doodleblue

DIGITAL STRATEGY CONSULTING

Functional programming

By Leo Allen

Date : 21st February 2019

www.doodleblue.com

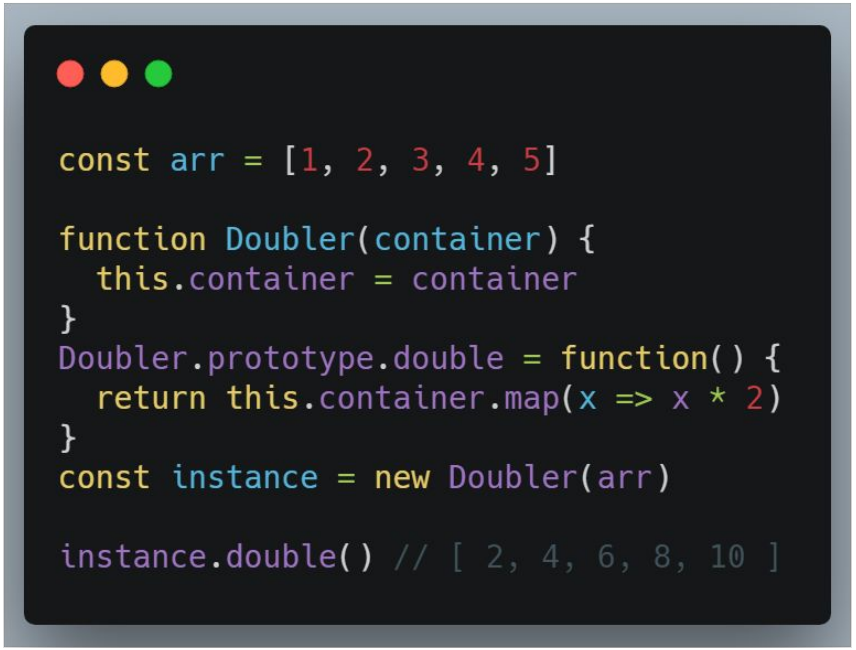
What is functional programming and why should you care

- Functional programming is a programming paradigm (🤓)
- Like object oriented programming, functional programming is simply a philosophy or a style of writing code
- It's way cooler than object oriented programming! (Open for debate) 😎
- Javascript plays really well with functional programming
- Your code becomes easy to reason with
- Easy unit testing

Imperative style

```
const arr = [1, 2, 3, 4, 5]

/* Imperative  */
function double(arr) {
  const result = []
  for (let i = 0; i < arr.lenght; i++) {
    result.push(arr[i] * 2)
  }
  return result
}

double(arr) // [ 2, 4, 6, 8, 10 ]
```

Object oriented style



```
const arr = [1, 2, 3, 4, 5]

function Doubler(container) {
  this.container = container
}
Doubler.prototype.double = function() {
  return this.container.map(x => x * 2)
}
const instance = new Doubler(arr)

instance.double() // [ 2, 4, 6, 8, 10 ]
```

Functional style

```
const arr = [1, 2, 3, 4, 5]

const mapper = fn => collection => collection.map(fn)

const multiplyByTwo = item => item * 2

const doubler = mapper(multiplyByTwo)

doubler(arr) // [ 2, 4, 6, 8, 10 ]
```

The difference

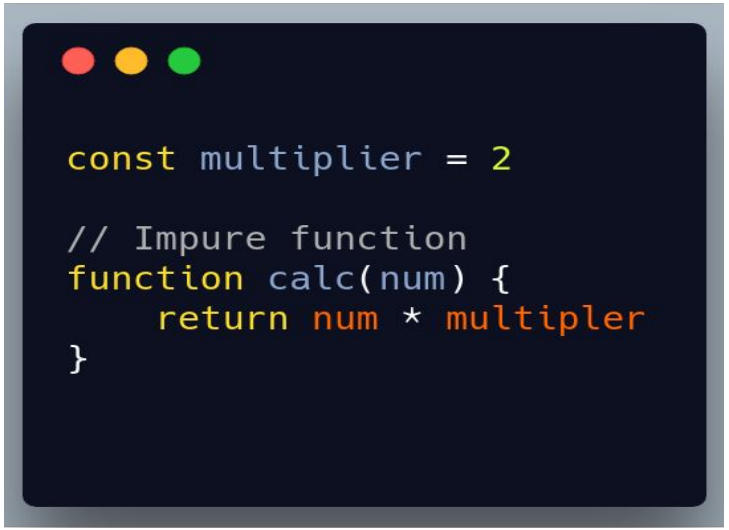
- From the example, we can see that functional programming breaks down a single monolithic function into small reusable functions
- Functions are easier to understand when they’re smaller
- Functional programming promotes separation of concerns. That makes functions easy to unit test

