

```
Maybe.prototype.chain = function(f) {
    return this.map(f).join();
};
```

```
function getUserBanner(user) {
    return Maybe.of(user)
        .map(R.prop('accountDetails'))
        .map(R.prop('address'))
        .map(R.prop('province'))
        .chain(getProvinceBanner);
}
```

And because Ramda's `path()` handles missing values in a sensible way, we can reduce this down even

```
function getUserBanner(user) {
    return Maybe.of(user)
        .map(path(['accountDetails', 'address', 'province']))
        .chain(getProvinceBanner);
}
```

With `chain()` we now have a way of interacting with functions that return other Maybe monads. Notice that with this code, there's no if-statements in sight. We don't need to check every possible little thing that might be missing. If a value is missing, the next step just isn't executed.

BUT WHAT DO YOU *DO* WITH IT?

You may be thinking, "That's all well and good, but my banner value is still wrapped up inside a Maybe. How do I get it out again?" And that's definitely a legitimate question. But let me ask you another question first: "Do you *need* to get it out?"

Think about it for a moment. When you wrap a value up inside a Promise, you *never* get it out again. The event loop moves on, and you can never come back to the context you started with. [3] Once you wrap the value in the Promise, you never unwrap it. And it's just fine. We work inside callback functions to do what we need to do. It's not a big deal.

Unwrapping a Maybe kind of defeats the purpose of having it at all. Eventually though, you will want to do *something* with your value. And we need to decide what to do if the value is `null` at that point. With our example, we will want to add our banner to the DOM. What if we wanted to have a

```
Maybe.prototype.orElse = function(default) {
    if (this.isNothing()) {
        return Maybe.of(default);
    }

    return this;
};
```

Now, if our visiting user happens to come from Nunavut, we can at least show *something*. And since we've got that sorted, let's aslo grab the banner element from the DOM. We'll wrap it up in a Maybe

```
// Provide a default banner with .orElse()
var bannerSrc = getUserBanner(user)
            .orElse('/assets/banners/default-banner.jpg');

// Grab the banner element and wrap it in a Maybe too.
var bannerEl = Maybe.of(document.querySelector('.banner > img'));
```

Now we have two Maybes: `bannerSrc` and `bannerEl`. We want to use them both together to set the banner image (our original problem). Specifically, we want to set the `src` attribute of the DOM element in `bannerEl` to be the string inside `bannerSrc`. What if we wrote a function that expected two

```
var applyBanner = function(mBanner, mEl) {
    mEl.__value.src = mBanner.__value;
    return mEl;
};

applyBanner(bannerSrc, bannerEl);
```

This would work just fine, until one of our values was `null`. Because we're pulling values out directly, we're not checking to see if the value is empty. It defeats the entire purpose of having things wrapped in a Maybe to start with. With `.map()`, we have a nice interface where our functions don't need to know anything about Maybe. Instead, they just deal with the values they're passed. If only there was some way to use .map() with our two Maybes...

```
var curry = require('ramda').curry;

var applyBanner = curry(function(el, banner) {
    el.src = banner;
    return el;
});
```

```
bannerEl.map(applyBanner);
// => Maybe([function])
```

We get a *function* wrapped in a Maybe. Now, stay with me. This isn't as crazy as it might seem. The basic building block of functional programming is first-class functions. And all that means is that we can pass functions around just like any other variable. So why not stick one inside a Maybe? All we need then is a version of `.map()` that works with a Maybe-wrapped function. In other words, a

```
Maybe.prototype.ap = function(someOtherMaybe) {
    return someOtherMaybe.map(this.__value);
}
```

Remember that in the context above, `this.__value` is a function. So we're using `map` the same way we

```
var mutatedBanner = bannerEl.map(applyBanner).ap(bannerSrc);
```

This works, but isn't super clear. To read this code we have to remember that `applyBanner` takes two parameters. Then also remember that it's partially applied by `bannerEl.map()`. And then it's applied to `bannerSrc`. It would be nicer if we could say "Computer, I've got this function that takes two regular

variables. Transform it into one that works with Maybe monads." And we can do just that with a

```
var liftA2 = curry(function(fn, m1, m2) {
    return m1.map(fn).ap(m2);
});
```

Note that we assume `fn` is curried. We now have a neat function that can take another function and

```
var applyBannerMaybe = liftA2(applyBanner);
var mutatedBanner    = applyBannerMaybe(bannerEl, bannerSrc);
```

Mission accomplished. We're now able to pluck the province value from deep within the user preference object. We can use that to look up a banner value, and then apply it to the DOM, safely, without a single if-statement. We can just keep mapping and chaining without a care in the world. Using Maybe, I don't have to think about all the checks for `null`. The monad takes care of that for me.

