JavaScript Functional Library Project

- Due No Due Date
- Points 1
- Submitting a website url

 (https://github.com/learn-co-curriculum/phase-1-javascript-functional-library-project)  (https://github.com/learn-co-curriculum/phase-1-javascript-functional-library-project/issues/new)

Learning Goals

- Gain a greater understanding of JavaScript's built-in collection-processing methods
- Gain a greater understanding of callback functions

Introduction

In this lab, you will gain a deeper understanding of JavaScript's built in collection-processing methods (`map` , `filter` etc.) by building your own implementation of them. You will also have the opportunity to practice using callbacks, including calling a callback from within a function, passing a callback to a function, and passing data between functions and callbacks.

The programming approach you will be using in this lab is an example of ***functional programming (FP)***. There is nothing new or different here — we've been guiding you all along to think in the "FP" mindset — but you should use this as an opportunity to start understanding some important distinctions between functional programming and other styles of programming.

A complete understanding of functional programming requires an understanding of a number of advanced topics in JavaScript, including **pure functions**, **side effects**, and **immutability**. However, at its most basic, FP can be understood as a programming style in which data manipulation occurs through functions that return the result of the manipulation without modifying the state of the original data. This style of programming can be contrasted with programming approaches that use **shared state**, in which data is manipulated and the results stored in a central location (commonly known as **state**). You will learn more about shared state when you get to React in the next phase.

Read through **this blog post about functional programming** (https://medium.com/javascript-scene/master-the-javascript-interview-what-is-functional-programming-7f218c68b3a0) before continuing with the lab, but don't worry too much if you don't understand everything you read. Your

goal should be to begin to get a feel for the concepts and for the distinction between functional programming and other styles of programming.

Instructions

Listed below are function signatures for each of the functions you will need to build. Each signature details what the name, parameters, and return value of the function should be. Pay close attention to these requirements as you work your way through. There are also some sample function calls provided with their expected return values; be sure to use them to test your functions.

The functions are divided into three categories: array functions, object functions, and functions that should work with **either** collection type. Your job is to develop the code to implement these functions.

The point of this exercise is to build ***your own implementation*** of the collection-processing methods. Don't simply re-use the built-in methods! Leverage all you know about callbacks, passing data, etc. to prove that you could build your own collection-processing framework whenever ***you*** want.

Hint: For the first set of functions, you will need to figure out how to make them work with either arrays or objects. There are multiple ways you could approach this. One option is to use an `if` statement to determine whether the collection is an object or an array and then process the collection accordingly. However, this approach would require writing two different versions of your code for each function, which doesn't sound very efficient. Another (better) option is to determine whether the collection is an object or an array and, if it's an object, use a JavaScript `Object` method to create an array that contains the object's values. You then only need to write code that processes an array, regardless of what data structure is passed in to your function. Use your Googling skills to figure out how to do this.

Collection Functions (Arrays or Objects)

myEach

```
myEach(collection, callback)
```

Parameter(s):

- a collection (either an object or an array)
- a callback function

Return value:

- The original collection

Behavior:

Iterates over the collection of elements, passing each element in turn to the callback function. Returns the original, unmodified, collection.

Example function calls:

```
myEach([1, 2, 3], alert);
=> alerts each number in turn and returns the original collection

myEach({one: 1, two: 2, three: 3}, alert);
=> alerts each number value in turn and returns the original collection
```

myMap

```
myMap(collection, callback)
```

Parameter(s):

- a collection (either an object or an array)
- a callback function

Return value:

- A new array

Behavior:

Produces a new array of values by mapping each value in `collection` through a transformation function, `callback`. Returns the new array without modifying the original.

Example function calls:

```
myMap([1, 2, 3], function(num){ return num * 3; });
=> [3, 6, 9]

myMap({one: 1, two: 2, three: 3}, function(num, key){ return num * 3; });
=> [3, 6, 9]
```

myReduce

`myReduce(collection, callback, acc)`

Parameter(s):

- a collection (either an object or an array)
- a callback function
- a starting value for the accumulator (optional)

Return value:

- A single value

Behavior:

Reduce iterates through a `collection` of values and boils it down into a single value. `acc` (short for accumulator) starts at the value that's passed in as an argument, and with each successive step is updated to the return value of `callback`. If a start value is not passed to `myReduce`, the first value in the collection should be used as the start value.

The `callback` is passed three arguments: the current value of `acc`, the current element/value in our iteration, and a reference to the entire collection.

Hint: For the case when a start value for the accumulator is not passed in as an argument, think about how you'll need to adjust your function to account for the fact that the first element of the collection has already been accounted for.

Example function calls:

```
myReduce([1, 2, 3], function(acc, val, collection) { return acc + val; }, 10);
=> 16

myReduce({one: 1, two: 2, three: 3}, function(acc, val, collection) { return acc + val; });
=> 6
```

myFind

`myFind(collection, predicate)`

Parameter(s):

- a collection (either an object or an array)
- a predicate (a callback function that returns `true` or `false`)

Return value:

- A single value

Behavior:

Looks through each value in the `collection`, returning the first one that passes a truth test (`predicate`) or undefined if no value passes the test. The function should return as soon as it finds an acceptable element, without traversing the rest of the collection.

