Functional Programming

In JavaScript

{with: categories}

Gain advanced understanding of the mathematics behind modern functional programming

JS

dimitris papadimitriou

v 2.0.0

FUNCTIONAL PROGRAMMING IN JS WITH CATEGORIES

Gain advanced understanding of the mathematics
behind modern functional programming

Version : 2.0.0 last updated 10-Oct-2020

Dimitris Papadimitriou

Feel free to contact me at:

https://www.linkedin.com/in/dimitrispapadimitriou/

https://leanpub.com/u/dimitrispapadim

https://medium.com/@dimpapadim3

https://github.com/dimitris-papadimitriou-chr

dimitrispapadim@live.com

Contents

Purpose

The tools we use have a profound (and devious!) influence on
our thinking habits, and, therefore, on our thinking abilities.
— Edsger Dijkstra

OO makes code understandable by encapsulating moving parts.
FP makes code understandable by minimizing moving parts.
—Michael Feathers (Twitter)

One of the main reasons for this book is to transfer in the community of object-oriented developers some of the ideas and advancements happening to the functional side. This book wants to be pragmatic. Unfortunately, some great ideas are coming from academia that cannot be used by the average developer in his day to day coding.

This is not an Introductory book to functional programming, even if I restate some of the most basic concepts in the light of category theory and the object-oriented paradigm. This book covers the middle ground. Nonetheless, it is written in a way that if someone pays attention and goes through the examples, he or she could understand the concepts. An introductory book will follow in Leanpub, covering more basic ideas, along with the extended version of this book covering more advanced topics. If you think the content of this book is more difficult than what you expected, please contact me to give you a free copy of the book "The Simplified Functional Programming in JavaScript, with categories" when it is finished in Leanpub.

In this book, I will try to simplify the mathematical concepts in a way that can be displayed with code. If something cannot be easily displayed with code probably will not be something that can be readily available to a developer's arsenal of techniques and patterns.

If you think a section is boring, then skip it and maybe finish it later.

One thing I admire in Software Engineers is their ability to infer things. The ability to connect the dots is what separates the exceptional developer from the good one. In some parts of the book, I might indeed have gaps or even assume things that you are not familiar with. Nonetheless, I am sure that even an inexperienced developer will connect the dots and assign meaning, as I usually do when left with unfinished

About this book coding conventions

Most of the code in this book use vanilla JavaScript. Instead of the standard way to create a class:

```
var Rectangle = function (height, width) {
    this.height = height;
    this.width = width;
    this.area = function () {
        return this.height * this.width;
    }
}
```

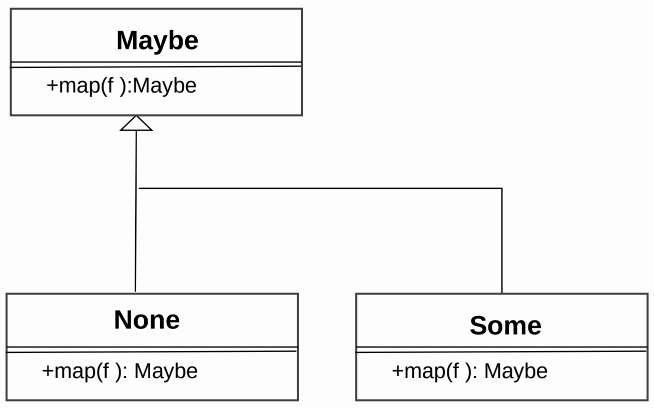
Most of the times I will use this brief notation:

```
const Rectangle = (height, width) => ({
    area: () => height * width,
    scale: (s) => Rectangle (s *  height, s * width),
});
```

Object literal notation forces immutability. In this way of writing, you must always return a new instance. In a larger Domain Model, this way of writing might not be viable. Here we do not need to use the `new` keyword to create an object. Also, we cannot access the height and width variables. If you want to mutate the state, you either must question your design of the object that forces you to think in a defensive programming style. Let's be pragmatic here, this style of coding that promotes purity should **not be an end in itself**, and we should always be able to convince ourselves to go back into an object-oriented style of coding even if it is not that pure. The important thing is to write **functional code that provides business value** to the organization and to the end-user that will eventually use it. We must be pragmatic and utilitarian when we code and abstractionists when we think about code.

