Functional Programming & JavaScript

Notes open for creative commons use @ developer blog: https://unfoldkyle.com, github: SmilingStallman, *email: kmiskell@protonmail.com*

Learning Resources:
Professor Friby's Mostly Adequate Guide to Functional Programming
Functional-Light Javascript (Simpson, 2017)
JavaScript Closure Guide

Functional Programming 101

-FP is declarative over imperative. Instead of using state, and mutable values, it strives to be stateless with immutable values.

-Better syntax with greater simplicity and readability. Less local variables and jumping around between functions. Like reading a book straight through instead of jumping back and forth from chapter to chapter.

-See intro to *Functional-Light JavaScript* for various additional resources (blogs, libraries, books, etc.)

ES6 Review

Default Params
-can have defaults for if no arg passed in: *function foo(x = 3) {…}*

-Function "arity" is the num of parameters in a function. 1 = unary, 2 = binary, 3+ = n-ary. Can inspect arity with *functionName.length.* A rest parameter is not counted in arity. If want to tell number of arguments passed in, call global *arguments.length* from within function.

Spread (Rest) Parameter
-If pass have *...someRestParam* as a parameter, you can pass in as many args as you like and all args passed in after list of named parameters will go into an array named *someRestParam.*
-Can also pass in a rest param during function call to pass array in as multiple args.

Destructuring
-Allows you to unpack elements or properties from arrays, objects, etc. into distinct variables by using [] or {} as left-hand assignment operator.
-With objects left hand holds value of object property with same name
    -ex. *const {b} = {a: 1, b: 2, c: 3) //b=2).*
-With arrays, based index position where extra commas skip index
    -ex. *const [a, , , b] = [1, 2, 3, 4, 5] //a=1, b=4*
-Important to recognize when doing *const {c, d} =* or *const [f,g] = [f, g]* are creating two separate consts in each instance.

Destructuring Combined with Default Param

-If *someParameter = {}* or *someParameter =* [], the default is an empty obj/array. Can combine this by giving default parameter a destructured binding (ex. *[x, y, ...args]*) then, whatever pass in will take place of default empty obj/array, and destructure into destructured bindings

-Ex.

```
function foo({a, c} = {}){
  console.log(`c=${c}`);
}
foo({b:2, c:3, e:5});                    //prints c=3

function arrayFoo([a, b, ...rest] = []){
  console.log(`a=${a}, rest=${rest}`);
}
arrayFoo([1, 2, 3, 4, 5, 6]);                    //prints a=1, rest=3,4,5,6
```

-Benefit is that instead of saying, "these are the parameters passed in and they have to be in this order," you say, "these are the parameters that may be passed in, you decide which ones you want to pass in and which to omit" which makes how you can call functions significantly more versatile. This technique is called using "named parameters" and "named arguments"

High Order Functions
-A function that takes in a function as an arg or returns a function
-When pass in function as arg such as in calling *forEach(list, func)*, can then call *func()* with args from inside *forEach().* Function can also return a call to a function that is defined outside the function.

Functions

First-Class Functions
-Functions are first class in a lang, if treated as "first-class citizens." ie, allow passing of functions as args to other functions, returning them as values from other functions, assigning them to variables or storing them in data-structures.

-Review: Functions can exist as bound to variables also. In that case, the binding references the function, and is called instead of the function name.
-ex. *const theBinding = function(arg) { ...}* //would call via *theBinding(someArg)*

-Code becomes more re-usable if they take in functions (high order functions) instead of defining functions as arguments within the parameters list. Functions also more re-usable by making names of functions and arguments relevant to many current and potential future use cases.

Pure Functions
-Pure functions are those that produce output without having any effect on any code input to the function or any side-effects on any code outside the function. They take in an input, produce and output, and affect nothing else. They also always produce the same output for the same input.

-In FP, functions should be pure. Don't use a method like *splice* to get a value, as it destroys the array. Use *slice* instead, as it gets the value without destroying the array (a side effect).

-A function that depends on mutable data (ex. *let*) is not pure, as if that data changes, the result of the function may also change. Use *const* instead of *let*. Stateless.

-If have mutable object (ex. *const, someArray*), can turn immutable by calling O*bject.freeze(theObject)* on, or defining object in args via *const immutableObj = Object.freeze({//objectDefinition});*

Side Effects
-A change of system state or observable interaction with code outside a function that occurs during the calculation of the function's output. Note the "observable interaction" part (obtaining user input, querying the DOM, accessing system state, etc.).

-To be pure, the function must only exist within itself and not interact with the world outside of it. Functional programming allows some side-effects, but they must be very contained and controlled (via functors, monads, etc.).

Why Pure Functions
-Cacheable: Since same input = same output, can cache function (increased speed), via memoization, where wrap function definition inside other function. Second call with same input will returned cached output, instead of needing to run again:

ex. *squareNum = memoize(x => x * x);* //if call squareNum(5) twice, second call is cached

-Portable & Self-Documenting: Pure functions take in all dependencies as attributes. They are not working on global objects outside the function, and thus you know everything the function is working with. This provides clarity and maintainability, where you don't have to worry about changing something outside the function, then screwing up the program because the function references it without being explicit about its dependency on it.

-Testable: Give it an input, it gives you an output. Bam. Test complete.

-Reasonable: Pure functions have "referential transparency." A pure function can be substituted for it's evaluated value without changing the behavior of the program. Ex. If have X function that produces X data and Y function calls X data, could simply set X function to be inline inside of Y function. (This technique is called "equational reasoning").

-Parallel Code: Since no side effects, no race conditions, so all code can run in parallel without worry.

