A Beginner's Guide to Currying in Functional JavaScript

M. David Green(https://www.sitepoint.com/author/mdavidgreen/) October 16, 2015 +81

sitepoint
FULL STACK JAVASCRIPT DEVELOPMENT WITH
MEAN
BY COLIN J. IHRIG & ADAM BRETZ
ELEGANT, POWERFUL, TESTABLE, EXTENDABLE

Cu of the functional techniques
tha miliar with more traditional
wa pplied properly, it can actually
ma readable.

More Readable And More Flexible

One of the advantages touted for functional JavaScript is shorter, tighter code that gets right to the point in the fewest lines possible, and with less repetition. Sometimes this can come at the expense of readability; until you're familiar with the way the functional programming works, code written in this way can be harder to read and understand.

If you've come across the term currying before, but never knew what it meant, you can be forgiven for thinking of it as some exotic, spicy technique that you didn't need to bother about. But currying is actually a very simple concept, and it addresses some familiar problems when dealing with function arguments, while opening up a range of flexible options for the developer.

What Is Currying?

Briefly, currying is a way of constructing functions that allows partial application of a function's arguments. What this means is that you can pass all of the arguments a function is expecting and get the result, or pass a subset of those arguments and get a function back that's waiting for the rest of the arguments. It really is that simple.

Currying is elemental in languages such as Haskell and Scala, which are built around functional concepts. JavaScript has functional capabilities, but currying isn't built in by default (at least not in current versions of the language). But we already know some functional tricks, and we can make currying work for us in JavaScript, too.

To give you a sense of how this could work, let's create our first curried function in JavaScript, using familiar syntax to build up the currying functionality that we want. As an example, let's imagine a function that greets somebody by name. We all know how to create a simple greet function that takes a name and a greeting, and logs the greeting with the name to the console:

```
var greet = function(greeting, name) {
  console.log(greeting + ", " + name);
};
greet("Hello", "Heidi"); //"Hello, Heidi"
```

This function requires both the name and the greeting to be passed as arguments in order to work properly. But we could rewrite this function using simple nested currying, so that the basic function only requires a greeting, and it returns another function that takes the name of the person we want to greet.

Our First Curry

```
var greetCurried = function(greeting) {
  return function(name) {
    console.log(greeting + ", " + name);
  };
};
```

This tiny adjustment to the way we wrote the function lets us create a new function for any type of greeting, and pass that new function the name of the person that we want to greet:

```
var greetHello = greetCurried("Hello");
greetHello("Heidi"); //"Hello, Heidi"
greetHello("Eddie"); //"Hello, Eddie"
```

We can also call the original curried function directly, just by passing each of the parameters in a separate set of parentheses, one right after the other:

```
greetCurried("Hi there")("Howard"); //"Hi there, Howard"
```

Why not try this out in your browser?

JS Bin on jsbin.com (http://jsbin.com/basoyi/embed?js,console)

Curry All the Things!

The cool thing is, now that we have learned how to modify our traditional function to use this approach for dealing with arguments, we can do this with as many arguments as we want:

```
var greetDeeplyCurried = function(greeting) {
  return function(separator) {
    return function(emphasis) {
      return function(name) {
        console.log(greeting + separator + name + emphasis);
      };
    };
  };
};
```

We have the same flexibility with four arguments as we have with two. No matter how far the nesting goes, we can create new custom functions to greet as many people as we choose in as many ways as suits our purposes:

```
var greetAwkwardly = greetDeeplyCurried("Hello")("...")("?");
greetAwkwardly("Heidi"); //"Hello...Heidi?"
greetAwkwardly("Eddie"); //"Hello...Eddie?"
```

What's more, we can pass as many parameters as we like when creating custom variations on our original curried function, creating new functions that are able to take the appropriate number of additional parameters, each passed separately in its own set of parentheses:

```
var sayHello = greetDeeplyCurried("Hello")(", ");
sayHello(".")("Heidi"); //"Hello, Heidi."
sayHello(".")("Eddie"); //"Hello, Eddie."
```

And we can define subordinate variations just as easily:

```
var askHello = sayHello("?");
askHello("Heidi"); //"Hello, Heidi?"
askHello("Eddie"); //"Hello, Eddie?"
```

JS Bin on jsbin.com (http://jsbin.com/tetugu/embed?js,console&height=450px)

Currying Traditional Functions

You can see how powerful this approach is, especially if you need to create a lot of very detailed custom functions. The only problem is the syntax. As you build these curried functions up, you need to keep nesting returned functions, and call them with new functions that require multiple sets of parentheses, each containing its own isolated argument. It can get messy.