Type Safety

Well. In this book, we will use minimal dependencies; this means that there is an orientation to vanilla JavaScript. And **let me tell you, there is no type safety in vanilla JavaScript**. Type safety became so prevalent because the benefits are exciding all the drawbacks and overheads. The current popularity of Typescript is not random. In Some difficult sections there would be some brief TypeScript paragraphs to allow a quick pick at the underlying Types. There is also a translation of this book in TypeScript on Leanpub. **The importance of paying attention to types in a dynamic type system becomes even more important for the developer**. That's why type theory is an essential part of this book. Duck typing is used throughout the book. What is duck typing? We use the duck test–"If it walks like a duck and it quacks like a duck, then it must be a duck"—to determine if an object can be used for a particular purpose. When we have hierarchies like the one below, for example, we will be using duck typing to implement it



```
var some = (v) => ({
    map: (f) => Some(f(v))
});

var none = () => ({
    map: (f) => none()
});
```

Run This: Js Fiddle

This way of writing is for explanatory purposes. In production code or code that belongs to libraries and application framework etc. proper inheritance might be considered, depending on the scope of the model

```
class Maybe {
  map(f) {
    throw new Error('You have to implement the method map!');
  }
}

class Some extends Maybe {
  constructor(v) {
    super();
    this.v = v;
  }
  map(f) {
    return new Some(f(this.v));
  }
}

class None extends Maybe {
  map(f) {
    return new None();
  }
}
```

Run This: Js Fiddle

We will also be using JavaScript mixins, to add functionality without involving inheritance (This became very famous with Scala traits).

Fantasy land

Fasntasy land is a set of specifications for JavaScript functional structures. Some functional libraries out there are following the naming conventions and structure of the fantasy land specifications. In this book, we will adopt fantasy-land conventions loosely.

Functors

«It should be observed first that the whole concept of a category is essentially an auxiliary one; Our basic concepts are essentially those of a functor and of a natural transformation."
S. Eilenberg and S. MacLane »

The Idea:

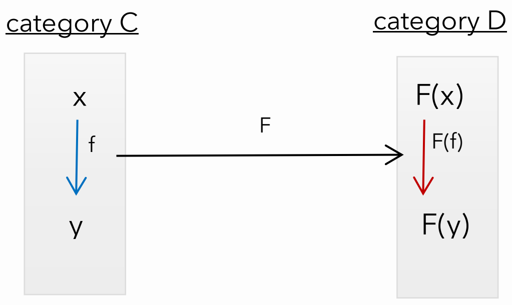
In JavaScript the most famous functional programming idea is to use Array.`map` to replace iterations instead of *for loops* in order to transform the values of the array. That is because an array is a Functor, which is a more abstract idea that we will explore in this section.

"Practically a Functor is anything that has a valid `.map(f)` method"

Functors can be considered the core concept of category theory.

1 Functors

In mathematics, a **functor** is a map between categories.



This Functor F must map two requirements

1. map each object x in *C* with an object F(x) in *D*,
2. map each morphism f in *C* with a morphism F(f) in *D*

For object-oriented programming, the best metaphor for functors is a ***container***, together with a **mapping** function. The Array as a **data structure** is a Functor, ***together*** with the **map** method. The **map** is the array method that transforms the items of the array by applying a function f.

1.1 The Identity Functor

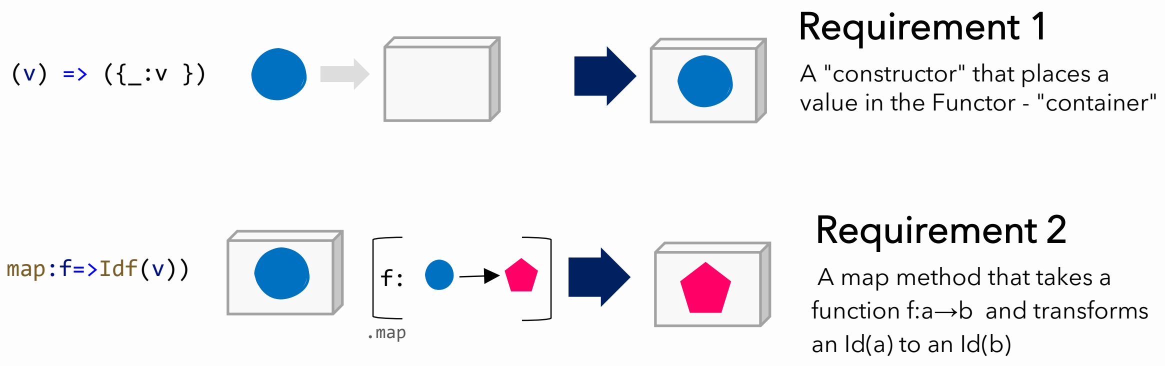
We will start by looking at the minimum structure that qualifies as a functor in JavaScript:

