Taking the next steps in functional programming

Session at SAP Inside Track Bangalore, 01 Feb 2020 (remote)

INTRODUCTION

Begin with some definitions:

Currying: ...
Partial Application: ...
Point Free Coding: ...

Start demo in Chrome Developer Console at page https://ramda.com/docs

PART 1

```
// Get first set of Northwind products
Products = [PASTE].value

// Bonus: functions are first class, can assign to vars
t = console.table

// Have a quick look at the data, get a feel for what it looks like
t(Products)

// Pick out a lead product, for reference
Leader = Products[0]

// Have a quick look at the detail, what are the properties
t(Leader)

// First go at filtering for similarities to lead product, passing a function (first class), pre-ES6 syntax (function()), hard coded (not reusable)
Products.filter(function(product) {
  return product.SupplierID === Leader.SupplierID
}

// Quick re-look at results of previous evaluation
t($_)

// But - changing the filter condition is cumbersome and brittle
Products.filter(function(product) {
```

```
  return product.CategoryID === Leader.CategoryID
}
```

```
// Switch to ES6 before we continue; neater but still clunky generally
Products.filter(product => product.CategoryID === Leader.CategoryID)
```

```
// Swap out the predicate to a separate function (no difference except easier to start to think about)
hasLeadCategory = product => product.CategoryID === Leader.CategoryID
Products.filter(hasLeadCategory)
```

```
// But look - we're already inadvertently using point free coding - just supplying a function reference to filter(), with no arguments. Let's just let that sink in for a moment
```

PART 2

```
// And while we're thinking about this, let's see what currying actually looks like, before we use it for real
```

```
// Start with something simple, adding three numbers
addstuff = (a, b, c) => a + b + c
```

```
// We can call that as expected
addstuff(1,2,3) //=> 6
```

```
// But what if we don't supply all the arguments? Before we try it, note that "not supplying all the arguments" is basically what partial application is - partially applying the function. If we take it literally with this existing function definition, it doesn't work well
addstuff(1,2) //=> NaN
1 + 2 + undefined //=> NaN
```

```
// In order to more fully understand why partial application might be useful, we have to think about currying at the same time. With our definition of currying in mind, let's have a simple example in this context.
```

```
// Redefine addstuff as a curried function
addstuff = R.curry((a, b, c) => a + b + c) //=> a function, still
```

```
// Now try the previous call again
addstuff(1,2) //=> we get a function, that's waiting for the final argument
```

```
// In fact we can make any sort of partial application of this curried function now
```

```
addstuff() //=> function waiting for 3 arguments
addstuff(1) //=> function waiting for 2 arguments
addstuff(1,2) //=> function waiting for 1 argument
addstuff(1)(2) //=> also a function waiting for 1 argument

// The (1)(2) seems odd, but it's really just the same thing. Consider, instead of defining the function like this:
addstuff = R.curry((a, b, c) => a + b + c)

// ... we can define it like this:
addstuff = (a) => (b) => (c) => a + b + c

// ... but as well as being a little bit clunky we lose some flexibility, in that we can't do this:
addstuff(1,2)(3) //=> still waiting for a 'third' argument as the 2 is dropped
```

PART 3

```
// So now we have seen simple currying in real life, let's get back to the original problem to use currying to help with a real problem

// What we effectively want to do is build something that will give us flexibility in defining our filter predicate

// Let's remind ourselves of the data, this time using R.props as a convenience function. First, let's see what properties we have
t(Leader) //=> table of property/value pairs
t(Products.map(R.props(['ProductName', 'SupplierID', 'CategoryID'])))


// Let's pick Ikura as a reference product for now, there are other products in the same category
Ikura = Products[9]

// What we effectively want to express, for our predicate function, is something along these lines: "Given a property, a reference product, and a product under test, check if those products share the same property value":
(prop, ref, x) => x[prop] === ref[prop] //=> we've just inadvertently expressed a pure function definition (and it *is* pure).

// It's not assigned to anything, so we can't directly use it, so let's fix that, and curry it at the same time
hasSame = R.curry((prop, ref, x) => x[prop] === ref[prop]) //=> a curried fn
```

// Now we can start to use that to construct other useful small functions that we could then utilise in higher order contexts. Here's one:
sameCatAsIkura = hasSame('CategoryID', Ikura) //=> a(n already) partially applied function, waiting for the final argument (x)

// Let's try this out 'directly', with Product[12] (Konbu, CategoryID 8), and Product[13] (Pavlova, CategoryID 3)
sameCatAsIkura(Products[12]) //=> true
sameCatAsIkura(Products[15]) //=> false

// So we have a **curried** function **hasSame** that we've **partially applied** and saved as a little utility function **sameCatAsIkura** that we can use now in our filter ... because the Array prototype filter function takes a function with a single argument and expects a boolean result. Perfect!
t(Products.filter(sameCatAsIkura)) //=> table of 3 products Ikura, Konbu and Carnarvon Tigers

WRAPPING UP

// It would be amiss of me to not talk briefly about **how** this all works; there's a concept at play that is fundamental in JavaScript and other languages where functions are treated properly, i.e. as first class citizens. And that's the concept of **closures**. From MDN: *"A closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment). In other words, a closure gives you access to an outer function's scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time."*

// Let's look back at the creation of the **sameCatAsIkura** function, which was done like this:
sameCatAsIkura = hasSame('CategoryID', Ikura)

// What happened is that we called **hasSame**, passing two values, the 'CategoryID' scalar string and the Ikura map (or object). Remember that **hasSame** actually expects three arguments, and we could write it, in the more clunky (but still partially appliable) way like this:
hasSame = (prop) => (ref) => (x) => x[prop] === ref[prop]

// ... or even
hasSame = prop => ref => x => x[prop] === ref[prop]

// So the result of calling **hasSame('CategoryID', Ikura)** causes the values of both arguments to be enclosed in the surrounding lexical scope surrounding the function that is returned, i.e. the variables **x** and **prop**

in **x => x[prop] === ref[prop]**. They are captured and stored forever within the context of the final function (the one waiting for the **x**) for as long as that function exists).