To address that problem, one approach is to create a quick and dirty currying function that will take the name of an existing function that was written without all the nested returns. A currying function would need to pull out the list of arguments for that function, and use those to return a curried version of the original function:

```
var curryIt = function(uncurried) {
  var parameters = Array.prototype.slice.call(arguments, 1);
  return function() {
    return uncurried.apply(this, parameters.concat(
      Array.prototype.slice.call(arguments, 0)
    ));
  };
};
```

To use this, we pass it the name of a function that takes any number of arguments, along with as many of the arguments as we want to pre-populate. What we get back is a function that's waiting for the remaining arguments:

```
var greeter = function(greeting, separator, emphasis, name) {
  console.log(greeting + separator + name + emphasis);
};
var greetHello = curryIt(greeter, "Hello", ", ", ".");
greetHello("Heidi"); //"Hello, Heidi."
greetHello("Eddie"); //"Hello, Eddie."
```

And just as before, we're not limited in terms of the number of arguments we want to use when building derivative functions from our curried original function:

```
var greetGoodbye = curryIt(greeter, "Goodbye", ", ");
greetGoodbye(".", "Joe"); //"Goodbye, Joe."
```

JS Bin on jsbin.com (http://jsbin.com/hayaka/embed?js,console&height=475px)

Getting Serious about Currying

Our little currying function may not handle all of the edge cases, such as missing or optional parameters, but it does a reasonable job as long as we stay strict about the syntax for passing arguments.

Some functional JavaScript libraries such as Ramda (http://ramdajs.com/) have more flexible currying functions that can break out the parameters required for a function, and allow you to pass them individually or in groups to create custom curried variations. If you want to use currying extensively, this is probably the way to go.

Regardless of how you choose to add currying to your programming, whether you just want to use nested parentheses or you prefer to include a more robust carrying function, coming up with a consistent naming convention for your curried functions will help make your code more readable. Each derived variation of a function should have a name that makes it clear how it behaves, and what arguments it's expecting.

Argument Order

One thing that's important to keep in mind when currying is the order of the arguments. Using the approach we've described, you obviously want the argument that you're most likely to replace from one variation to the next to be the last argument passed to the original function.

Thinking ahead about argument order will make it easier to plan for currying, and apply it to your work. And considering the order of your arguments in terms of least to most likely to change is not a bad habit to get into anyway when designing functions.

Conclusion

Currying is an incredibly useful technique from functional JavaScript. It allows you to generate a library of small, easily configured functions that behave consistently, are quick to use, and that can be understood when reading your code. Adding currying to your coding practice will encourage the use of partially applied functions throughout your code, avoiding a lot of potential repetition, and may help get you into better habits about naming and dealing with function arguments.

If you enjoyed, this post, you might also like some of the others from the series:

- An Introduction to Functional JavaScript (http://www.sitepoint.com/introduction-functional-javascript/)
- Higher-Order Functions in JavaScript (http://www.sitepoint.com/higher-order-functions-javascript/)
- Recursion in Functional JavaScript (http://www.sitepoint.com/recursion-functional-javascript/)

Tags: currying (https://www.sitepoint.com/tag/currying/), functional programming (https://www.sitepoint.com/tag/functional-programming/), functional-js (https://www.sitepoint.com/tag/functional-js/), nested currying (https://www.sitepoint.com/tag/nested-currying/), partial application (https://www.sitepoint.com/tag/partial-application/)

M. David Green (https://www.sitepoint.com/author/mdavidgreen/)

(https://twitter.com/mdavidgreen)

I've worked as a Web Engineer, Writer, Communications Manager, and Marketing Director at companies such as Apple, Salon.com, StumbleUpon, and Moovweb. My research into the Social Science of Telecommunications at UC Berkeley, and while earning MBA in Organizational Behavior, showed me that the human instinct to network is vital enough to thrive in any medium that allows one person to connect to another.

Mike Mx • 8 months ago
Any practical usage examples?
12 • Reply • Share ›

Iven Marquardt Mike Mx • 3 months ago
Currying reveals its benefits only in combination with another, more advanced functional feature: combinatorics. Function composition for example, is based on the compose function which is also known as the k-combinator. There are many others. When you start functional programming, currying is going to become the most natural matter for you.
• Reply • Share ›

Abdulsattar Mohammed Mike Mx • 8 months ago
It goes really well with other functional programming features. For example, if you use compose, you can write `compose(sum,

curryMap(square))([1..100])` if sum and square are functions.
• Reply • Share ›

fheck ➦ Mike Mx • 8 months ago
https://medium.com/@fabrik42/c...