Ground rules

- Pure functions
- Avoiding side effects
- Immutability
- Functions as first class entities (Treating functions like a value that can be passed around)

Purity of a function

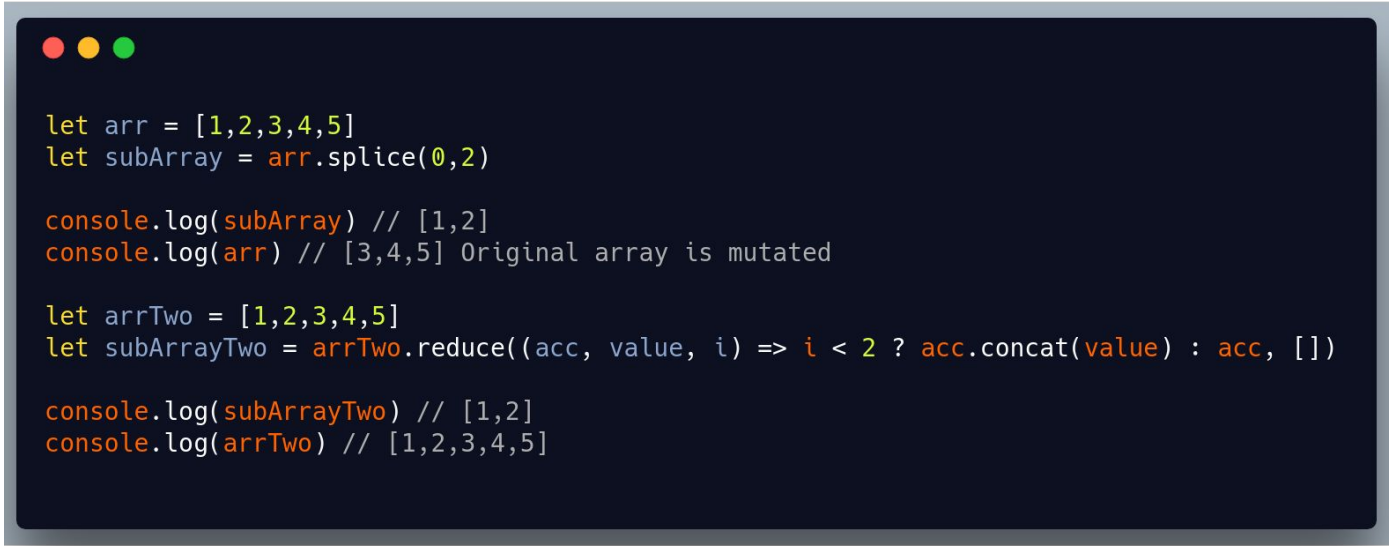
- A function is said to be pure if it's return value is dependant only on the arguments that are being passed to it. That is, for a given set of input, the function always returns the same output value



```
const multiplier = 2

// Impure function
function calc(num) {
    return num * multipler
}
```

Immutability

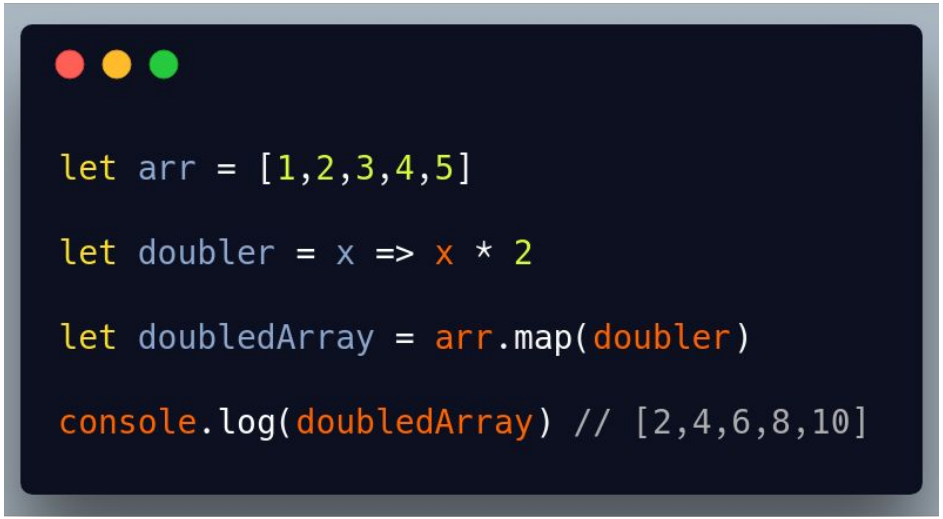
- An operation is said to be immutable if it doesn't modify or mutate any of the operands



```
let arr = [1,2,3,4,5]
let subArray = arr.splice(0,2)

console.log(subArray) // [1,2]
console.log(arr) // [3,4,5] Original array is mutated

let arrTwo = [1,2,3,4,5]
let subArrayTwo = arrTwo.reduce((acc, value, i) => i < 2 ? acc.concat(value) : acc, [])

console.log(subArrayTwo) // [1,2]
console.log(arrTwo) // [1,2,3,4,5]
```

