CODESMITH

Software Engineering Residency

Cohort 10 January 2017

Will Sentance

Academic work:
Oxford, Harvard

Currently:
• CEO & Cofounder Codesmith
• Frontend Masters

Previously:
• Cocreator & Engineer @Icecomm
• Software Engineer @Gem

Functional programming - a paradigm for structuring our complex code

Let’s suppose we have a quiz game we’re building

Every line of code either saves data to memory (e.g. a user’s score) or uses that data (e.g. increase that user’s score)

But with 1000s of lines of code - every line of code can potentially use (and **depend**) on that data 🥴

High risk - every time I want to change any line, I have to consider whether it could affect other lines (and when I miss something, I get bugs)

What’s the answer?

The answer has been *functions*

Compartmentalise - Reduce the potential impact of any given line to maybe 10 other lines (inside the function)

But even within a 10 line function, ‘reasoning’ through what each line does is hard - & the code may still have affects outside the function (via global variables)

Functions as we know them may not be enough!

Imagine if we could structure our code into individual pieces where *almost every single line* is self-contained

— The only thing any given line depends on are inputs (explicitly stated in that very line)

— The only consequences of the line would be its output (explicitly stated in that very line)

— *And* each line could get a nice human-readable label for when we use it!

This could transform how we write code, debug it and read about others' code.

But how could we do it!? 😲

Functional programming

Tiny functions: Save every single line (or few lines) as its own function

No consequences except on that line Each function's only 'consequence' is to have its result given to specifically the next line of code ('function call') and not to any other lines

Recombine/compose Build up our application by using these small blocks of self-contained code combining them up line-by-line by *referring to their human-readable name*

We could produce a beautiful ‘to do list’ of our code

```
pipe(
  getPlayerName,
  getFirstName,
  properCase,
  addUserLabel,
  createUserTemplate
)([{name: 'will sentance', score: 3}]);
```

And render to the webpage

Functional programming techniques - How do we recombine?

Combining our functions is going to require a ton of interesting techniques to:

— rejoin these ‘lines’ of code (tiny functions) into full-sized tasks:
— make it easy to reuse these functions all over the place
— ensure that the tiny functions truly are self-contained

We’re going to see many of these techniques today

(1) Higher order functions, (2) Function composition

- A lot these atoms of code (tiny functions) will be reusable
  - They’re small enough that they’re tasks like incrementing a number, looping through an array
  - We want to write once, use again and again - even for tasks that are not quite identical - keeping our code DRY

(3) Pure functions, Immutability of state

We cannot have our lines of code rely on any external data except their explicitly stated inputs

It’s especially important when you’re reusing/ recombining lots of these little single-step functions in lots of totally different scenarios - they better be self contained!

(4) Closure, (5) Function Decoration (6) Partial application & Currying

We will need ways to:

- Adjust the functions in case they don't quite fit together as initially saved
- Give our functions extra features without having to write a new function from scratch.

If we can do all that, our code will become:

- **More readable** - every line of code has a name that indicates its goal known as ’declarative’ programming
- **Easier to debug** - Each line of code is an individual unit with clear input and output - no unexpected consequences of using that line (‘function’)
- **Easier to add features** - most new things we want do do are combinations of something we’ve done elsewhere in our app

Our end and beginning

Functions become reusable, versatile, flexible piece of code - a series of independent self-contained readable and predictable steps passing data from one to the next

But it all starts with us being confident with the core principles of JavaScript

JavaScript the Hard Parts - Functional Programming

1. **Principles of JavaScript**
2. Higher order functions
3. Arrow and anonymous functions
4. `reduce`, `filter` and chaining higher order functions
5. Function composition & pure functions
6. Closure
7. Function decoration
8. Partial application and currying

What happens when javascript executes (runs) my code?

```
const num = 3;
const multiplyBy2 = (inputNumber) => {
  const result = inputNumber*2;
  return result;
}
const name = "Will"
```

Code is saved (defined) in functions - to be run later

As soon as we start running our code, we create a *global execution context*

— Thread of execution (parsing and executing the code line after line)

— Live memory of variables with data (known as a Global Variable Environment)

Running/calling/invoking a function

This is not the same as defining a function

```
const num = 3;
const multiplyBy2 = (inputNumber) => {
  const result = inputNumber*2;
  return result;
}

const output = multiplyBy2(num);
const newOutput = multiplyBy2(10);
```

