Advanced Topics in Functional and Reactive Programming: Functional Programming

Majeed Kassis

Arity

- The **number** of arguments a function takes.

```
1  // unary
   const identity = (x) => x;
2  // binary
   const add = (x,y) => x + y;
3  // variadic
   function variadic(){
4    console.log(arguments)
   }
5
6
7
8
```

- Functions in Javascrip
  - Arguments are storec
  - **arguments** object is t
  - Available for every fu

```
1 variadic(1)
  ↗ [1]
2 variadic(1,2,3)
  ↗ => [1,2,3]
3
4
```

Higher-Order Functions (HOF)

- A function which takes a function as an argument and/or returns a function.
- Examples:

```
const filter = (predicate, xs) => xs.filter(predicate)

const is = (type) => (x) => Object(x) instanceof type

filter(is(Number), [0, '1', 2, null]) // [0, 2]
```

Higher-Order Functions: 1

- **every**: check if **every** element in container is True following a **predicate**

```
1  //define
2  const every = (arr,fn) => {
3    let result = true;
4    for(const value of arr)
5      result = result && fn(value);
6    return result;
7  }
8  //use
9  var fruit = ['cherries', 'apples', 'bananas'];
10 console.log(every(fruit, (x) => x.includes('c'))) //false
11 console.log(every(fruit, (x) => x.includes('s'))) //true
```

Higher-Order Functions: 2

- **some**: check if **some** element in container is True following a **predicate**

```
1  //define
2  const some = (arr,fn) => {
3    let result = true;
4    for(const value of arr)
5      result = result || fn(value);
6    return result;
7  }
8  //use
9  var fruit = ['cherries', 'apples', 'bananas'];
10 console.log(some(fruit, (x) => x.includes('c'))) //true
11 console.log(some(fruit, (x) => x.includes('g'))) //false
```

Higher-Order Functions: 3

- **`sort`**: sorts a container following a **compareFunction**

```
1 var fruit = ['cherries', 'apples', 'bananas'];

2 //sort
  fruit.sort();
3 console.log(fruit) //["apples", "bananas", "cherries"]

4 //reverse order sort
  fruit.sort((a,b) => { return (a[0] < b[0]) ? 1 : (a[0] > b[0]) ? -1 : 0 });
5 console.log(fruit) //["cherries", "bananas", "apples"]

6

7
8
```

Closure

- A closure is a **scope** which **retains** variables and their values.
  - Closures are needed for when we return **inner** functions, to be executed **later**
  - This is important for partial application to work.
- Example:

```
const addTo = (x) => {
    return (y) => {
        return x + y
    }
}
```

```
var addToFive = addTo(5)
addToFive(3) // => 8
```

- Explanation:
  - In this case the x is retained in `addToFive`'s **closure** with the value 5.
  - We can then call `addToFive` with the y and get back the desired number.
- Closures are commonly **used** in **event handlers** so that they still have access to variables defined in their **parents** when they are eventually called.

Closure: Scope Access 1

- The closure has access to **three** different scopes:
  - Local Scope: Variables that are declared in its own declaration
  - Global Scope: Access to the global variables
  - Outer Scope: Access to the outer function's variables
- Example 1:

```
function outer() {
  function inner() {
    let a = 5;
    console.log(a)
  }
  inner() //call the inner function.
}
```

- Output: 5

Closure: Scope Access 2

- Example2:

```
let global = "global"
function outer() {
  function inner() {
    let a = 5;
    console.log(global)
  }
  inner() //call the inner function.
}
```

- Output: g

Closure: Scope Access 3

- Example3:

```
let global = "global"
function outer() {
  let outer = "outer"
  function inner() {
    let a = 5;
    console.log(outer)
  }
  inner() //call the inner function.
}
```

- Output: outer

Higher-Order Functions: 4

- **tap**: takes a value and returns a function that **has** the **closure** over value and it will be executed on the value.
  - Used for **debugging** purposes - mostly
  - Instead of modifying a function – we wrap it using tap!
    - This is done for a **specific** value we wish to test on!

```
1 const tap = (value) => {
    return (fn) => {
2     // added debugging functionality
      console.log('received: ', value);
3     if (typeof(fn) === 'function'){
        return fn(value);
4     }
    };
5 };

6
```

tap


```
1  //function to debugValue
2  const myFn = (it) => 5 + it;
3
4  // define value to begin with
5  const debugValue = 5;
6
7  // tap value
8  const tapped = tap(debugValue);
9
10 // execute function
11 console.log('returned: ', tapped(myFn));
```