Closure
-Works as JS uses lexical scoping, meaning you have inner (ex. *let x* is defined inside function) and outer (ex. *let y)* scope, and inner scope will have access to outer scope, but not vice versa (ex. could do *x + y* inside function, but not outside function).
-Note that since JS runs top to bottom, if var is defined below function, function cannot reference it.

-ex.
```
let out = 2;
function sum(num){ return num + out});
sum(4);          //returns 6

//but if was to move let out = 2;   below sum(4);, function would error out as undefined
//likewise, if I was to put
```

-Closure when an inner function makes reference to a variable from the outer function. When a function remembers and accesses variables outside it's own scope, even when that function is executed in a different scope.

-ex. outer function is passed an *xyz* variable. Function is defined within outer and inner function references *xyz* without it being passed to it as an arg.

-Interesting part of closure is that when an inner function gains access to a variable through closure, it remembers that value even after the outer function finishes executing. This is because of lexical scope as defined above.

-Way generally use is to make an outer function(s), that returns an inner function(s), that returns some value. Call to outer function returns an instance of inner function, which has reference to the args passed to outer function. Then, at later time can call inner function via calling value returned from outer function, and pass it some other arg to complete. Good for encapsulation.

-ex.
```
function returnFunction(x){
    return function returnSum(y){
      return x+y;
    }
}

let innerInstance = returnFunction(3);
//innerInstance holds reference to returnSum, which has access to x passed in to returnFunction

console.log(`completing function: ${innerInstance(2)}`);
 //call innerInstance, which calls returnSum(2), which adds 3+2 and returns 5
```

-More on closure: **https://www.youtube.com/watch?v=71AtaJpJHw0**

this

-Since functional programming shoots for pure functions, you shouldn't be using this, as this implies state, which implies impurity.
-Instead of using *this*, create a *const* object or pass an anon object which holds the properties you need, then reference, pass, etc. that const

Delaying Evaluation

-Can make impure function pure by breaking logic down into multiple unary parts, and delaying the call on later parts. Function pure as only one input, with only one output per input, with later called function returned from function handling second input.

-Ex. have function that takes in individual params, but want to pass in an array, can wrap function that takes in individual params in a function that spreads unary params, then passes them in

-ex.
```
function returnFunction(x){
    return function returnSum(y){
            return x+y;
     }
```

```
}

let innerInstanceA = returnFunction(2);
let innerInstanceB = returnFunction(3);
console.log(innerInstanceA(2));          //prints 4
console.log(innerInstanceA(4));          //prints 7
```

Currying

Review: Bind

-Review of *bind()*. when call *.bind()* on an object, whatever pass into bind will be used to set the *this* scope for object. Ex.:

```
let dog = {
   sound: 'woof',
   talk: function() {
      console.log(this.sound);
   }
}

dog.talk();          //prints woof, as talk() has access to sound, as is within dog scope

let talkFunction = dog.talk;
talkFunction();      //undefined as talk from window scope does not have access to sound

talkFunction = talkFunction.bind(dog);
talkFunction();      //'woof', as talkFunction now bound to scope of dog
```

Currying 101

When a function is called with fewer arguments that it expects and it returns a function that takes the remaining arguments. Works due to closer, where returned function has access to variable to arg passed to outer function, even though outer function call finished.

-ex.
```
const add = x =>
    y => x * y;

const increment = add(1);          //returns y => x * y;
const addTen = add(3);
increment(2);                      //returns 3
addTen(2);                         //returns 5
```

-Currying lets you return a function that already has one argument pre-loaded, so when you have later arguments available, you can call the returned function. This is called partial application. Currying lets you make all functions into unary functions, which makes calling them much easier if you only have some arguments, some of the time.

Unary Functions

-If want to force function to only take a single arg, can pass functionX to outer functionY, then pass function to inner functionZ, then call inner function and have it return a call to functionX with single arg allowed to inner function, passed to functionX

ex. *const unary = someFunction =>*
someArg => someFunction (someArg);

Functions that Require Functions
-If using an API that requires a function (ex. *then()* used in promises) but want to pass is a non-function value, consider writing a function that returns a function, that returns the desired value, then pass value into outer function call for arg and pass to *functionThatRequiresFunction().*

-ex. *const valueFunction = someValue =>*
() => someValue;

.....then(constant(value));

Reduction of Arity
-Partial application is when you reduce the arity of a function, by creating another function to handle the processing of some of the original functions args (ex. turn ternary to binary)

-Useful for if have a function where some of the args are available immediately, but others will not be available until later. Can create two functions, to handle both sets of args as they are available, then a third function to combine the args

Coding by Composing

-Composition: passing in multiple functions to a function that returns a function that combines the functions by wrapping them around each other.
-ex. *const compose = (functionA, functionB) => x => functionA(functionB(x));*

-Can compose any number of functions together, including functions composed other other compositions

-An alternative to useing high order functions what that in other functions as args

-The following applies as a mathematical law for all composition:
compose(f, compose(g, h)) === compose(compose(f, g), h);

-Instead of having a function call a function from in side of it { ... someFunctionDefOrCall } use comp

Pointfree
-Functions that never mention the data they are called on
-ex. *const lowercase = word => word.toLowerCase();* //not pointfree since references word
const case = lowercase; //pointfree

-More flexible, since don't need to have data ready to call method on it. Once data is avaiable, pass it to function instead. Keeps code generic.

-Aim for code to be point free whenever possible. Is a good litmus test for if code is functional.

Identity
-A morphism (function) that takes and input and returns exact same output as input, untouched in value. Often used to identify key or key value pairs in objects and data structures.

ex. *const id = v => v;* //const is a function that takes in v and returns v

-example. *Array.identity*() - takes in an array and returns values from array indexes