When you execute a function you create a new execution context comprising:

1. The thread of execution (we go through the code **in the function** line by line)
2. A local memory ('Variable environment') where anything defined in the function is stored

We keep track of the functions being called in JavaScript with a Call stack

Tracks which execution context we are in - that is, what function is currently being run and where to return to after an execution context is popped off the stack

One global execution context, multiple function contexts

JavaScript the Hard Parts - Functional Programming

1. Principles of JavaScript
2. **Higher order functions**
3. Arrow and anonymous functions
4. `reduce`, `filter` and chaining higher order functions
5. Function composition & pure functions
6. Closure
7. Function decoration
8. Partial application and currying

Suppose we have a function copyArrayAndMultiplyBy2. Let's diagram it out

```
const copyArrayAndMultiplyBy2 = (array) => {
  const output = [];
  for (let i = 0; i < array.length; i++) {
    output.push(array[i] * 2);
  }
  return output;
}
const myArray = [1,2,3]
const result = copyArrayAndMultiplyBy2(myArray)
```

What if want to copy array and divide by 2?

```
const copyArrayAndDivideBy2 = (array) => {
  const output = [];
  for (let i = 0; i < array.length; i++) {
    output.push(array[i] /2);
  }
  return output;
}
const myArray = [1,2,3]
const result = copyArrayAndDivideBy2(myArray);
```

Or add 3?

```
const copyArrayAndAdd3 = (array) => {
  const output = [];
  for (let i = 0; i < array.length; i++) {
    output.push(array[i] +3);
  }
  return output;
}
const myArray = [1,2,3]
const result = copyArrayAndAdd3(myArray);
```

What principle are we breaking?

We're breaking DRY

What could we do?

We could generalize our function so that we pass in our specific instruction only when we run the copyArrayAndManipulate function!

```
const copyArrayAndManipulate = (array, instructions) => {
  const output = [];
  for (let i = 0; i < array.length; i++) {
    output.push(instructions(array[i]));
  }
  return output;
}


const multiplyBy2 = (input) => {
  return input * 2;
}


const result = copyArrayAndManipulate([1, 2, 3], multiplyBy2);
```

How was this possible?

Functions in javascript = first class objects

They can co-exist with and can be treated like any other javascript object

1. assigned to variables and properties of other objects
2. passed as arguments into functions
3. returned as values from functions

Callback vs. Higher-order function

```
const copyArrayAndManipulate = (array, instructions) => {
  const output = [];
  for (let i = 0; i < array.length; i++) {
    output.push(instructions(array[i]));
  }
  return output;
}

const multiplyBy2 (input) => {
  return input * 2;
}

const result = copyArrayAndManipulate([1, 2, 3], multiplyBy2);
```

Which is our callback function? Which is our higher order function?

Callback vs. Higher-order function

```
const copyArrayAndManipulate = (array, instructions) => {
  const output = [];
  for (let i = 0; i < array.length; i++) {
    output.push(instructions(array[i]));
  }
  return output;
}

const multiplyBy2 (input) => {
  return input * 2;
}

const result = copyArrayAndManipulate([1, 2, 3], multiplyBy2);
```

The function we *pass in* is a callback function. The outer function that *takes in* the function (our callback) is a higher-order function

Higher-order functions

Takes in a function or passes out a function

Just a term to describe these functions - any function that does it we call that - but there's nothing different about them inherently

Higher order functions

- **Easier to add features** - we don’t need to build a brand new `copyArrayAndAdd3` function - just use `copyArrayManipulate` with the input of `add3`. Higher order functions keep our code DRY
- **More readable** - copyArrayManipulate(multiplyBy2) - I know what this is doing more readily than the for loop style
- **Easier to debug** - As long as we understand what’s happening under-the-hood

Pair-programming

JavaScript the Hard Parts - Functional Programming

1. Principles of JavaScript
2. Higher order functions
3. **Arrow and anonymous functions**
4. `reduce`, `filter` and chaining higher order functions
5. Function composition & pure functions
6. Closure
7. Function decoration
8. Partial application and currying

Arrow functions in ES2015

```
const multiplyBy2 = (input) => { return input*2 }
```

So where the function is a single expression to evaluate and then return, ES2015 lets us remove the {} and return keyword

```
const multiplyBy2 = (input) => input*2

const output = multiplyBy2(3) //6
```