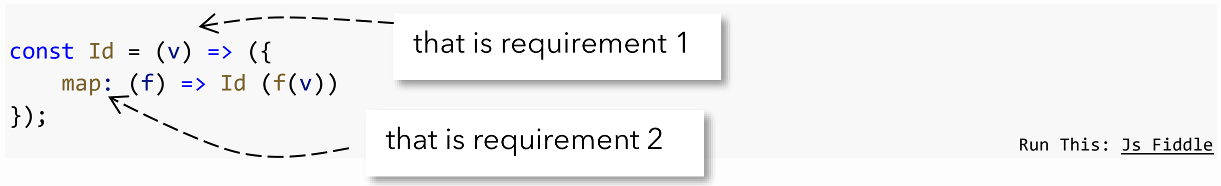
```
const Id = (v) => ({
    map: (f) => Id (f(v))
});
```

Run This: Js Fiddle

This is the minimum construction that we could call a functor because it has exactly two things :

1. A "**constructor**" that maps a value v to an object literal `Id = (v) => {value: v}`
2. and it has a **mapping** method `map:(f) => {_}` that lifts functions f


(v) => ({_:v })
Requirement 1
A "constructor" that places a value in the Functor - "container"
map:f=>Idf(v))
f:
.map
Requirement 2
A map method that takes a function f:a→b and transforms an Id(a) to an Id(b)



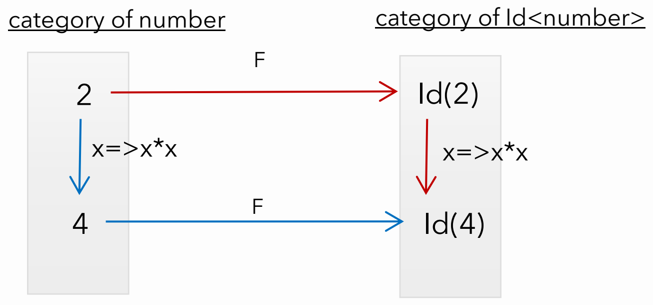
```
const Id = (v) => ({
    map: (f) => Id (f(v))
});
```

that is requirement 1

that is requirement 2

Run This: Js Fiddle

Because it's the minimal functor structure it goes by the name Identity functor. Let us see a simple example where we have two integers 2 and 4 (here we take for simplicity the category of integers as our initial category C) also in this category there is the function `square = x=> x*x` that maps 2 to 4.



If we apply the Id(_) constructor we can map each integers to the Id<number> category. For example 2 will be mapped to Id(2) and 4 maps to Id(4), the only part missing is the correct lifting of the function f Id(f) to this new category. It is easy to see that the correct mapping is:

```
this.map =  (f) => Id(f(value));
```

Because:

```
var square  = x=>x * x;

Id(2).map(square) = Id(square(2))
```

TypeScript

```
class Id<T> {
  private Value: T;
  constructor(value: T) {
    this.Value = value;
  }

  map<T1>(f: (y: T) => T1): Id<T1> {
    return new Id<T1>(f(this.Value));
  }
}

// map
let result1: Id<number> = new Id(5).map(x => x + 2);
```

Run This: TS Fiddle

The type of the functor map method is :

map: (a → b) → f(a) → f(b)

[Side Note: if you rearrange the terms f(a) → (a → b) → f(b) this can be interpreted as if you give me a function from a to b (a → b) and I have an f(a), I can get an f(b)]

Libraries

Mapping objects with. of(_)

In order to cover the first requirement that a functor should map each object x in *C* with an object F(x) in *D* we have used a "**constructor**" that maps a value v to an object literal `Id = (v) => { }`. In many functional libraries online you may find the explicit definition of an .of() that does the same thing

```
Id.of =v=>({ v: v, map: f => Id.of(f(v))  })
```

Functors that provide an `.of` method are called **Pointed Functors**

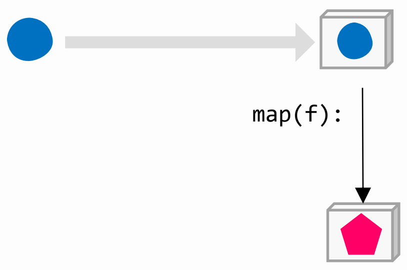
Run This: Js Fiddle

1.2 Commutative Diagrams

There is one important thing about the mapping function, though. The mapping should get the same result for [4] if we take any of the two possible routes to get there. This means **map (aka lifting of a function from C to D) should preserve the structure of C.**

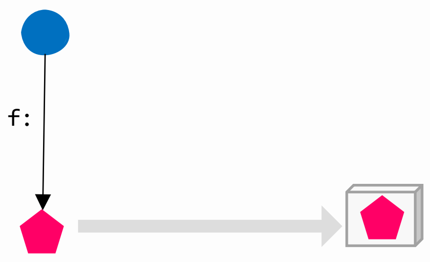
1. We could first get the Functor and then map it. This is the red path on the diagram.

```
Id(y) = Id(x).map(f);
```