POINTFREE STYLE

Now, at this point you may be thinking "Hold on just a second there, Sir. You keep talking about functional programming, but all I see is objects and methods. Where's the function composition?" And that is a valid objection. But we've been writing functional JavaScript all along, just using a

```
// map :: Monad m => (a -> b) -> m a -> m b
var map = curry(function(fn, m) {
    return m.map(fn);
});

// chain :: Monad m => (a -> m b) -> m a -> m b
var chain = curry(function(fn, m) {
    return m.chain(fn);
});
```

```
// ap :: Monad m => m (a -> b) -> m a -> m b
var ap = curry(function(mf, m) { // mf, not fn, because this is a wrapped function
    return mf.ap(m);
});

// orElse :: Monad m => m a -> a -> m a
var orElse = curry(function(val, m) {
    return m.orElse(val);
});
```

```
var pipe        = require('ramda').pipe;
var bannerEl    = Maybe.of(document.querySelector('.banner > img'));
var applyBanner = curry(function(el, banner) { el.src = banner; return el; });

// customiseBanner :: Monad m => String -> m DOMElement
var customiseBanner = pipe(
    Maybe.of,
    map(R.path(['accountDetails', 'address', 'province'])),
    liftA2(applyBanner, bannerEl)
);

customiseBanner(user);
```

There are still two impure functions, but `customiseBanner` is now pointfee. And here's were things start to get interesting...

Note that when we defined the functional forms of `map`, `chain`, `ap` etc. we didn't include any mention of `Maybe`. This means that any object that implements `.map()` can work with the `map`function. Any object that implements `.chain()`can work with `chain`. And so on. Imagine if we had other objects that implemented these methods...

PIPELINES

To show how this works, I'm going to break all the rules for a moment. I'm going to alter the `Promise`prototype. Note that this is being performed by a trained professional under controlled

```
Promise.of              = Promise.resolve;
Promise.prototype.map   = Promise.prototype.then;
Promise.prototype.chain = Promise.prototype.then;
Promise.prototype.ap    = function(otherPromise) {
    return this.then(otherPromise.map);
};
```

```
// Set the innerHTML attribute on an element.
// Note, this method mutates data. Use with caution.
var setHTML = curry(function (el, htmlStr) {
    el.innerHTML = htmlStr;
    return el;
});

// Get an element.
// Note, this is an impure function because it relies on the global document.
// Use with caution.
var getEl = compose(Promise.of, document.querySelector);

// Fetch an update from a server somewhere.
```

```
// Takes a URL and returns a Promise for JSON.
var fetchUpdate = compose(Promise.of, $.getJSON);

// Process an update.
var processUpdate = pipe(
    map(R.path(['notification', 'message'])),
    liftA2(setHTML, getEl('.notifications'))
);

var updatePromise = fetchUpdate('/path/to/update/api');
processUpdate(updatePromise);
```

Take a moment to look at that `processUpdate`function again. We've defined a pipeline that takes a monad input and then map and lift to transform it. But there's nothing in the pipeline that assumes we're working with a Promise. The pipeline would work just as well with our Maybe monad. And, in fact, it would work with any object that meets the *Fantasyland Monad Spec*.

So, let's recap what we've looked at:

- A monad is like a Promise in that you don't act on a value directly. Instead, we use `map` to apply a callback, just like `then` with Promises.
- The Maybe monad will only map if it has a value. So, when we map a Maybe, we don't have to worry about `null` or `undefined` values.
- If we use monad libraries that conform to a specification, we can compose pipelines. These pipelines can work interchangeably with different types of monad.

FURTHER READING

There is a lot more to learn about monads, and there are many other types of monads besides Maybe. I encourage you to keep reading and find out more. There are three main resources I've found helpful:

- *Professor Frisby's Mostly Adequate Guide to Functional Programming* by *Brian Lonsdorf*
- *The Perfect API* by *James Forbes*
- *The Fantasyland Specification* sets out rules that keep monads and other algebraic structures interoperable.
- *A Map to Success: Functors in Javascript* by *Kevin Welcher* (a functor is just an object that implements `map()`, so monads are functors that implement a couple of extra things on top).

Slowly, it begins to make sense. You wouldn't claim to 'understand' monads, but you can see how using Maybe might save a lot of effort. So, you slip it in to your next commit, neatly avoiding a couple of null checks. You don't make a big deal about it. The other programmers don't say anything, but you know that they noticed. There's still a lot to learn, but monads are no longer a complete mystery. They're tools for getting a job done.

1. Monads and Gonads (YUIConf Evening Keynote), *https://www.youtube.com/watch?v=dkZFtimgAcM* ↩
2. This implementation of `map` is based on the `Maybe` monad found in *Professor Frisby's Mostly Adequate Guide to Functional Programming* by *Brian Lonsdorf*. ↩
3. OK. OK. Yes, with async/await and generators you can kind-of do something that looks like that, but you definitely can't with ES5. And yet we still managed to get things done before we had all this syntactic sugar. ↩

4. Note this implementation of `.orElse()` differs from other libraries. This one expects you to return a regular value, while others expect you to wrap the return value in a Maybe yourself. ↩
5. Yes, this function is impure and mutates data, but we'll ignore that for now. ↩