Output:

'received: ‘ 5

'returned: ‘ 10

JavaScript: Parameters and Arguments 1/4

- Definition:

```
1 const addTwo = (x, y) => x + y;
```

- Example1:

```
1 console.log(addTwo(1));
```

```
2 ↗ NaN
```

- Example2:

```
1 console.log(addTwo(1, 2));
```

```
2 ↗ 3
```

- Example3:

```
1 console.log(addTwo(1, 2, 3));
```

```
2 ↗ 3
```

Higher-Order Functions: 5

- **unary**: **wraps** a function that takes more than one argument, and forces it to take **one** argument **only**.
- Use Case:
  - array.map() – provides **three** arguments:
    - element, index, array

```
1 ;['1', '2', '3'].map((x, y, z) => console.log(x, y, z));
2
3 =>'1'0[ "1", "2", "3" ]
4 =>'2'1[ "1", "2", "3" ]
5 =>'3'2[ "1", "2", "3" ]
```

Use Case:

- parseInt() – accepts **two** arguments:
  - element, radix (base)
- JavaScript will provide parseInt() both element, and index.
  - This results in unexpected behavior!

```
1 [1, 2, 3].map((a) => { return a * a })
  ↗ [1, 4, 9]
2
  ['1', '2', '3'].map(parseInt)
3 ↗ [1, NaN, NaN]
4
5
```

- Solution? unary

unary: Implementation


```
1  // implementation
   const unary = (fn) =>
2    fn.length === 1 // if argument size for fn is 1
       ? fn // return fn as is
3      : (arg) => fn(arg) //otherwise, wrap fn with function accepting 1 argument
4
```

```
1  // use example
   ['1', '2', '3'].map(unary(parseInt))
2
   ↗ [1, 2, 3]
3
4
```

Higher-Order Functions: 6 1/2

- **once**: A higher-order function that will run the given function exactly once.
  - Consequent executions will not run the function!
  - Basically separating this functionality to its own function.
- Use examples:
  - To set up a third-party library only once
  - To Initiate the payment set up only once
  - To do a bank payment request only once

once


```
1  // once implementation
2  const once = (fn) => {
3    let done = false;
4    return function () {
5      return done
6        ? undefined
7        : ((done = true), fn.apply(this, arguments));
8    };
9  };
10
11 //my function
12 var payment = () => console.log("Payment is done");
13 console.log('execute normally:');
14 payment();
15 payment();
16
17 var payOnce = once(payment);
18 console.log('execute once:');
19 payOnce();
20 payOnce();
```

```
13  'execute normally:'
14  'Payment is done'
15  'Payment is done‘
18   'execute once:'
19  'Payment is done'
20
```

Higher-Order-Functions: 7 1/2

- `Memoize`: A special higher order function that allows the function to **remember** or memorize its result.
  - Our caching function – useful for functions that require long CPU time.
    - And input is repeated more than once

```
1 const memoize = (fn) => {
2   const lookupTable = {};
3   return (arg) => lookupTable[arg] || (lookupTable[arg] = fn(arg));
4 }
```

Memoize: Verbose Version 2/2

```
const verboseMemoize = (fn) => {
  const lookupTable = {};
  return (arg) => {
    if (lookupTable[arg]){
      console.log('returning value from table')
      return lookupTable[arg]
    } else {
      console.log('value not found: calculating')
      lookupTable[arg] = fn(arg);
      return lookupTable[arg];
    }
  };
}
```

```
const memoizedFactorial =
            verboseMemoize(factorial);
console.log(memoizedFactorial(42))
console.log(memoizedFactorial(42))

//output:
'value not found:
calculating' 1.40500611775287 98e+51
'returning value from table'
1.40500611775287 98e+51
```

Partial Application of Functions

- Partially applying a function means creating a new function by pre-filling **some** of the arguments to the original function.

```
// Helper to create partially applied functions
// Takes a function and some arguments
const partial = (f, ...args) =>
    // returns a function that takes the rest of the arguments
    (...moreArgs) =>
        // and calls the original function with all of them
        f(...args, ...moreArgs)

// Something to apply
const add3 = (a, b, c) => a + b + c
// Partially applying `2` and `3` to `add3` gives you a one-argument function
const fivePlus = partial(add3, 2, 3) // (c) => 2 + 3 + c

fivePlus(4) // 9
```

