FUNCTIONAL PROGRAMMING

Functional programming is a style of programming where solutions are simple, isolated functions, without any side effects outside of the function scope.

INPUT -> PROCESS -> OUTPUT

Functional programming is about:

1) Isolated functions - there is no dependence on the state of the program, which includes global variables that are subject to change

2) Pure functions - the same input always gives the same output

3) Functions with limited side effects - any changes, or mutations, to the state of the program outside the function are carefully controlled

**Functional Programming: Understand Functional Programming Terminology**

The FCC Team had a mood swing and now wants two types of tea: green tea and black tea. General Fact: Client mood swings are pretty common.

With that information, we'll need to revisit the getTea function from last challenge to handle various tea requests. We can modify getTea to accept a function as a parameter to be able to change the type of tea it prepares. This makes getTea more flexible, and gives the programmer more control when client requests change.

But first, let's cover some functional terminology:

*Callbacks* are the functions that are slipped or passed into another function to decide the invocation of that function. You may have seen them passed to other methods, for example in filter, the callback function tells JavaScript the criteria for how to filter an array.

Functions that can be assigned to a variable, passed into another function, or returned from another function just like any other normal value, are called *first class* functions. In JavaScript, all functions are first class functions.

The functions that take a function as an argument, or return a function as a return value are called *higher order* functions.

When the functions are passed in to another function or returned from another function, then those functions which gets passed in or returned can be called a *lambda*.

## Functional Programming: Understand the Hazards of Using Imperative Code

Functional programming is a good habit. It keeps your code easy to manage, and saves you from sneaky bugs. But before we get there, let's look at an imperative approach to programming to highlight where you may have issues.

In English (and many other languages), the imperative tense is used to give commands. Similarly, an imperative style in programming is one that gives the computer a set of statements to perform a task.

Often the statements change the state of the program, like updating global variables. A classic example is writing a for loop that gives exact directions to iterate over the indices of an array.

In contrast, functional programming is a form of declarative programming. You tell the computer what you want done by calling a method or function.

JavaScript offers many predefined methods that handle common tasks so you don't need to write out how the computer should perform them. For example, instead of using the for loop mentioned above, you could call the map method which handles the details of iterating over an array. This helps to avoid semantic errors, like the "Off By One Errors" that were covered in the Debugging section.

Consider the scenario: you are browsing the web in your browser, and want to track the tabs you have opened. Let's try to model this using some simple object-oriented code.

A Window object is made up of tabs, and you usually have more than one Window open. The titles of each open site in each Window object is held in an array. After working in the browser (opening new tabs, merging windows, and closing tabs), you want to print the tabs that are still open. Closed tabs are removed from the array and new tabs (for simplicity) get added to the end of it.

The code editor shows an implementation of this functionality with functions for tabOpen(), tabClose(), and join(). The array tabs is part of the Window object that stores the name of the open pages.

## Functional Programming: Avoid Mutations and Side Effects Using Functional Programming

If you haven't already figured it out, the issue in the previous challenge was with the splice call in the tabClose() function. Unfortunately, splice changes the original array it is called on, so the second call to it used a modified array, and gave unexpected results.

This is a small example of a much larger pattern - you call a function on a variable, array, or an object, and the function changes the variable or something in the object.

One of the core principles of functional programming is to not change things. Changes lead to bugs. It's easier to prevent bugs knowing that your functions don't change anything, including the function arguments or any global variable.

The previous example didn't have any complicated operations but the splice method changed the original array, and resulted in a bug.

Recall that in functional programming, changing or altering things is called mutation, and the outcome is called a side effect. A function, ideally, should be a pure function, meaning that it does not cause any side effects.

Let's try to master this discipline and not alter any variable or object in our code.

## Functional Programming: Pass Arguments to Avoid External Dependence in a Function

The last challenge was a step closer to functional programming principles, but there is still something missing.

We didn't alter the global variable value, but the function incrementer would not work without the global variable fixedValue being there.