We can even remove the parenthesis if there's only 1 parameter (expected input)

```
const multiplyBy2 = input => input*2

const output = multiplyBy2(3) //6
```

Arrow functions fit our our functional programming goals

Every line is an independent, labelled piece of code where we know exactly what data it uses and affects

Therefore a lot of our functions are going to be just

1. Take Input
2. Use the input in some way
3. Return that as the output in the same line.

Arrow functions let us condense our functions to show this

Let’s review our use of copyArrayAndManipulate

```
const copyArrayAndManipulate = (array, instructions) => {
    const output = [];
    for (let i = 0; i < array.length; i++) {
      output.push(instructions(array[i]));
    }
    return output;
}

const multiplyBy2 = (input) => {
        return input*2
}

const result = copyArrayAndManipulate([1, 2, 3], multiplyBy2);
```

Let’s use our shortened version of our callback function `multiplyBy2`

```
const copyArrayAndManipulate = (array, instructions) => {
    const output = [];
    for (let i = 0; i < array.length; i++) {
      output.push(instructions(array[i]));
    }
    return output;
}

const multiplyBy2 = input => input*2

const result = copyArrayAndManipulate([1, 2, 3], multiplyBy2);
```

We can even insert our callback function ‘multiplyBy2’ directly (‘anonymously’)

```
const copyArrayAndManipulate = (array, instructions) => {
    const output = [];
    for (let i = 0; i < array.length; i++) {
      output.push(instructions(array[i]));
    }
    return output;
}

const result = copyArrayAndManipulate([1, 2, 3], input => input*2);

//No multiplyBy2 function independently declared/saved
```

Developers tend to call copyArrayAndManipulate, map

```
const map = (array, instructions) => {
    const output = [];
    for (let i = 0; i < array.length; i++) {
      output.push(instructions(array[i]));
    }
    return output;
}

const multiplyBy2 = input => input*2

const result = map([1, 2, 3], multiplyBy2);
```

Let’s diagram it in a different way and talk about the purpose of diagramming

JavaScript the Hard Parts - Functional Programming

1. Principles of JavaScript
2. Higher order functions
3. Arrow and anonymous functions
4. **`reduce`, `filter` and chaining higher order functions**
5. Function composition & pure functions
6. Closure
7. Function decoration
8. Partial application and currying

reduce and reducers

- *The* most versatile higher order function of all
- Takes a mental shift to look at problems through the reduce lens
- Can even enable function composition (to come)

We have already seen ‘reduction’ in action

```
const map = (array, instructions) => {
    const output = [];
    for (let i = 0; i < array.length; i++) {
      output.push(instructions(array[i]));
    }
    return output;
}

const multiplyBy2 = input => input*2

const result = map([1, 2, 3], multiplyBy2);
```

What is map actually doing? (accumulator, push etc)

We combined/reduced by:

- Taking the array [] and combine with array[0] by pushing, take that combined value and combine with array[1] by pushing and so forth

How else could we combine and use, combine and use?

How else could we ‘combine and use, combine and use’?

- Take the number 0 and combine with array[0] by adding, take that combined value and combine with array[1] by adding and so forth…
- Take the empty string “ ” and combine with array[0] by appending, take that combined value and combine with array[1] by appending and so forth…

We’d want to write our function so that it could handle:

- Any ‘accumulator’ (array, string, number)
- Any combining logic/code/functionality (the ‘reducer’)

This function is known as `reduce` - it can handle a remarkable range of tasks

```
const reduce = (array, howToCombine, buildingUp) => {
    for (let i = 0; i < array.length; i++){
        buildingUp = howToCombine(buildingUp, array[i])
    }
    return buildingUp
}


const add = (a, b) => a + b

const summed = reduce([1,2,3], add, 0)
```

Should be called 'reduce from 2 things to 1 repeatedly, inside'

Interlude - arrays, objects and functions have access to 'methods'

- 'methods' are functions that they can use on themselves

```
const array = [1,2,3]
array.push(10) // Where's this push method come from?
```

- The 'methods' that arrays get are stored in an object that every array has access to when you refer to the array, followed by a 'dot' and the method name (See OOP JS Hard Parts)
- The link to this object full of shared methods is on the `__proto__` property