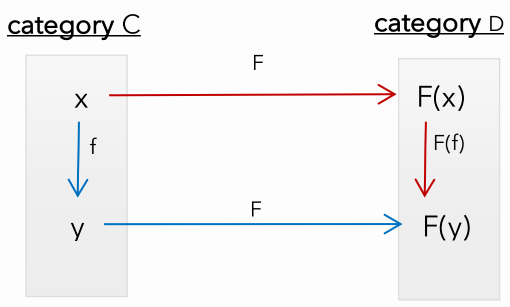
2. Or first lift 2 with f and then get the Functor.

```
Id(y) = Id(f(x));
```



Two objects that were connected with an arrow in category C, should also be connected with an arrow in category D. And reversely, if there was no connection in C there should be no connection in D.

When the red and blue paths give the same result, then we say that the diagram **commutes**



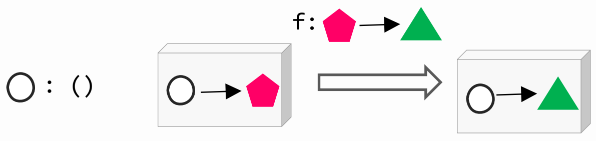
Moreover, that means that the lifting of morphisms (aka arrows, aka functions in programming) preserves the structure of the objects in C.

In practical day to day programming, commutating diagrams means that **we can exchange the order of operations and still get the same result.** It is not something that happens automatically and is something extremely helpful to have when coding.

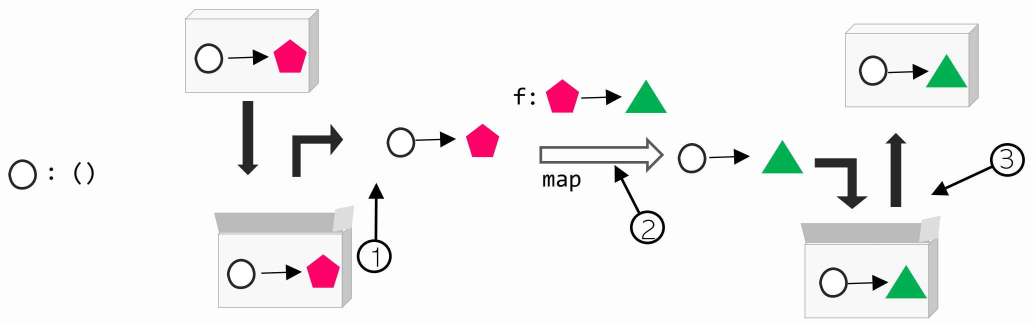
1.3 IO Functor

Functional programming in the same lines provides the IO functor

```
var IO = fn => ({
   map: f => IO(() => f(fn())),
   run: () => fn(),
 })
```



the map implementation should remind you slightly of the *promise map* line of reasoning while implementing it. We first have to "run" the previous computation `fn()` then use the function f to get the lifted value `f(fn())` and then finally rewrap this new value in a new `IO(() => f(fn()))`.



An example application coming from the frontend web design universe uses IO, to isolate the effects of accessing the DOM elements with jQuery

```
var getJqueryElementIO = IO(() => $("#container"));

var title = getHtmlElementIO
.map(x=>x.text())
.map(x.toUpperCase())
.run()
```

Run This: Js Fiddle

It's the same as the Id functor but encloses the value that we provide in a thunk () => { } just to make it lazy. We would have to collapse the function fn() in order to get back the value, most IO implementations around usually provide a run () function that does just that. You can take a look at monet.js library implementation of IO. Also, with a prototype extension on the Object, we can create IOs out of everything on the fly

```
Object.prototype.toIO = function () { return IO(() => this); }
```

And use it like this:

```
var getJqueryElementIO =  $("#container").toIO ();
```

Run This: Js Fiddle

The Lazy loading design pattern (with the map method) is **equivalent** to the **IO Functor**