Another principle of functional programming is to always declare your dependencies explicitly. This means if a function depends on a variable or object being present, then pass that variable or object directly into the function as an argument.

There are several good consequences from this principle. The function is easier to test, you know exactly what input it takes, and it won't depend on anything else in your program.

This can give you more confidence when you alter, remove, or add new code. You would know what you can or cannot change and you can see where the potential traps are.

Finally, the function would always produce the same output for the same set of inputs, no matter what part of the code executes it.

## Functional Programming: Refactor Global Variables Out of Functions

So far, we have seen two distinct principles for functional programming:

1) Don't alter a variable or object - create new variables and objects and return them if need be from a function.

2) Declare function arguments - any computation inside a function depends only on the arguments, and not on any global object or variable.

Adding one to a number is not very exciting, but we can apply these principles when working with arrays or more complex objects.

Rewrite the code so the global array bookList is not changed inside either function. The add function should add the given bookName to the end of the array passed to it and return a new array (list). The remove function should remove the given bookName from the array passed to it.

**Note:** Both functions should return an array, and any new parameters should be added before the bookName parameter.

## Functional Programming: Use the map Method to Extract Data from an Array

So far we have learned to use pure functions to avoid side effects in a program. Also, we have seen the value in having a function only depend on its input arguments.

This is only the beginning. As its name suggests, functional programming is centered around a theory of functions.

It would make sense to be able to pass them as arguments to other functions, and return a function from another function. Functions are considered first class objects in JavaScript, which means they can be used like any other object. They can be saved in variables, stored in an object, or passed as function arguments.

Let's start with some simple array functions, which are methods on the array object prototype. In this exercise we are looking at Array.prototype.map(), or more simply map.

The map method iterates over each item in an array and returns a new array containing the results of calling the callback function on each element. It does this without mutating the original array.

When the callback is used, it is passed three arguments. The first argument is the current element being processed. The second is the index of that element and the third is the array upon which the map method was called.

See below for an example using the map method on the users array to return a new array containing only the names of the users as elements. For simplicity, the example only uses the first argument of the callback.

const users = [

{ name: 'John', age: 34 },

{ name: 'Amy', age: 20 },

{ name: 'camperCat', age: 10 }

];

const names = users.map(user => user.name);

console.log(names); // [ 'John', 'Amy', 'camperCat' ]

The watchList array holds objects with information on several movies. Use map on watchList to assign a new array of objects with only title and rating keys to the ratings variable. The code in the editor currently uses a for loop to do this, so you should replace the loop functionality with your map expression.

## Functional Programming: Implement map on a Prototype

As you have seen from applying Array.prototype.map(), or simply map() earlier, the map method returns an array of the same length as the one it was called on. It also doesn't alter the original array, as long as its callback function doesn't.

In other words, map is a pure function, and its output depends solely on its inputs. Plus, it takes another function as its argument.

It would teach us a lot about map to try to implement a version of it that behaves exactly like the Array.prototype.map() with a for loop or Array.prototype.forEach().

Note: A pure function is allowed to alter local variables defined within its scope, although, it's preferable to avoid that as well.

Write your own Array.prototype.myMap(), which should behave exactly like Array.prototype.map(). You may use a for loop or the forEach method.

## Functional Programming: Use the filter Method to Extract Data from an Array

Another useful array function is Array.prototype.filter(), or simply filter().

filter calls a function on each element of an array and returns a new array containing only the elements for which that function returns true. In other words, it filters the array, based on the function passed to it. Like map, it does this without needing to modify the original array.

The callback function accepts three arguments. The first argument is the current element being processed. The second is the index of that element and the third is the array upon which the filter method was called.