JavaScript has a built-in version of `reduce` available to all arrays

Reminder of our version of reduce

```
const add = (a, b) => a + b

const summed = reduce([1,2,3], add, 0) // summed is 6
```

With the built-in version , the array is inserted into the `reduce` function automatically as the first input/argument. But they produce the same result

```
const add = (a, b) => a + b

const summed = [1,2,3].reduce(add, 0) // summed is 6
```

In fact we get a number of built-in higher order functions in JavaScript

— map, forEach, filter, flatMap, reduce, reduceRight

— All ‘iterate’ through each element of the array and run a function on each

— But behave differently - MDN guides us

```
const array = [1,2,3,4,5,6]
const greaterThan2 = num => num > 2

const filteredArray = array.filter(greaterThan2) // [3,4,5,6]
```

And we can ‘chain’ these higher order functions - pass the output of one as the input of the next

```
const array = [1,2,3,4,5,6]
const greaterThan2 = num => num > 2
const add = (a, b) => a + b

const sumOfGreaterThan2 = array.filter(greaterThan2).reduce(add,0)
```

— The output of each higher order function (HOF), where it’s an array, has access to all the HOFs (map, filter, reduce) through the prototype chain

`reduce`, `filter` and chaining higher order functions

- **Easier to add features** - we can reuse `filter` in a 1000 different filtering scenarios and can chain it up with other functions
- **More readable** - `array.filter(greaterThan2).reduce(add,0)` - more readable than individual steps with explicit loop
- **Easier to debug** - As long as we understand what’s happening under-the-hood

And `reduce` is going to enable something even more powerful

Pair programming

JavaScript the Hard Parts - Functional Programming

1. Principles of JavaScript
2. Higher order functions
3. Arrow and anonymous functions
4. `reduce`, `filter` and chaining higher order functions
5. **Function composition & pure functions**
6. Closure
7. Function decoration
8. Partial application and currying

Function composition

- Chaining with dots relies on JavaScript prototype feature - functions return arrays which have access to all the HOFs (map, filter, reduce)
- I’m passing my output into the next function automatically
- What if I want to chain functions that just return a regular output
- e.g. `multiplyBy2`, `add3`, `divideBy5`

We could keep track with global variables

```
const multiplyBy2 = x => x*2
const add3 = x => x+3
const divideBy5 = x => x/5

const initialResult = multiplyBy2(11)
const nextStep = add3(initialResult)
const finalStep = divideBy5(nextStep)

console.log("finalStep", finalStep)
```

But that’s risky, people can overwrite

Or we can use the fact that JavaScript evaluates every function call before it moves on

```
const multiplyBy2 = x => x*2
const add3 = x => x+3
const divideBy5 = x => x/5

const result = divideBy5(add3(multiplyBy2(11)))
```

Now this is pretty unreadable though 😕

(Btw This relies on our functions being 'referentially transparent' - we can replace the call to the function with its return value with no consequences on our app)

We’re combining a function with a value to get a result then combining that result with another function to get another result and so on

What’s this remind you of?

Reduce as *the* most versatile function in programming

```
const multiplyBy2 = x => x*2
const add3 = x => x+3
const divideBy5 = x => x/5

const reduce = (array, howToCombine, buildingUp) => {
    for (let i = 0; i < array.length; i++){
        buildingUp = howToCombine(buildingUp, array[i])
    }
    return buildingUp
}

const runFunctionOnInput = (input,fn) => {
    return fn(input)
}

const output = reduce([multiplyBy2, add3, divideBy5], runFunctionOnInput, 11)
```

Listing out our ‘lines of code’ (functions) by name with each one’s consequence limited to only affect the next ‘line’ (function call/invocation)

```
const multiplyBy2 = x => x*2
const add3 = x => x+3
const divideBy5 = x => x/5
const subtract4 = x => x-4

const reduce = (array, howToCombine, buildingUp) => {
    for (let i = 0; i < array.length; i++){
        buildingUp = howToCombine(buildingUp, array[i])
    }
    return buildingUp
}

const runFunctionOnInput = (input,fn) => { return fn(input) }

const output = reduce([
        multiplyBy2,
        add3,
        divideBy5,
        subtract4
    ],
    runFunctionOnInput, 11
)
```

Function composition

- **Easier to add features** - This is the *essential* aspect of functional javascript - being able to list of our units of code by name and have them run one by one as independent, self-contained pieces
- **More readable** - `reduce` here is often wrapped in `compose` to say ‘combine up’ the functions to run our data through them one by one. The style is ‘point free’
- **Easier to debug** - I know exactly the line of code my bug is in - it’s got a label!