TypeScript

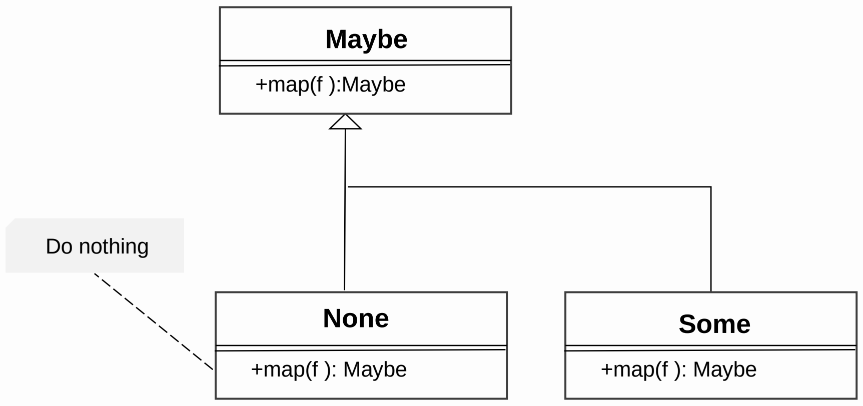
```
class IO<T>
{
    Fn: () => T;
    constructor(fn: () => T) {
        this.Fn = fn;
    }
    map<T1>(f: (y: T) => T1): IO<T1> {
        return new IO<T1>(() => f(this.Fn()))
    };
    match<T1>(f: (y: T) => T1): T1 {
        return f(this.Fn());
    };
    run(): T {
        return this.Fn();
    };
}
```

Run This: TS Fiddle

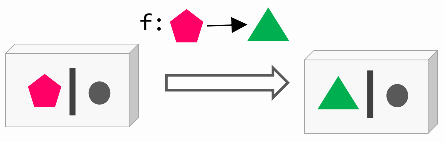
1.4 Maybe Functor

In Functional programming, we use the **Maybe Type** [sometimes also called Option] in order to represent the possibility of something being null. The Maybe solves the null problem by moving the mechanics of null checking inside a functor `map`. **Thus, instead of applying the objects on functions, we reverse the flow and apply the functions onto objects**. Now, we can isolate the effect inside a single point; the `map`.

The standard implementation of the maybe/option Type is a hierarchy with a base class and two sub-types: the `Some` and the `None`. **The None represents the case of null**



After all this discussion, hopefully, the implementation of maybe functor would be apparent



```
var Some = (v) => ({
    map: (f) => Some (f(v)),
    match: pattern => pattern . Some(v),
});
var None = () => ({
    map: (f) => None(),
    match: pattern  => pattern.None(v),
});
```

The None ignores the function f

Run This: Js Fiddle

1.4.1 Pattern matching

For Maybe the pattern matching `.match()` implementation is straightforward. Since it is a hierarchy, we must implement for each sub-type the match, and we are going to call different callbacks, so we can distinguish (pattern match) between Some and None . And we can use it like this:

```
import { Some,None } from "./maybe.js"

Some(5)
.map(x=>x+2)
.match({
    some:v=>console.log(`Some:${v}`),   //we reach this case
    none:()=>console.log(`None`)
})

// the null case
None()
.map(x=>x+2)
.match({
    some:v=>console.log(`Some:${v}`),
    none:()=>console.log(`None`)      //we reach this case
})
```

Run This: Js Fiddle

The type of the pattern matching `pattern` is `({ none:() => T1; some:(v: T)=> T1 })` in typescript just a pair of callbacks for each case (sub-class) (( ) →T1, T →T1) you can see this by running the typescript code below :

TypeScript

```
abstract class Maybe<T> {
  abstract match<T1>(pattern: { none: () => T1; some: (v: T) => T1 }): T1;
  abstract map<T1>(f: (v: T) => T1): Maybe<T1>;
}

export class Some<T> extends Maybe<T> {
  Value: T;
  map<T1>(f: (v: T) => T1): Maybe<T1> {
    return new Some<T1>(f(this.Value));
  }

  match<T1>(pattern: { none: () => T1; some: (v: T) => T1 }): T1 {
    return pattern.some(this.Value);
  }
}

export class None<T> extends Maybe<T> {
  map<T1>(f: (v: T) => T1): Maybe<T1> {
    return new None<T1>();
  }
  match<T1>(pattern: { none: () => T1; some: (v: T) => T1 }): T1 {
    return pattern.none();
  }
}
```

Run This: TS Fiddle

1.5 Maybe Functor Example

The core example use case, upon which we are going to build the various ideas in this book is **a simple retrieval of a `Client` from a database `Repository` that matches a specific Id**.