See below for an example using the filter method on the users array to return a new array containing only the users under the age of 30. For simplicity, the example only uses the first argument of the callback.

const users = [

{ name: 'John', age: 34 },

{ name: 'Amy', age: 20 },

{ name: 'camperCat', age: 10 }

];

const usersUnder30 = users.filter(user => user.age < 30);

console.log(usersUnder30); // [ { name: 'Amy', age: 20 }, { name: 'camperCat', age: 10 } ]

The variable watchList holds an array of objects with information on several movies. Use a combination of filter and map on watchList to assign a new array of objects with only title and rating keys. The new array should only include objects where imdbRating is greater than or equal to 8.0. Note that the rating values are saved as strings in the object and you may need to convert them into numbers to perform mathematical operations on them.

## Functional Programming: Implement the filter Method on a Prototype

It would teach us a lot about the filter method if we try to implement a version of it that behaves exactly like Array.prototype.filter(). It can use either a for loop or Array.prototype.forEach().

Note: A pure function is allowed to alter local variables defined within its scope, although, it's preferable to avoid that as well.

Write your own Array.prototype.myFilter(), which should behave exactly like Array.prototype.filter(). You may use a for loop or the Array.prototype.forEach() method.

## Functional Programming: Return Part of an Array Using the slice Method

The slice method returns a copy of certain elements of an array. It can take two arguments, the first gives the index of where to begin the slice, the second is the index for where to end the slice (and it's non-inclusive). If the arguments are not provided, the default is to start at the beginning of the array through the end, which is an easy way to make a copy of the entire array. The slice method does not mutate the original array, but returns a new one.

Here's an example:

var arr = ["Cat", "Dog", "Tiger", "Zebra"];

var newArray = arr.slice(1, 3);

// Sets newArray to ["Dog", "Tiger"]

Use the slice method in the sliceArray function to return part of the anim array given the provided beginSlice and endSlice indices. The function should return an array.

## Functional Programming: Remove Elements from an Array Using slice Instead of splice

A common pattern while working with arrays is when you want to remove items and keep the rest of the array. JavaScript offers the splice method for this, which takes arguments for the index of where to start removing items, then the number of items to remove. If the second argument is not provided, the default is to remove items through the end. However, the splice method mutates the original array it is called on. Here's an example:

var cities = ["Chicago", "Delhi", "Islamabad", "London", "Berlin"];

cities.splice(3, 1); // Returns "London" and deletes it from the cities array

// cities is now ["Chicago", "Delhi", "Islamabad", "Berlin"]

As we saw in the last challenge, the slice method does not mutate the original array, but returns a new one which can be saved into a variable. Recall that the slice method takes two arguments for the indices to begin and end the slice (the end is non-inclusive), and returns those items in a new array. Using the slice method instead of splice helps to avoid any array-mutating side effects.

Rewrite the function nonMutatingSplice by using slice instead of splice. It should limit the provided cities array to a length of 3, and return a new array with only the first three items.

Do not mutate the original array provided to the function.

## Functional Programming: Combine Two Arrays Using the concat Method

Concatenation means to join items end to end. JavaScript offers the concat method for both strings and arrays that work in the same way. For arrays, the method is called on one, then another array is provided as the argument to concat, which is added to the end of the first array. It returns a new array and does not mutate either of the original arrays. Here's an example:

[1, 2, 3].concat([4, 5, 6]);

// Returns a new array [1, 2, 3, 4, 5, 6]

Use the concat method in the nonMutatingConcat function to concatenate attach to the end of original. The function should return the concatenated array.

## Functional Programming: Add Elements to the End of an Array Using concat Instead of push

Functional programming is all about creating and using non-mutating functions.

The last challenge introduced the concat method as a way to combine arrays into a new one without mutating the original arrays. Compare concat to the push method. Push adds an item to the end of the same array it is called on, which mutates that array. Here's an example:

var arr = [1, 2, 3];

arr.push([4, 5, 6]);

// arr is changed to [1, 2, 3, [4, 5, 6]]

// Not the functional programming way

Concat offers a way to add new items to the end of an array without any mutating side effects.

Change the nonMutatingPush function so it uses concat to add newItem to the end of original instead of push. The function should return an array.

## Functional Programming: Use the reduce Method to Analyze Data

Array.prototype.reduce(), or simply reduce(), is the most general of all array operations in JavaScript. You can solve almost any array processing problem using the reduce method.