Currying

- A functional technique that allows transforming a function of arity N into N functions of arity 1, to executing!
  - Pass all of the arguments, get a result.
  - Pass a subset of those arguments, get a function back that's waiting for the rest of the arguments.
- Conversion:

```
var greet = function(greeting, name) {
    console.log(greeting + ", " + name);
};

greet("Hello", "Heidi"); //"Hello, Heidi"
```

```
var greetCurried = function(greeting) {
    return function(name) {
        console.log(greeting + ", " + name);
    };
};
```

- Use:

```
var greetHello = greetCurried("Hello");
greetHello("Heidi"); //"Hello, Heidi"
greetHello("Eddie"); //"Hello, Eddie"
```

```
//"Hi there, Howard"
greetCurried("Hi there")("Howard");
```

Auto Currying

- A function that **automatically** performs currying:
  - if given **less** than its correct number of arguments returns a function that takes the rest.
  - When the function gets the correct number of arguments it is then evaluated.
- Example: [`Ramda.js`]

```
const add = (x, y) => x + y
const curriedAdd = R.curry(add)

curriedAdd(1, 2) // 3
y = curriedAdd(1) // (y) => 1 + y
y(3) // 4
curriedAdd(1)(2) // 3
```

Ramda.js: A Javascript Library

- A library designed specifically for:
  - A **functional programming** style
  - Allows creating functional pipelines easily
  - Enforces immutability: pipeline never mutates user data.
- Ramda emphasizes a purer functional style:
  - Immutability of data
  - Side-effect free functions
- Ramda functions are automatically curried!
  - The data to be operated on is generally supplied last.
    - Allows easy use of currying.
  - https://ramdajs.com/docs/#curry

Currying Use Case 1: find numbers in Array

```
let match = R.curry(function(expr, str) {
  return str.match(expr);
});

let filter = R.curry(function(f, ary) {
  return ary.filter(f);
});

let hasNumber = match(/[0-9]+/)
let findNumbersInArray = filter(hasNumber)

console.log(findNumbersInArray(["a5", "js","n1"]))

//output:
=> [ "a5", "n1" ]
```

Currying Use Case 2: apply a function on an array

```
1  let map = R.curry(function(f, ary) {
     return ary.map(f);
2  });

3  let squareAll = map((x) => x * x)

4  squareAll([1,2,3])
   => [1,4,9]
5
6
7
8
```

- Allows giving meaningful names to functions
- Map is a **method**, squareAll is a **function**
  - Increases code readability
  - Makes it more functional

Continuation

- Continuation: The part of the code that is **yet** to be executed.
- In asynchronous programming:
  - When the program needs to **wait** to receive data before it can continue.
  - The response is often passed off to the **rest** of the program

```
const continueProgramWith = (data) => {
  // Continues program with data
}

readFileAsync('path/to/file', (err, response) => {
  if (err) {
    // handle error return
  }
  continueProgramWith(response)
})
```

Purity and Side Effects

- **Purity**: If the return value is **only** determined by its input values and does not produce side effects.
- **Side Effect**: If a function interacts with (reads from or writes to) external mutable state.
- Pure Function:

```
const greet = (name) => `Hi, ${name}`
greet('Brianne') // 'Hi, Brianne'
```

- Relies on data **outside** function scope: [Side Effect]

```
window.name = 'Brianne'
const greet = () => `Hi, ${window.name}`
greet() // "Hi, Brianne"
```

- Modifies state of variable **outside** function scope: [Side Effect]

```
let greeting
const greet = (name) => { greeting = `Hi, ${name}` }
greet('Brianne') greeting // "Hi, Brianne"
```

Idempotent Functions

- A function is idempotent if reapplying it to its result does not produce a different result.

$$f(f(x)) \simeq f(x)$$

- Examples:

```
Math.abs(Math.abs(10))

sort(sort(sort([2, 1])))
```

Point-Free Style or Tacit programming

- Function definition does **not explicitly** identify the arguments used
  - Instead the definitions merely **compose** other functions
    - This style requires currying or Higher-Order functions

```
const map = (fn) => (list) => list.map(fn)
const add = (a) => (b) => a + b

// Not points-free - `numbers` is an explicit argument
const incrementAll = (numbers) => map(add(1))(numbers)

// Points-free - The list is an implicit argument
const incrementAll2 = map(add(1))
```