The mock repository with an in-memory storage of the Clients in an Array might look like this:

```
var Repository = ({
    getById: id =>
```

```
        [new Client(1, "jim"),
        new Client(2, "jane")]
        .filter(client => client.id == id) //return an array [Client]. Could be an
 Empty []
});
```

the `Array.filter(_)` returns an array with the array elements that satisfy the predicate `client => client.id == id`. We can then create a functional extension that:

- Will return a `Maybe.Some` with the first element found.
- Or return a `Maybe.None` if no matching element found. This method is easy to implement :

```
// takes an array [T] and returns a Maybe<T>
export let firstOrNone = function (array) {
    return array.length > 0 ? Some(array[0]) : None()
  }
```

Now we can use `firstOrNone` in order to return a `Maybe<Client>` as the result of our mock Repository:

```
var Repository = ({
    getById: id => firstOrNone(Clients.filter(client => client.id == id)) //return
s a Maybe<T>
});
```

After we have a `Maybe<Client>` then

- we can use the map to access the `Client.name`
- and then use the match to pattern match on the two possible states `Some` and `None`

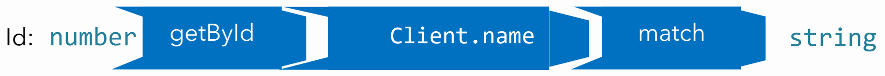
```
import { Client } from "./Client.js"
import { firstOrNone } from "./utils.js"

var Clients = [new Client(1, "jim"), new Client(2, "jane")];

var Repository = ({
    getById: id => firstOrNone(Clients.filter(client => client.id == id)) //return
s a Maybe<T>
});

Repository.getById(1)          //returns a Maybe<Client>
    .map(Client.name)          //Maybe<String>
    .match({                   //String
        some: name => console.log(`client name:${name}`),
        none: () => console.log(`no client found`)
    })
```

Run This: Js Fiddle

TypeScript

```
class ClientService {
     ClientRepository: IRepository<Client> = new MockClientRepository();

     getClientNameById(id: number): String {
       return this.ClientRepository.getById(id)
         .map(Client.getName) //Maybe<String>
         .match({
           some: name => `client name:${name}`,
           none: () => `no client found`
         });
     }
   }
```

Run This: TS Fiddle



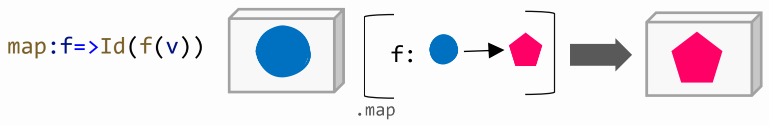
We will progressively build upon this example in order to display the various functional concepts.

In the next section we are going to integrate this example into a simple react.js application.

Monads

2 Monads

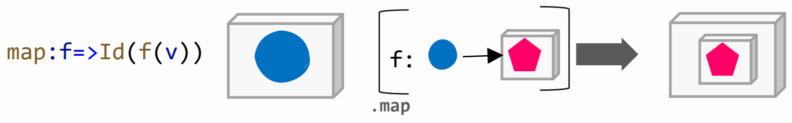
If you remember in the section “Intro to Functors” we stated that when we use the `map` on a function `f: A → B` we transform the value A inside the functor `F(A)` to a new type `F(B)`.



Requirement 2

A map method that takes a function f:a→b and transforms an Id(a) to an Id(b)

Sometimes it might happen that the type `B` is inside a Functor **F** itself: `F(B)`, this means that the function f looks like this `f: A → F(B)` . In those situations, after the map we end up with something like `F(F(B))`. A **Functor-In-A-Functor** situation.

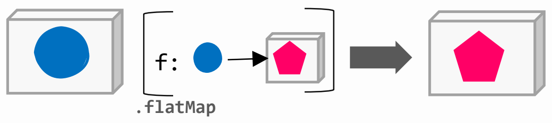


You can see this in the following example in TypeScript (to witness the types) where the type of the result is `result: Id<Id<number>>`

```
class Id<T>
{
  private Value: T
  constructor(value: T) { this.Value = value; }

  map<T1>(f: (y: T) => T1): Id<T1> { return new Id<T1>(f(this.Value)) };

  match(pattern: (y: T) => void): void { pattern(this.Value) };
}

let result: Id<Id<number>> = new Id(5).map(x => new Id(x + 2));
//NOTE: we use the bind only when the lambda returns an Id() type
```

If F is a monad then for those situations where the function is of this type `f: A → F(B)` we can use a method called **`flatMap`** instead of the `map` [In the functional world is known as **`bind`** or `chain` sometimes].