The reduce method allows for more general forms of array processing, and it's possible to show that both filter and map can be derived as special applications of reduce. The reduce method iterates over each item in an array and returns a single value (i.e. string, number, object, array). This is achieved via a callback function that is called on each iteration.

The callback function accepts four arguments. The first argument is known as the accumulator, which gets assigned the return value of the callback function from the previous iteration, the second is the current element being processed, the third is the index of that element and the fourth is the array upon which reduce is called.

In addition to the callback function, reduce has an additional parameter which takes an initial value for the accumulator. If this second parameter is not used, then the first iteration is skipped and the second iteration gets passed the first element of the array as the accumulator.

See below for an example using reduce on the users array to return the sum of all the users' ages. For simplicity, the example only uses the first and second arguments.

const users = [

{ name: 'John', age: 34 },

{ name: 'Amy', age: 20 },

{ name: 'camperCat', age: 10 }

];

const sumOfAges = users.reduce((sum, user) => sum + user.age, 0);

console.log(sumOfAges); // 64

In another example, see how an object can be returned containing the names of the users as properties with their ages as values.

const users = [

{ name: 'John', age: 34 },

{ name: 'Amy', age: 20 },

{ name: 'camperCat', age: 10 }

];

const usersObj = users.reduce((obj, user) => {

obj[user.name] = user.age;

return obj;

}, {});

console.log(usersObj); // { John: 34, Amy: 20, camperCat: 10 }

The variable watchList holds an array of objects with information on several movies. Use reduce to find the average IMDB rating of the movies **directed by Christopher Nolan**. Recall from prior challenges how to filter data and map over it to pull what you need. You may need to create other variables, and return the average rating from getRating function. Note that the rating values are saved as strings in the object and need to be converted into numbers before they are used in any mathematical operations.

## Functional Programming: Sort an Array Alphabetically using the sort Method

The sort method sorts the elements of an array according to the callback function.

For example:

function ascendingOrder(arr) {

return arr.sort(function(a, b) {

return a - b;

});

}

ascendingOrder([1, 5, 2, 3, 4]);

// Returns [1, 2, 3, 4, 5]

function reverseAlpha(arr) {

return arr.sort(function(a, b) {

return a === b ? 0 : a < b ? 1 : -1;

});

}

reverseAlpha(['l', 'h', 'z', 'b', 's']);

// Returns ['z', 's', 'l', 'h', 'b']

JavaScript's default sorting method is by string Unicode point value, which may return unexpected results. Therefore, it is encouraged to provide a callback function to specify how to sort the array items. When such a callback function, normally called compareFunction, is supplied, the array elements are sorted according to the return value of the compareFunction: If compareFunction(a,b) returns a value less than 0 for two elements a and b, then a will come before b. If compareFunction(a,b) returns a value greater than 0 for two elements a and b, then b will come before a. If compareFunction(a,b) returns a value equal to 0 for two elements a and b, then a and b will remain unchanged.

Use the sort method in the alphabeticalOrder function to sort the elements of arr in alphabetical order.

## Functional Programming: Return a Sorted Array Without Changing the Original Array

A side effect of the sort method is that it changes the order of the elements in the original array. In other words, it mutates the array in place. One way to avoid this is to first concatenate an empty array to the one being sorted (remember that slice and concat return a new array), then run the sort method.

Use the sort method in the nonMutatingSort function to sort the elements of an array in ascending order. The function should return a new array, and not mutate the globalArray variable.

## Functional Programming: Split a String into an Array Using the split Method

The split method splits a string into an array of strings. It takes an argument for the delimiter, which can be a character to use to break up the string or a regular expression. For example, if the delimiter is a space, you get an array of words, and if the delimiter is an empty string, you get an array of each character in the string.

Here are two examples that split one string by spaces, then another by digits using a regular expression:

var str = "Hello World";

var bySpace = str.split(" ");

// Sets bySpace to ["Hello", "World"]

var otherString = "How9are7you2today";

var byDigits = otherString.split(/\d/);

// Sets byDigits to ["How", "are", "you", "today"]

Since strings are immutable, the split method makes it easier to work with them.