Predicate

- A function that returns true or false for a given value.
  - A common use of a predicate is as the callback for array filter.
- Example:

```
const predicate = (a) => a > 2

[1, 2, 3, 4].filter(predicate) // [3, 4]
```

Pipeline: Pipelining functions 1/2

- Combines n functions. A pipe flowing from **left** to **right**
  - Calling each function with the output of the previous one
  - Equivalent to `.then()` in Java functional programming

```
1 const pipe = (...fns) => {
2   return (value) =>
3     fns.reduce((acc, fn) => fn(acc), value)
4 };
```

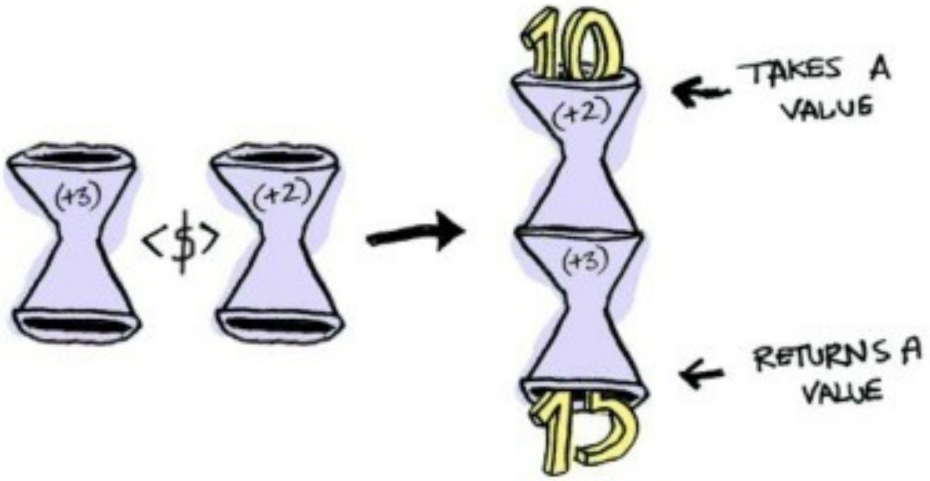
Pipeline: Pipelining functions 2/2

```
let splitIntoSpaces = (str) => str.split(' ');
let count = (array) => array.length;

const countWords = pipe(splitIntoSpaces, count);

//count the words
let result = countWords("hello you're learning about pipelining!");
console.log(result)
=> 5
//return even if even number or odd if odd number
let oddOrEven = (ip) => ip % 2 === 0 ? 'even' : 'odd';

//create a pipeline: split sentence, count words, check if odd/even
const oddOrEvenWords = pipe(splitIntoSpaces, count, oddOrEven);
result = oddOrEvenWords("hello you're learning about pipelining!")
console.log(result)
↗ Odd
```

compose: functions composition

- The act of putting two functions together to form a third function
  - The output of one function is the input of the other.

```
// Definition: two function composition
const compose = (f, g) => (a) => f(g(a))
// Usage
const floorAndToString = compose((val) =>
    val.toString(), Math.floor)
// '121'
floorAndToString(121.212121)
```



compose: multiple functions composition 1/3

```
1 //multiple functions composition
2 const compose = (...fns) => {
3   return (value) => fns.reverse().reduce((acc, fn) => fn(acc), value)
4 };
5
6 //split sentence into words following space delimiter
7 let splitIntoSpaces = (str) => str.split(' ');
8 //return size of array
9 let count = (array) => array.length;
10 //return if even if even number or odd if odd number
11 let oddOrEven = (ip) => ip % 2 === 0 ? 'even' : 'odd';
12
13 //compose 2 functions
14 const countWords = compose(count, splitIntoSpaces);
15 //compose 3 functions
16 const oddOrEvenWords = compose(oddOrEven, count, splitIntoSpaces);
```

compose: multiple functions composition 2/3

```
1 //count the words
2 let result = countWords(“hello you're learning about composition!”);
3 console.log(result)
4 => 5
5 const oddOrEvenWords = compose(oddOrEven, count, splitIntoSpaces);
6 result = oddOrEvenWords(“hello you're learning about composition!”)
7 console.log(result)
8 => ‘odd’
```

compose: Composition is **associative** 3/3

```
1  //compose(f, compose(g, h)) == compose(compose(f, g), h);
2
3  //compose(compose(f, g), h)
4  let oddOrEvenWords = compose(compose(oddOrEven,count),splitIntoSpaces);
5  let oddOrEvenWords("hello you're learning about composition!")
6  ↗ ['odd']
7
8  //compose(f, compose(g, h))
9  let oddOrEvenWords = compose(oddOrEven,compose(count,splitIntoSpaces));
10 let oddOrEvenWords("hello you're learning about composition!")
11 ↗ ['odd']
```