Functions as first class entities

- Javascript lets functions to be passed around like variables. This is very useful in function composition.



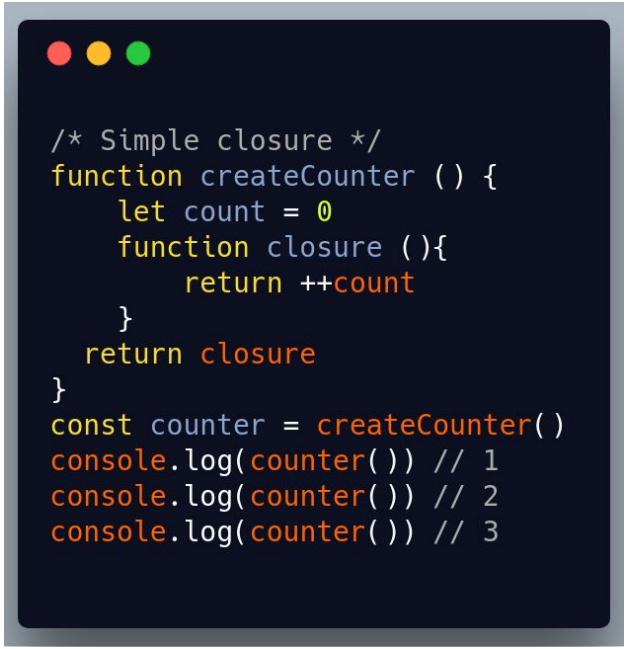
```
let arr = [1,2,3,4,5]

let doubler = x => x * 2

let doubledArray = arr.map(doubler)

console.log(doubledArray) // [2,4,6,8,10]
```

Basic patterns in FP

- Closures
- Factories
- Decorators
- Currying
- Composition

Closures

- When a function returns another function or an objects, the resulting methods/functions inherit the scope of the parent function. Closures are functions with an internal state



```
/* Simple closure */
function createCounter () {
    let count = 0
    function closure (){
        return ++count
    }
  return closure
}
const counter = createCounter()
console.log(counter()) // 1
console.log(counter()) // 2
console.log(counter()) // 3
```

Applications of closures - Memoization

```
function memoize (fn) {
    let cache = {}
    return value => {
      if (cache[value]){
        return `Cache: ${cache[value]}`
      } else {
        cache[value] = fn(value)
        return `Computed: ${cache[value]}`
      }
    }
}
const calcPower = memoize(x => x * x)
console.log(calcPower(3)) // "Computed: 9"
console.log(calcPower(5)) // "Computed: 25"
console.log(calcPower(3)) // "Cache: 9"
```

Factories

- Factories are functions that return an object. Kinda similar to how closures return a function
- The object's methods will contain an internal state which is the scope of the parent function
- Factories are an equivalent to classes in OOPS.
- They have ability to have private variables and states. (ES5 classes don't support that currently)