Pure functions

- Functions as tiny units to be combined and run automatically **must** be highly predictable
- We rely on using their evaluated result to pass the input to the next unit of code (automatically). Any ‘side effects’ would destroy this

```
let num = 10

const add3 = x => {
    num++
    return x+3
}

add3(7)
```

Immutability

```
const array = [1,2,3]
const multiplyBy2 = num => num*2

const result = array.map(multiplyBy2) // [2,4,6]
const newResult = array.map(multiplyBy2) // [2,4,6]
```

If we want the only consequence of map to be on that line and to achieve ‘referential transparency’ (I can the function call with its output and it’s the same) - then I need to preserve my data and not manipulate it

JavaScript passes a reference (‘link back’) to the `arr` when it’s inserted into the function `map`. If we change (‘mutate’) the input array our function is not pure - it’s unpredictable - I can’t figure out what it does just be reading it and looking at its output there in that line - undoes all our hard work

Pure functions & immutability

- **Easier to add features** - Every saved function be safely used in new combinations , confident it won’t break other parts of the app
- **More readable** - Every line is ‘complete’ - it’s fully descriptive - exactly what it does is discoverable its name and limited to that input/output
- **Easier to debug** - No 1000s of lines of interdependence

Pair programming

JavaScript the Hard Parts - Functional Programming

1. Principles of JavaScript
2. Higher order functions
3. Arrow and anonymous functions
4. `reduce`, `filter` and chaining higher order functions
5. Function composition & pure functions
6. **Closure**
7. Function decoration
8. Partial application and currying

Closure

- Most esoteric concept in JavaScript
- Functions are our units to build with but they’re limited - they forget everything each time they finish running - with no global state
- Imagine if we could give our functions memories

Reminding ourselves of how functions actually work

```
const multiplyBy2 = inputNumber => inputNumber*2;

const output = multiplyBy2(7);
const newOutput = multiplyBy2(3);
```

No memory of the previous execution - imagine if we could give our functions permanent memories

It begins with returning a function from a function

Let's call (run) our generated function with the input 3

```
const functionCreator = () => {
    let counter = 0
    const add3 = (num) => {
        const result = num+3
        return result
    }
    return add3
}

const generatedFunc = functionCreator()

const result = generatedFunc(2) //5
```

Calling a function inside the function call in which it was defined

```
const outer = () => {
  let counter = 0;
  const incrementCounter = () => {
    counter ++;
  }
  incrementCounter();
}

outer();
```

What determines what data your function have access to when you call the function? Where we call it?

But what if we call our function outside of where it was defined?

```
const outer = () => {
  let counter = 0;
  const incrementCounter = () => {
    counter ++;
  }
}

outer();

incrementCounter();
```

What happens here?

There is a way to run a function outside where it was defined

Without an error - we return the inner function and assign it to a new global label

```
const outer = () => {
    let counter = 0;
    const incrementCounter = () => {
      counter ++;
    }
    return incrementCounter
}

const newFunction = outer();
```

Now we can call the function that was originally saved as `incrementCounter` by its new global label `newFunction`

```
const outer = () => {
    let counter = 0;
    const incrementCounter = () => {
      counter ++;
    }
    return incrementCounter
}

const newFunction = outer();
newFunction()
newFunction()
```

But we have a problem 😭

The bond

When a function is defined, it gets a bond to the surrounding Local Memory (“Variable Environment”) in which it has been defined

```
const outer = () => {
    let counter = 0;
    const incrementCounter = () => {
      counter ++;
    }
    return incrementCounter
}

const newFunction = outer();
newFunction()
newFunction()
```

The ‘Backpack’

1. When `incrementCounter` is defined, it gets a bond to the surrounding Local Memory of live data in `outer` in which it has been defined
2. We then return `incrementCounter` out of `outer` into global and store it in `myNewFunction`
3. BUT we maintain the bond to the surrounding live local memory from inside of `outer` - this live memory gets ‘returned out’ attached to the `incrementCounter` function definition and is therefore now stored attached to `myNewFunction` - even though `outer`’s execution context is long gone
4. When we run `myNewFunction` in the global execution context, it will first look in its own local memory for any data it needs (as we’d expect), but **then in its ‘backpack’** before it looks in global memory

What’s the official name for the ‘backpack’?

