|  |  |  |
| --- | --- | --- |
| **Javascript and Functional Programming: Currying (Pt.4)**  *Note: This is part of the “Javascript and Functional Programming” series on learning functional programming techniques in JavaScript ES6+. To start from the ground up check out*[*<Part 1>*](https://hackernoon.com/javascript-and-functional-programming-an-introduction-286aa625e26d)  https://cdn-images-1.medium.com/max/800/1*0e3zOemCl026ujZZFh-KiA.jpeg  Currying is when we call a function with fewer arguments than it expects. In turn, the invoked function returns a function that takes the remaining arguments.  const magicPhrase =   (magicWord) =>   (muggleWord) =>   magicWord + muggleWord  We could then invoke this function with the following pattern  https://cdn-images-1.medium.com/max/800/1*PdMGq8oovylfrpQWULwqvw.png  Call it maaagic  Writing functions that return functions, that in turn return some output (possibly another function!) can get quite cumbersome. Luckily we have functional JS helper libraries like Ramda and lodash which provide us with utility methods such as curry. The curry utility wraps normally declared functions and transforms them into a series of one-argument functions. So we could convert the previous code to:  import \_ from "lodash"  const magicPhrase = \_.curry((magicWord, muggleWord) => magicWord + muggleWord)  const muggleWordAccepter = magicPhrase("Abra kedabra ")  muggleWordAccepter("dishwasher")  Another example would be a revamped implementation of our favorite add function  import \_ from "lodash"  const addFunction = \_.curry((a, b) => a + b)  const addOne = add(1)  addTen(1)  So we are essentially, “pre loading” the add function with the first variable. Our function has the ability to remember the first value passed thanks to JS closure.  **Why You Should Care About Currying**   1. Currying gives us the ability to compose terse, concise and reusable functions.   2. We use these functions as clean, testable units of logic to compose the more logically complex parts of our programs.  3. With currying, any function that works on single elements can be converted into a function that works on arrays (lists), simply by wrapping it with *map*.  const getObjectId = (obj) => obj.id // works on single object  const arrayOfObjects = [{id: 1}, {id: 2}, {id: 3}, {id: 4}]  const arrayOfIDs = arrayOfObjects.map(getObjectId)  https://cdn-images-1.medium.com/max/800/1*o1vQuXVYH0JVSfxKDTwluQ.png  BAM! Our function that worked on single elements can work on arrays!  **Examples**  The only real way to get familiar with these concepts is to practice :) Let’s get to it. We shall start with one more example of converting a function that operates on a single element to a function that operates on an array.  const getFirstTwoLettersOfWord = (word) => word.substring(0,2)  // We can convert it, by wrapping it in the map method  ["aabb", "bbcc", "ccdd", "ddee"].map(getFirstTwoLettersOfWord)  https://cdn-images-1.medium.com/max/800/1*2vrdwE1jHeOidWZBq6Lpog.png  The next example comes out of the amazing Mostly Adequate guide, with a small ES6 refactors :)  Let’s refactor the max function so that it won’t reference any arguments.  arr = [2,4,6,8,9]  // LEAVE BE: const getMax = (x, y) => {  return x >= y ? x : y; };  // REFACTOR THIS ONE: const max = (arr) => {  return arr.reduce((acc, x) => {  return getMax(acc, x);  }, -Infinity); };  const max = arr.reduce(getMax, -Infinity)  Let’s wrap the native JS *slice method*so that it functional and curried.  import \_ from "lodash"  const arr = ["barney", "fred", "dave"]  arr.slice(0, 2) // ["barney", "fred"]  const slice = \_.curry((start, end, arr) => arr.slice(start, end)); const sliceWithSetIndexes = slice(0,2)  sliceWithSetIndexes(arr) // ["barney", "fred"]  **Summary**  We’ve seen several examples where we curry JS functions. Currying refers to the process of transforming a function with multiple arity (arguments accepted) into the same function with less arity. It utilizes JS closure to remember the arguments used in the previous invocations. Currying twists functions around so that they can work more naturally together. Its biggest advantage is that it easily allows for function composition, which we will explore in depth in the next post! | **Javascript и функциональное программирование: каррирование (ч.4)**  https://cdn-images-1.medium.com/max/800/1*0e3zOemCl026ujZZFh-KiA.jpeg  Каррированием называется метод, при котором мы вызываем функцию с меньшим количеством аргументов. Но функция эта возвращает значения со всеми недостающими аргументами.  