`flatmap:f=>f(v)`



The same reasoning goes for all functors , here some other examples leading to the same situation :

```
IO(()=>5).map(x=>IO(()=> x+1)) === IO(()=>IO(()=> 6))

[5].map(x=>[x+1]) === [[5]]

maybe(5).map(x=>x+1) === maybe(maybe(5))
```

2.1 The Identity Monad

For the Identity functor it's easy to see how someone could implement this flatMap(_). We simply ignore the wrap inside the map `map: f => Id(f(v))`in this way we have just one container at the end wrapped around the value instead of two.

```
const Id = (v) => ({

    value: v,
    map: f => Id(f(v)),
    bind: f => f(v)
  });

let result = Id(5).bind(x => Id(x + 2));
```

There is also the Typescript implementation in order to witness the Types involved:

```
class Id<T>
{
  private Value: T
  constructor(value: T) { this.Value = value; }

  map<T1>(f: (y: T) => T1): Id<T1> { return new Id<T1>(f(this.Value)) };

  bind<T1>(f: (y: T) => Id<T1>): Id<T1> { return f(this.Value) };
}

// map
```

```
let result1: Id<Id<number>> = new Id(5).map(x => new Id(x + 2));
log(result1);
//bind
let result2: Id<number> = new Id(5).bind(x => new Id(x + 2));
log(result2);
```

Run This: Js Fiddle

Run This

2.2 The Array Monad

The native JavaScript array has a default bind method which is called flatMap for the same reason that we described so far . For example, if we have a list of numbers `array = [2, 3, 4, 5]` and want to get a list of the squares, and cubes, we might try to go about it by using map:

```
let result1 = array.map(x => [x * x, x * x * x]);
```

this would result in an Array in an Array situation `[[ 4, 8 ], [ 9, 27 ], [ 16, 64 ], [ 25, 125 ]]`

```
var array = [2, 3, 4, 5]

// map
let result1 = array.map(x => [x * x, x * x * x]);
console.log(result1);
//bind
let result2  =  array.flatMap(x => [x * x, x * x * x]);
console.log(result2);
```

or using the Typescript syntax :

```
var array: Array<number> = [2, 3, 4, 5];

// map
let result1: Array<Array<number>> = array.map(x => [x * x, x * x * x]);
log(result1);
//bind
let result2: Array<number> = array.flatMap(x => [x * x, x * x * x]);
log(result2);
```

Run This

In the next sections we are going now to Extend our Maybe,Either and Promise into Monads.

2.3 Maybe Monad

Maybe monad is an extension of the Maybe Functor. In order to make our `Maybe` functor implementation into a Monad, we must provide a bind method that combines two Maybe Monads into one.

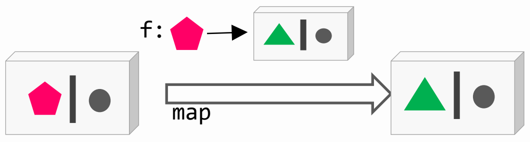
We will use the bind instead of the map in situations where the arrow inside the map returns a `Maybe`.

For example:

```
None().bind(x => Some(5 + x))
```

There are 4 different ways to combine the 2 possible states of maybe (some,none) . A valid `bind` implementation would be the following:

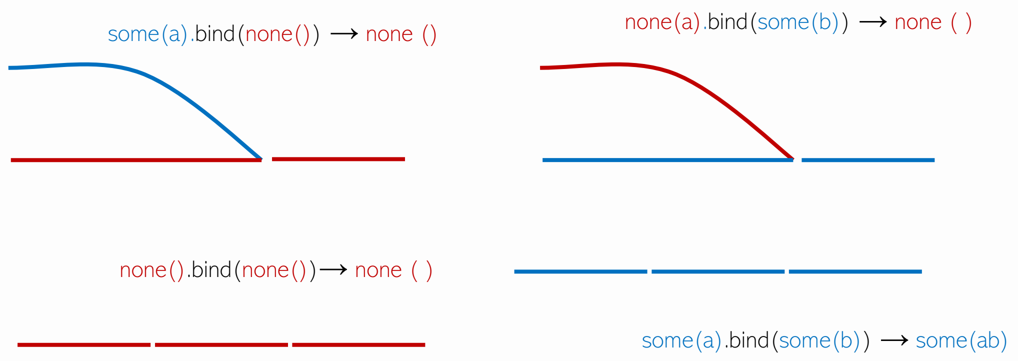
```
 export let Some = (v) => ({
  map: f => Some(f(v)),
  match: pattern => pattern.some(v),
  bind: f => f(v),
});

export let None = () => ({
  map: _ => None(),
  match: pattern => pattern.none(),
  bind: _ => None(),
});
```



the only combination that leads to a new some (_) is when the two paths that both have values are being combined.

```
some(x).bind(none)
none( ).bind(some)
some(x).bind(some)  //only this gives a some result
none( ).bind(none)
```

You can run the example below to verify that only in the case of `Some.bind(Some)` you get a `Some` result. All other combinations result into a `None`


some(a).bind(none()) ⟶ none ()
none(a).bind(some(b)) ⟶ none ( )
none().bind(none())⟶ none ( )
some(a).bind(some(b)) ⟶ some(ab)

