Kyle Shevlin
Software Engineer

April 26, 2019

335 strokes bestowed

Just Enough FP: Currying

edit

Currying is, by far, one of the coolest things I've learned in the last few years. When it clicked in my brain, it was such an intense epiphany that I literally ran to the other room and begged my wife to join me in the office so I could explain it to her on the whiteboard. She doesn't know a thing about programming, but she listened to me. "Yeah, Kyle, that *kinda* makes sense." Ha. I just had to share it with the first person I could find. Bless her for putting up with me.

Anecdote over, on with the blog post!

What is Currying?

My least jargon filled way of answering that question is this: Currying is the act of refactoring a function that normally receives all its arguments at once into a series of functions that only take one argument at a time.

In JavaScript, functions can receive any number of arguments. The number of arguments a function expects to receive is known as its *arity*. Normally, we do not worry too much about how many arguments a function receives. If it needs more than one argument, we write it so that it receives all its arguments at one time. I believe this makes a lot of intuitive sense and maps to how our brains model and understand functions. Give the function all the arguments, do something with them, give me back the result. Easy peasy. But it's not the only way.

In functional programming, all functions only receive one argument at a time. A function with an arity of one is known as a unary function. All curried functions are unary functions. A curried function that requires more than one argument to fulfill its operation, returns a new unary function with each argument until it has all the arguments it needs to finally evaluate.

Let me demonstrate the currying process with the canonical example, an `add` function.

```
// A common add function
function add(x, y) {
  return x + y
}

add(2, 3) // 5
```

This function has an arity of two, aka a *binary function*. It expects both arguments at the same time, adds them together, and gives the result back

to you. To make this a curried function, we are going to refactor it so it receives the `x` and `y` arguments one at a time.

```
// A curried add function
function add(x) {
  return function(y) {
    return x + y
  }
}

// Or its arrow function equivalent
const add = x => y => x + y

// If we only have our `x` argument, we can appl
// function awaiting the `y` argument with the `
const add2 = add(2)

// We can then supply the 'y' argument
add2(3) // 5

// For good measure, if you had both arguments c
// You can supply both by immediately invoking t
add(4)(6) // 10
```

Our curried function receives the `x` and `y` arguments one at a time. When we receive the `x` argument, we return a new function awaiting the `y`. This new function has our `x` value stored in closure. This is known as *partial application*. Partial application is a very powerful feature of currying that I'll explore in more detail in the next post, but for now just keep following along and try to grok it through context. It'll be ok.

If you're following with me, you now understand what currying is, but are probably still confused as to why this is an important technique. I'm going to give you an example that admittedly will skip ahead a bit, but should show you some of what gets unlocked when functions are curried.

Let's consider the array method `filter`. `filter` receives a predicate function (a function that returns a boolean) that operates on each item in an array and returns a new array with only the items that return true. You can check out the docs here. What if we could use the same filter on multiple arrays more easily? Let's create our own `filter` function that's curried.

```
const filter = predicate => array => array.filte
```

Our function receives the `predicate` argument first, returns a new function that awaits the `array` and then evaluates and returns the result. Let's create a predicate and pass it to filter.

```
const filter = predicate => array => array.filte
const filterForEvens = filter(x => x % 2 === 0)
```

We have now created a new function, `filterForEvens` which has the predicate partially applied. This predicate returns true if dividing by two returns a remainder of 0. The `filterForEvens` function awaits an an `array` argument. So let's give it some arrays.

```
const filter = predicate => array => array.filte
const filterForEvens = filter(x => x % 2 === 0)

const arr1 = [1, 2, 3, 4, 5, 6, 7]
const arr2 = [1, 4, 9, 16, 25, 36, 49]
const arr3 = [1, 8, 27, 64, 125, 216, 343]

filterForEvens(arr1) // [2, 4, 6]
filterForEvens(arr2) // [4, 16, 36]
filterForEvens(arr3) // [8, 64, 216]
```

By currying our `filter` function, we've given ourselves the opportunity to delay supplying the `array` argument. This allows us to supply as many different arrays to our `filterForEvens` function as we want and we never have to resupply the `predicate` argument.

As we'll see in the next post, currying and partial application is a powerful combination. We can DRY up code in our application through it's use. We'll also explore two other topics related to currying: *argument order* and *function composition*.

In this series, I'm going over material from my Just Enough Functional Programming in JavaScript course on egghead. This post is based on the lesson Refactor a Function to Use Currying in JavaScript.

Finished reading?

Here are a few options for what to do next.

Like

Liked the post? Click the beard up to 50 times to show it

 +0

Share

Sharing this post on Twitter & elsewhere is a great way to help me out

Click to share on Twitter

Support

Was this post valuable to you? Make a donation to show it

Make a Donation 

Related Posts:

Just Enough Functional Programming Course Launch

Just Enough FP: Higher Order Functions

Just Enough FP: Pure Functions

Just Enough FP: Immutability

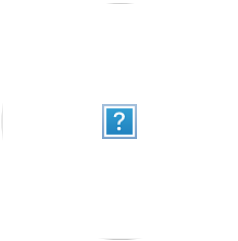
Just Enough FP: Partial Application

Just Enough FP: Argument Order

Just Enough FP: Pointfree

Just Enough FP: Composition

Tags Functional Programming JavaScript



Kyle Shevlin is a software engineer who specializes in JavaScript, React and front end web development.

Let's talk some more about JavaScript, React, and software engineering.

I write a newsletter to share my thoughts and the projects I'm working on. I would love for you to join the conversation. You can unsubscribe at any time.

First Name Email Address

You you@example.com

Submit

Just Enough Functional Programming

Check out my courses!

Liked the post? You might like my courses, too. Click the button to view this course or go to Courses for more information.

View on egghead.io

I would like give thanks to those who have contributed fixes and updates to this blog. If you see something that needs some love, you can join them. This blog is open sourced at https://github.com/kyleshevlin/blog