Example function calls:

```
myFind([1, 2, 3, 4, 5, 6], function(num){ return num % 2 == 0; });
=> 2

myFind({one: 1, three: 3, four: 4, six: 6}, function(num){ return num % 2 == 0; });
=> 4
```

myFilter

`myFilter(collection, predicate)`

Parameter(s):

- a collection (either an object or an array)
- a predicate (a callback function that returns `true` or `false`)

Return value:

- An array

Behavior:

Looks through each value in the `collection`, returning an array of all the values that pass a truth test (`predicate`). If no matching values are found, it should return an empty array.

Example function call:

```
myFilter([1, 2, 3, 4, 5, 6], function(num){ return num % 2 == 0; });
=> [2, 4, 6]

myFilter({one: 1, three: 3, five: 5}, function(num){ return num % 2 == 0; })
=> []
```

mySize

`mySize(collection)`

Parameter(s):

- a collection (either an object or an array)

Return value:

- An integer

Behavior:

Return the number of values in the `collection`.

Example function calls:

```
mySize({one: 1, two: 2, three: 3});
=> 3

mySize([]);
=> 0
```

Array Functions

myFirst

`myFirst(array, [n])`

Parameter(s):

- an array
- an integer (optional)

Return value:

- A single element **OR** an array

Behavior:

Returns the first element of an `array`. Passing `n` will return the first **n** elements of the array.

Example function calls:

```
myFirst([5, 4, 3, 2, 1]);
=> 5
```

```
myFirst([5, 4, 3, 2, 1], 3);
=> [5, 4, 3]
```

myLast

```
myLast(array, [n])
```

Parameter(s):

- an array
- an integer (optional)

Return value:

- A single element **OR** an array

Behavior:

Returns the last element of an `array`. Passing `n` will return the last **n** elements of the array.

Example function calls:

```
myLast([5, 4, 3, 2, 1]);
=> 1

myLast([5, 4, 3, 2, 1], 3);
=> [3, 2, 1]
```

BONUS: mySortBy

Note: Coding the `mySortBy` function is optional for this lab, so the tests for it have been disabled. You are free to skip it, but if you'd like to complete this additional challenge, simply un-comment out the relevant test code in `test/indexTest.js`.

```
mySortBy(array, callback)
```

Parameter(s):

- an array
- a callback function

Return value:

- A new array

Behavior:

Returns a sorted copy of `array`, ranked in ascending order by the results of running each value through `callback`. The **values from the original array** (not the transformed values) should be returned in the sorted copy, in ascending order by the value returned by `callback`.

Note: The point of this exercise is not to write your own sorting algorithm and you are free to use the native **JS sort (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/sort#sorting_non-ascii_characters)**. You **will**, however, need to construct your `compareFunction` (see the documentation) so that it will handle either numeric or string values.

Example function calls:

```
mySortBy([1, 2, 3, 4, 5, 6], function(num){ return Math.sin(num) });
=> [5, 4, 6, 3, 1, 2];

const stooges = [{name: 'moe', age: 40}, {name: 'larry', age: 50}, {name: 'curly', age: 60}];
mySortBy(stooges, function(stooge){ return stooge.name });
=> [{name: 'curly', age: 60}, {name: 'larry', age: 50}, {name: 'moe', age: 40}];
```

BONUS: If you would like to go deeper and try to construct your own sorting algorithm, this is a great extension. Check out this list of **sorting algorithms (http://blog.benoitvallon.com/sorting-algorithms-in-javascript/sorting-algorithms-in-javascript-all-the-code/)** implemented in JS with additional resources.

BONUS: myFlatten

Note: Coding the `myFlatten` function is optional for this lab, so the tests for it have been disabled. You are free to skip it, but if you'd like to complete this additional challenge, simply un-comment out the relevant test code in `test/indexTest.js`.

`myFlatten(array, [shallow], newArr=[])`

Parameter(s):

- an array
- a boolean value (optional)
- a new array (with an assigned default value of an empty array) that will contain the flattened elements

Return value:

- The new array

Behavior:

Flattens a nested `array` (the nesting can be to any depth).

If you pass `true` for the second argument, the array will only be flattened a single level.

Example function calls:

```
myFlatten([1, [2], [3, [[4]]]]);
=> [1, 2, 3, 4];

myFlatten([1, [2], [3, [[4]]]], true);
=> [1, 2, 3, [[4]]];
```

Hint: This one is challenging! You will need to use recursion to make this work for the multi-level case. Think about why we need that third argument here. Also think about how to handle the two optional arguments when you call the function recursively.

Object Functions

myKeys

`myKeys(object)`

Parameter(s):

- an object

Return value:

- An array

Behavior:

Retrieve all the names of the `object` 's enumerable properties.

Example function call:

```
myKeys({one: 1, two: 2, three: 3});
=> ["one", "two", "three"]
```

myValues

`myValues(object)`

Parameter(s):

- an object

Return value:

- an array

Behavior:

Return all of the values of the `object` 's properties.

Example function call:

```
myValues({one: 1, two: 2, three: 3});
=> [1, 2, 3]
```

Conclusion

Building a functional library is a great experience for learning to see how many functions can build off of each other. This lab asked you to take on some of the basic tasks that you would face when writing a functional library.

Expand your vocabulary by visiting a library like **lodash (https://lodash.com)** or **ramda (https://ramdajs.com/docs/)**. Look at methods like Ramda's **filter (https://ramdajs.com/docs/#filter)** or **flip (https://ramdajs.com/docs/#flip)**. Can you imagine how to write that? These libraries are providing the functionality just like you did!

You've pushed your skills to a whole new level. Congratulations!

Resources

- **lodash (https://lodash.com)**
- **ramda (https://ramdajs.com/docs/)**