const magicPhrase =   (magicWord) =>   (muggleWord) =>   magicWord + muggleWord  Таким образом, данную функцию можно вызвать следующим синтаксисом:   |  | | --- | | > **const muggleWordAccepter = magicPhrase (“Абракадабра”)**  **<- undefined**  **>muggleWordAccepter (“швабра”)**  **<- “Абракадабра швабра”** |   Волшебствооо  Написать функцию, которая будет возвращать какие-то выходные значения (возможно, другую функцию!), не так уж и просто. К счастью, существуют функциональные библиотеки-помощники по JS (Ramda, LoDash) со всяческими служебными методами, в т.ч. и с каррированием. Утилита *curry* находит функции с обычным объявлением и преобразует их в несколько функций с одним аргументом. Таким образом, из предыдущего кода можно сделать следующее:  import \_ from "lodash"  const magicPhrase = \_.curry((magicWord, muggleWord) => magicWord + muggleWord)  const muggleWordAccepter = magicPhrase("Абракадабра")  muggleWordAccepter("швабра")  Другой пример – это обновленная реализация нашей любимой функции *add*:  import \_ from "lodash"  const addFunction = \_.curry((a, b) => a + b)  const addOne = add(1)  addTen(1)  В первой переменной мы как бы «пред-загружаем» функцию add. И благодаря JS-замыканию наша функция может запоминать первое переданное ей значение.  **Зачем нужно каррирование**   1. С каррированием можно создавать краткие и лаконичные функции, подходящие для многоразового использования. 2. Эти функции используются в качестве чистых и пригодных для тестирования логических единиц при создании сложных с точки зрения логики частей программ. 3. Каррирование позволяет преобразовать все функции с одним элементом к функциям с поддержкой массивов (списки). Делается это путем оборачивания функции в *map*.   const getObjectId = (obj) => obj.id // работает с одним объектом  const arrayOfObjects = [{id: 1}, {id: 2}, {id: 3}, {id: 4}]  const arrayOfIDs = arrayOfObjects.map(getObjectId)  https://cdn-images-1.medium.com/max/800/1*o1vQuXVYH0JVSfxKDTwluQ.png  Бам! Наша функция с одним элементом стала работать с массивом!  **Примеры**  Единственный правильный способ разобраться во всем этом – практика :) Итак, приступим. Начнем с еще одного примера преобразования функции с одним элементом к функции с поддержкой массива.  const getFirstTwoLettersOfWord = (word) => word.substring(0,2)  // Мы преобразуем эту строку, оборачивая ее в метод map  ["aabb", "bbcc", "ccdd", "ddee"].map(getFirstTwoLettersOfWord)  https://cdn-images-1.medium.com/max/800/1*2vrdwE1jHeOidWZBq6Lpog.png  Следующий пример зародился из чудесного Mostly Adequate guide с небольшим рефакторингом ES6 :)  Давайте преобразуем функцию *max* так, чтобы она перестала ссылаться на какие-либо аргументы.  arr = [2,4,6,8,9]  // ОСТАВЛЯЕМ: const getMax = (x, y) => {  return x >= y ? x : y; };  // ДЕЛАЕМ РЕФАКТОРИНГ ВОТ ЭТОГО: const max = (arr) => {  return arr.reduce((acc, x) => {  return getMax(acc, x);  }, -Infinity); };  const max = arr.reduce(getMax, -Infinity)  Далее оборачиваем нативный JS-метод *slice*, делая его каррированным и работоспособным.  import \_ from "lodash"  const arr = ["барни", "фред", "дэйв"]  arr.slice(0, 2) // ["барни", "фред"]  const slice = \_.curry((start, end, arr) => arr.slice(start, end)); const sliceWithSetIndexes = slice(0,2)  sliceWithSetIndexes(arr) // ["барни", "фред"]  **Заключение**  Мы рассмотрели несколько примеров использования каррирования JS-функций. Каррирование относится к процессам преобразования функций с множественной арностью (количеством аргументов) к той же самой функции с меньшей арностью. Для запоминания ранее используемых аргументов используется замыкание JS. Каррирование реализует переплетение функций для улучшения их общей работоспособности. Но самое главное в том, что благодаря каррированию можно без проблем осуществлять композицию функций! |

https://hackernoon.com/javascript-and-functional-programming-currying-pt-4-96e3230782ab