Example factory

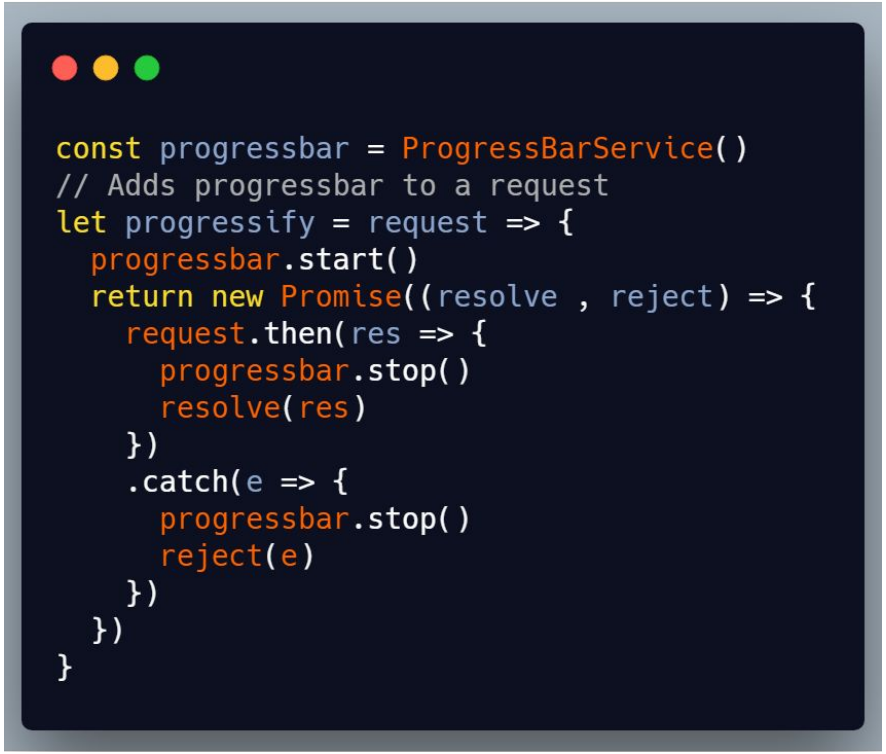
```
function DogFactory (name) {
  let distance = 0;
  const methods = {
    bark : () =>  `${name}: Woof!`,
    run : () => {
      distance++
      return `${name}: is running!`
    },
    distanceCovered : () => `${name} ran ${distance} kms!!`
  }
  return methods
}

let sven = DogFactory ('Sven')
sven.bark() // "Sven: Woof!"
for(let i = 0; i < 50; i++) {
  sven.run()
}
sven.distanceCovered() // "Sven ran 50 kms!!"
```

Decorators

Decorators are functions which take another function as an input and add additional behaviours to it

```
const logToConsole = (value) => console.log(value)

function addTimeStamp(fn) {
  return function(value) {
    value = `${Date.now()}: ${value}`
    fn.call(null, value)
  }
}

let logger = addTimeStamp(logToConsole)
logger('Test log') // "1552329282795: Test log"
```

Promise decorators



```
const progressbar = ProgressBarService()
// Adds progressbar to a request
let progressify = request => {
  progressbar.start()
  return new Promise((resolve , reject) => {
    request.then(res => {
      progressbar.stop()
      resolve(res)
    })
    .catch(e => {
      progressbar.stop()
      reject(e)
    })
  })
}
```

Currying

- Breaking down function calls with multiple arguments into chains of function calls with a single argument for each call

```
const dogInfo = name => type => action => `${name} is a ${type} which ${action}`

console.log(dogInfo('Sven')('dog')('barks')) // Sven is a dog which barks
```

Currying real life example

```
const chalk = require('chalk')

function logs() {
  const methods = {
    print: type => (...msg) =>
      process.env.NODE_ENV === 'dev' ? console.log(chalk[type](...msg)) : undefined,
    log: (...msg) => methods.print('green')(...msg),
    info: (...msg) => methods.print('blue')(...msg),
    error: (...msg) => methods.print('red')(...msg)
  }

  return Object.freeze(methods)
}

module.exports = logs()
```

Composition

- Composition is a cool trick which lets you combine two functions or more functions into one
- It works best with functions with a single arity
- Consider, two functions f and g
- Then, h(x) = f(g(x)) (If that made sense )

The compose function

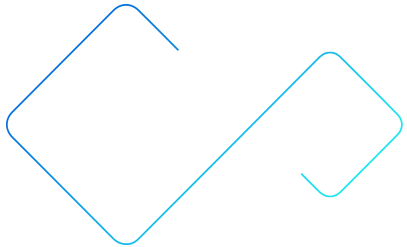
```
const compose = (...fns) => value => fns.reduce((acc, next) => next(acc), value)

const square = x => x * x

const addOne = x => x + 1

const squareAndAddOne = compose(square , addOne)

[1,2,3].map(squareAndAddOne) // [2, 5, 10]
```

Functional programming libraries

- Check out https://ramdajs.com/
- RAMDA is a functional programming's equivalent to lodash
- Ramda functions are automatically curried. This allows you to easily build up new functions from old ones simply by not supplying the final parameters.

Auto-curried map example in Ramda

```
const R = require('ramda')

const grabId = R.map(x => x._id)

const data = [{ _id: '514313123', name: 'Leo' }, { _id: '5123123123', name: 'Allen' }]

grabId(data) // ['514313123' , '5123123123']
```



Thank you