Use the split method inside the splitify function to split str into an array of words. The function should return the array. Note that the words are not always separated by spaces, and the array should not contain punctuation.

## Functional Programming: Combine an Array into a String Using the join Method

The join method is used to join the elements of an array together to create a string. It takes an argument for the delimiter that is used to separate the array elements in the string.

Here's an example:

var arr = ["Hello", "World"];

var str = arr.join(" ");

// Sets str to "Hello World"

Use the join method (among others) inside the sentensify function to make a sentence from the words in the string str. The function should return a string. For example, "I-like-Star-Wars" would be converted to "I like Star Wars". For this challenge, do not use the replace method.

## Functional Programming: Apply Functional Programming to Convert Strings to URL Slugs

The last several challenges covered a number of useful array and string methods that follow functional programming principles. We've also learned about reduce, which is a powerful method used to reduce problems to simpler forms. From computing averages to sorting, any array operation can be achieved by applying it. Recall that map and filter are special cases of reduce.

Let's combine what we've learned to solve a practical problem.

Many content management sites (CMS) have the titles of a post added to part of the URL for simple bookmarking purposes. For example, if you write a Medium post titled "Stop Using Reduce", it's likely the URL would have some form of the title string in it (".../stop-using-reduce"). You may have already noticed this on the freeCodeCamp site.

Fill in the urlSlug function so it converts a string title and returns the hyphenated version for the URL. You can use any of the methods covered in this section, and don't use replace. Here are the requirements:

The input is a string with spaces and title-cased words

The output is a string with the spaces between words replaced by a hyphen (-)

The output should be all lower-cased letters

The output should not have any spaces

## Functional Programming: Use the every Method to Check that Every Element in an Array Meets a Criteria

The every method works with arrays to check if every element passes a particular test. It returns a Boolean value - true if all values meet the criteria, false if not.

For example, the following code would check if every element in the numbers array is less than 10:

var numbers = [1, 5, 8, 0, 10, 11];

numbers.every(function(currentValue) {

return currentValue < 10;

});

// Returns false

Use the every method inside the checkPositive function to check if every element in arr is positive. The function should return a Boolean value.

## Functional Programming: Use the some Method to Check that Any Elements in an Array Meet a Criteria

The some method works with arrays to check if any element passes a particular test. It returns a Boolean value - true if any of the values meet the criteria, false if not.

For example, the following code would check if any element in the numbers array is less than 10:

var numbers = [10, 50, 8, 220, 110, 11];

numbers.some(function(currentValue) {

return currentValue < 10;

});

// Returns true

Use the some method inside the checkPositive function to check if any element in arr is positive. The function should return a Boolean value.

## Functional Programming: Introduction to Currying and Partial Application

The arity of a function is the number of arguments it requires. Currying a function means to convert a function of N arity into N functions of arity 1.

In other words, it restructures a function so it takes one argument, then returns another function that takes the next argument, and so on.

Here's an example:

//Un-curried function

function unCurried(x, y) {

return x + y;

}

//Curried function

function curried(x) {

return function(y) {

return x + y;

}

}

//Alternative using ES6

const curried = x => y => x + y

curried(1)(2) // Returns 3

This is useful in your program if you can't supply all the arguments to a function at one time. You can save each function call into a variable, which will hold the returned function reference that takes the next argument when it's available. Here's an example using the curried function in the example above:

// Call a curried function in parts:

var funcForY = curried(1);

console.log(funcForY(2)); // Prints 3

Similarly, partial application can be described as applying a few arguments to a function at a time and returning another function that is applied to more arguments. Here's an example:

//Impartial function

function impartial(x, y, z) {

return x + y + z;

}

var partialFn = impartial.bind(this, 1, 2);

partialFn(10); // Returns 13

Fill in the body of the add function so it uses currying to add parameters x, y, and z.