You can run the example below to verify that only in the case of `Some.bind(Some)` you get a `Some` result. All other combinations result into a `None`

```
import { Some, None } from "./maybe.js"

//some.bind(None)==None
Some(3)
    .bind(x => None())
    .match({
        some: v => console.log(`Some:${v}`),
        none: () => console.log(`None`)
    })

//None.bind(some)==None
None()
    .bind(x => Some(5 + x))
    .match({
        some: v => console.log(`Some:${v}`),
        none: () => console.log(`None`)
    })

//None.bind(None)==None
None()
    .bind(x => None())
    .match({
        some: v => console.log(`Some:${v}`),
        none: () => console.log(`None`)
    })

//Some.bind(Some)==Some
Some(3)
    .bind(x => Some(5 + x))
    .match({
        some: v => console.log(`Some:${v}`),
        none: () => console.log(`None`)
    })
```

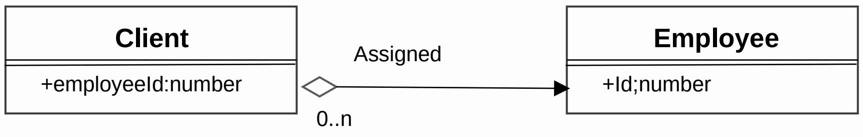
Run This: Js Fiddle

In the following section we are going to see why and how a case for the use of the `bind` *method arise naturally.*

2.4 Maybe Monad Example

Let us say that we have this scenario where we have a list of clients and each client is assigned to an employee. Then let us say we want to get the name of the assigned employee for that client.

This is the standard one-to-many relationships and the way to model it is to add on the Client entity a property that will store the value of the Id of the related employee :



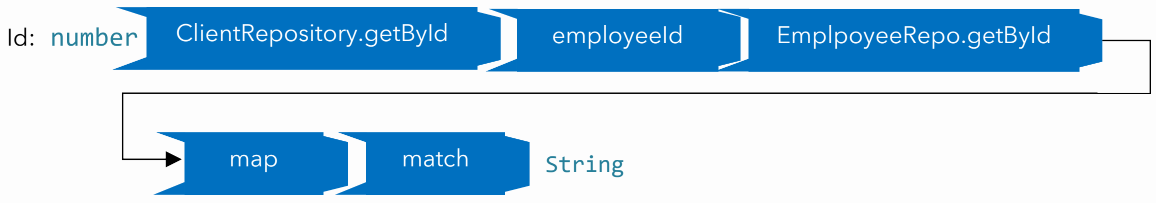
```
class Client {
    constructor(id, name, employeeId) {
      this.id = id;
      this.name = name;
      this.employeeId = employeeId;
    }
```

We will use the `.bind(client=>employees.getById(client.employeeId))`

- First, capture the `employeeId` value from the Maybe returned object from the `ClientRepository`
- And then pass it to the `EmployeeRepository.getById` which returns a Maybe. The final result of the bind is in fact a Maybe :

```
import { Client, Employee } from "./model.js"
import { ClientRepository, EmployeeRepository } from "./Repositories.js"

let ClientService = {
    getEmployeeNameByClientId: clientId =>
        ClientRepository.getById(clientId)       //  Maybe<Client>
            .map(Client.employeeId)              //  Maybe<Integer>
            .bind(EmployeeRepository.getById)    //  Maybe<Employee>
            .map(Employee.name)                  //  Maybe<String>
            .match({                             //  String
                some: value => `employee name: ${value}`,
                none:  _ => `could not get employee name `
            })
};
```

Run This: Js Fiddle



Hopefully, it is easy to understand why this works but it follows a simple breakdown line by line:

- `ClientRepository.getById(clientId)`

  we have a `Maybe<Client>` from the Client Repository

- `.map(Client.employeeId)`

  we get the `employeeId` if there was a Client, so we have a `Maybe <Integer>`

- `.bind(EmployeeRepository.getById)`

  we use bind to access the `employeeId` and pass it to the `EmployeeRepository.getById` this will return a `Maybe<Employee>` which the bind will flatten, and we will have a `Maybe<Employee>` as a result

- `.map(Employee.name)`

  we then map to the name of the employee and thus get a `Maybe<String>`

- `.match({ ...})`

  finally, we pattern match in order to get a final `String` result.

Only in the case where both the `ClientRepository` and the `EmployeeRepository` have returned a `Maybe.Some` then the result of the `.bind(EmployeeRepository.getById)` is a `Maybe.Some` that contains the Employee assigned to the `Client` in all other case the overall result is a None

Next:

The problem with using `Maybe`, is that we cannot distinguish which repository didn't return an Object. We will solve this problem by using the Either monad instead in the next section.