Maybe here :)
• Reply • Share ›

Jacob Groß ➦ Mike Mx • 8 months ago
Same question here. As I've never written Haskell and such I have no idea why I should use this.
• Reply • Share ›

Panayiotis “PVgr” Velisarakos • 8 months ago
Good explanation, and the article is easy to follow, but I still don’t get the why. I mean, what is a real-world problem that can be solved better with currying. Thanks!
2 • Reply • Share ›

josh ➦ Panayiotis “PVgr” Velisarakos • 8 months ago
As I understand it, any time you find yourself passing the same parameter value lots of times could be curried so that isn't happening:

callFunc(123, 'xyz')

callFunc(123, 'abc')

callFunc(123, 'def')

vs

callFuncCurried123('xyz')

callFuncCurried123('abc')

callFuncCurried123('def')
1 • Reply • Share ›

c3p0 ➦ josh • 2 months ago
In this case couldn't you just pass in an array to callFunc() like:

var myArr = [123, 'xyz'];
callFunc(myArr);

Still trying to wrap my head around when / how I should use this. Thanks!
• Reply • Share ›

Panayiotis “PVgr” Velisarakos ➦ josh • 8 months ago
Thank you!
• Reply • Share ›

RentBin.com • 12 days ago
Wow, great explanation!!!
• Reply • Share ›

Anas Raza Firdousi • 2 months ago
If you guys need a more detailed intro to functional programming in JavaScript, feel free to go over this:

http://anasfirdousi.com/introd...
• Reply • Share ›

Tung Dang • 7 months ago
Thanks for useful article.
In the past, I used "_.partial" (http://underscorejs.org/#parti... sometimes. Now I understand its meaning clearly.
• Reply • Share ›

Qbe Root • 7 months ago
No mention of Function.prototype.bind? :-(Aside from locking the this value, it allows partial application of functions, like currying.
https://developer.mozilla.org/...
• Reply • Share ›

mdavidgreen ➦ Qbe Root • 7 months ago
Function.prototype.bind can be used to curry, but it's usually more verbose than using a specialized curry function, and may
make your intention less clear to someone reading the code who's familiar with how bind is usually used

make your intention less clear to someone reading the code who's familiar with how bind is usually used.
• Reply • Share ›

David Tang • 8 months ago
So what is the difference between currying and partial application? Can those words be used interchangeably?
• Reply • Share ›

Slava Ganzin → David Tang • 7 months ago
curry is partial application with 1 arity in math.
But in js everyone just don't give a fuck and mix them.
1 • Reply • Share ›

David Tang → Slava Ganzin • 7 months ago
and 1 arity means?
• Reply • Share ›

Slava Ganzin → David Tang • 7 months ago
arity mean number of arguments of a function
2 • Reply • Share ›

xaralabos karypidis • 8 months ago
Great explanation right in point! Thanks for writing this.
• Reply • Share ›

Subscribe   Add Disqus to your site Add Disqus Add   Privacy

COURSES >
(https://www.sitepoint.com/prem
q=&content_types[]=Course&utm_

3:07:36

JavaScript: Next Steps

M. David Green

★★★★★

(https://www.sitepoint.com/premium/cou
next-steps-2921/?
utm_source=sitepoint&utm_medium=relat

1:11:20

React The ES6 Way

Darin Haener

★★★★☆

(https://www.sitepoint.com/premium/cou
the-es6-way-2914/?
utm_source=sitepoint&utm_medium=relat

1:49:07

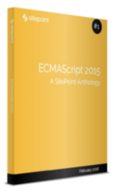
Your First Meteor 1.2 Application

David Turnbull

★★★★☆

(https://www.sitepoint.com/premium/cou
first-meteor-1-2-application-2919/?
utm_source=sitepoint&utm_medium=relat

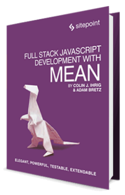
BOOKS >
(https://www.sitepoint.com/prem
q=&content_types[]=Book&utm_s



ECMAScript 2015: A SitePoint Anthology

James Hibbard

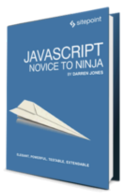
★★★★☆
(https://www.sitepoint.com/premium/boo
2015-a-sitepoint-anthology/?
utm_source=sitepoint&utm_medium=relat



Full Stack JavaScript Development with MEAN

Colin Ihrig

★★★★☆
(https://www.sitepoint.com/premium/boo
stack-javascript-development-with-
mean/?
utm_source=sitepoint&utm_medium=relat



JavaScript: Novice to Ninja

Darren Jones

★★★★☆

(https://www.sitepoint.com/premium/boo
novice-to-ninja/?

utm_source=sitepoint&utm_medium=relat

SCREENCASTS > (https://www.sitepoint.com/prem q=&content_types[]=ScreenCast&

Building Realtime Web Apps with Firebase

Thomas Greco

(https://www.sitepoint.com/premium/tutc realtime-web-apps-with-firebase/? utm_source=sitepoint&utm_medium=relat

Using Helpers and Conditionals in Ember

Thomas Greco

(https://www.sitepoint.com/premium/tutc helpers-and-conditionals-in-ember/? utm_source=sitepoint&utm_medium=relat

Modifying a Bootstrap Theme with Handlebars

Kauress

(https://www.sitepoint.com/premium/tutc a-bootstrap-theme-with-handlebars/? utm_source=sitepoint&utm_medium=relat