The Closed over Variable Environment (COVE) or ‘Closure’

This ‘backpack’ of live data that gets returned out with `incrementCounter` is known as the ‘closure’

The ‘backpack’ (or ‘closure’) of live data is attached `incrementCounter` (then to `myNewFunction`) through a hidden property known as `[[scope]]` which persists when the inner function is returned out

Closure in functional JavaScript

- **Easier to add features** - Our functions can now have persistent permanent memories attached to them - it's going to let us build dramatically more powerful functions
- **Easier to debug** - Definitely need to know how it's working under the hood!

Pair programming

JavaScript the Hard Parts - Functional Programming

1. Principles of JavaScript
2. Higher order functions
3. Arrow and anonymous functions
4. `reduce`, `filter` and chaining higher order functions
5. Function composition & pure functions
6. Closure
7. **Function decoration**
8. Partial application and currying

Functional Decoration

- Now we can convert functions more easily to make them suit our task
- Without writing a new function from scratch
- We can run code *on other bits of code* to appear to change them

To add a permanent memory to an existing function we have to create a new function that will run the existing function inside of itself

```
const oncify = (convertMe) => {
    let counter = 0
    const inner = (input) => {
        if (counter === 0){
            const output = convertMe(input)
            counter++
            return output
        }
        return "Sorry"
    }
    return inner
}

const multiplyBy2 = num => num*2

const oncifiedMultiplyBy2 = oncify(multiplyBy2)

oncifiedMultiplyBy2(10) // 20
oncifiedMultiplyBy2(7) // Sorry
```

Function decoration

- **Easier to add features** - We can ’pseudo’ edit our functions that we’ve already made - into functions that behave similar but with bonus features!
- **Easier to debug** - Definitely need to know how it’s working under the hood!

Pair programming

JavaScript the Hard Parts - Functional Programming

1. Principles of JavaScript
2. Higher order functions
3. Arrow and anonymous functions
4. `reduce`, `filter` and chaining higher order functions
5. Function composition & pure functions
6. Closure
7. Function decoration
8. **Partial application and currying**

Function composition is powerful but every function needs to behave the same way

- Taking in one input and returning out one output
- What if I have a function I want to use that expects two inputs
  - This is ‘arity mismatch’
- We need to ‘decorate’ our function to prefill *one* of its inputs

This means creating a new function that calls our multi-argument function - with he argument and the multi-argument function stored conveniently in the backpack

It’s known as ‘Partial application’

```
const multiply = (a, b) => a * b

function prefillFunction (fn, prefilledValue){
    const inner = (liveInput) => {
        const output = fn(liveInput, prefilledValue)
        return output
    }
    return inner
}

const multiplyBy2 = prefillFunction(multiply, 2)

const result = multiplyBy2(5)
```

Partial application and currying

- In practice we may have to prefill one, two… multiple arguments at different times
- We can convert (‘decorate’) any function to a function that will accept arguments one by one and only run the function in full once it has all the arguments
- This is a more general version of partial application

Partial application & currying

- **Easier to add features** - Mismatched arity - no problem! We write a function multiply *once* and then reuse it for different situations by ‘editing’ its arguments
- **More readable** - We can use our composition/reduce to list out functions to run one-by-one on our data, even if the functions excepted more than 1 input!
- **Easier to debug** - Individual units of functionality possible even with 1+ input expected

Functional programming

Every line of code is named (or if not, its so short we can see exactly what it does), is an independent unit that has all of its consequences in that single line

We can then couple up (compose) these single units of code/instructions (functions) up into complex tasks

But with every component of the task independent, recognizable, reusable, versatile and easily debuggable!

We have to do some feats to wrestle our tiny units of code (functions)

Combining up functions with multiple inputs from libraries

We’ve seen many of them (higher order functions, reduction/composition, closure, function decoration, partial application and currying). And there are even more - monads, applicators et al!

Readable, debuggable and easy to add features

```
pipe(
  getPlayerName,
  getFirstName,
  properCase,
  addUserLabel,
  createUserTemplate
)([{name: 'will sentance', score: 3}]);
```

But our code is now a set of independent, self contained steps we can wield to solve any problem and become true composers of our code.

Fin