- Composition is **fundamental** to functional programming!
  - Use it abundantly in your code!

Contracts

- Specifies the obligations and guarantees of the behavior from a function or expression at runtime.
  - This acts as a set of rules that are expected from the input and output of a function or expression
  - Also includes errors are generally reported whenever a contract is violated.

```
// Define our contract : int -> boolean
const contract = (input) => {
  if (typeof input === 'number')
    return true
  throw new Error('Contract violated: expected int -> boolean')
}

const addOne = (num) => contract(num) && num + 1
addOne(2) // 3
addOne('some string') // Contract violated: expected int -> boolean
```

Contracts: Example

```
// Define our contract : int -> boolean
const contract = (input) => {
  if (typeof input === 'number')
    return true
  throw new Error('Contract violated: expected int -> boolean')
}

const addOne = (num) => contract(num) && num + 1
addOne(2) // 3
addOne('some string') // Contract violated: expected int -> boolean
```

Pointed Functor

- A **subset** of Functor, which has an interface containing the **of** contract
  - It stored a **single** variable inside a container.
  - Purpose: Using a more functional code, instead of **new** (object oriented)
  - Called **lifting** and implemented using of

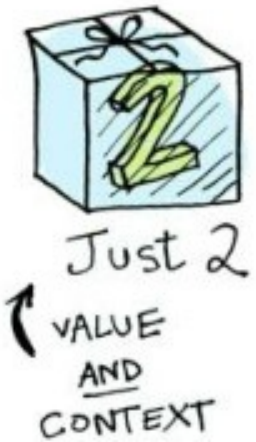
```
1  // Defining Container Object
2  const Container = function(val) { this.value = val; }
3
4  // Before: Creating a new Container
5  let testValue = new Container(3)
6  ↗Container(value:3)
7  let testObj = new Container({a:1})
8  ↗Container(value:{a:1})
9  let testArray = new Container([1,2])
10 ↗Container(value:[1,2])
```

```
1  // Adding of support
2  Container.of = function(value){return new Container(value);}
3
4  // After: Creating a new Container
5  testValue = Container.of(3)
6  ↗Container(value:3)
7  testObj = Container.of({a:1})
8  ↗Container(value:{a:1})
9  testArray = new Container([1,2])
10 ↗Container(value:[1,2])
```

Functor

- A **container** that implements a **map** contract adhering to two rules:
  - Preserves Identity: `object.map(x => x) ≍ object`
  - Composability: `object.map(compose(f, g)) ≍ object.map(g).map(f)`
- Example:
- **Container**:
  - an object that **holds** a **value** inside it
  - allows applying operations on the value.

```
let arr = [1, 2, 3] // functor
const f = x => x + 1
const g = x => x * 2

// identity
arr.map(x => x) // = [1, 2, 3]

// composability
arr.map(x => f(g(x))) // = [3, 5, 7]
arr.map(g).map(f) // = [3, 5, 7]
```



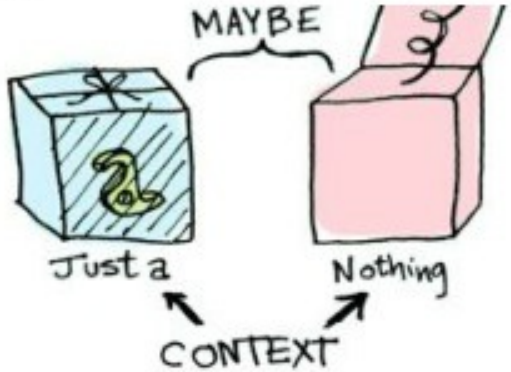
Functor: Map Contract Code Example

```
1  const Container = function(val) { this.value = val; };
2  // adding of support
3  Container.of = function(value) { return new Container(value); }
4  // adding map support
5  Container.prototype.map = function(fn){ return Container.of(fn(this.value)); };
6
7  // identity
8  let id = (x) => x;
9  // function
10 let double = (x) => x + x;
11 // composition
12 let doubleDouble = (x) => double(double(x));
13
14 let testValue = new Container(3);
15 console.log('element=', testValue);
16 testValue = testValue.map(id);
17 console.log('identity=', testValue)
18 let testValue1 = testValue.map(double);
19 console.log(testValue1);
20 let testValue2 = testValue.map(doubleDouble);
21 console.log(testValue2);
```

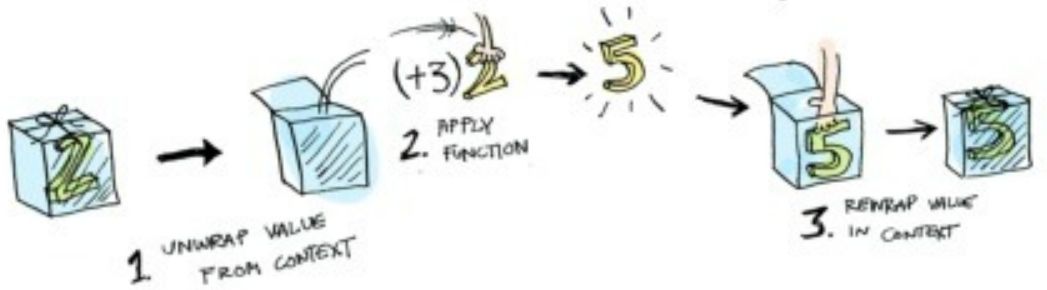

```
1  output:
15 'element='{ "value": 3 }
17 'identity='{ "value": 3 }
19 { "value": 6 }
21 { "value": 12 }
```

The **Maybe** Functor

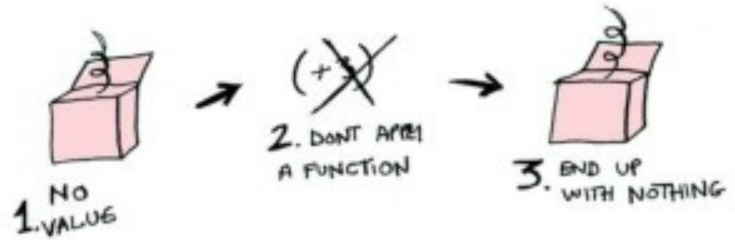
- Maybe:
  - A functor that **allows** undefined/null elements.
  - Does not throw exceptions in case of undefined/null elements.
- Main Use:
  - Used to store responses return from a **network** request
  - Cases where the result might return errors
    - Network queries
    - Functions with exceptions
  - Basically, in any case an **error** might return due to a **request**
    - Instead of throwing an exception due to the error
    - The Maybe object will hold a **null** instead



The **Maybe** Contract



- A Functor:
  - Implements the **map** contract
  - In addition, implements the isNothing contract
- isNothing:
  - Checks whether an element is undefined/null or not.
- map behavior:
  - **Avoids** applying the passed function to undefined/null elements using isNothing



The **Maybe** Functor: Implementation 3/6

```
1  const MayBe = function(val) {
2    this.value = val;
3  }
4
5  MayBe.of = function(val) {
6    return new MayBe(val);
7  }
8
9  MayBe.prototype.isNothing = function() {
10   return (this.value === null || this.value === undefined);
11 }
12
13 MayBe.prototype.map = function(fn) {
14   return this.isNothing() ? MayBe.of(null) : MayBe.of(fn(this.value));
15 }
```

The **Maybe** Functor: Use Example


```
1 // maybe with with data
2 const maybe2 = MayBe.of(5);
3 console.log('maybe 1:‘, maybe2)
4 console.log('map maybe 1:‘, maybe2.map(x => x*x))
5
6 // maybe with without data
7 const maybe = MayBe.of(null);
8 console.log('maybe 2:‘, maybe)
9 console.log('map maybe 2:‘, maybe.map(x => x*x))
```

```
3 'maybe 1:' { "value": 5 }
  'map maybe 1:' { "value": 25 }
4
  'maybe 2:' { "value": null }
8 'map maybe 2:' { "value": null }

9
```

The **Maybe** Functor: Real World Example 5/6

- Execute an HTTP request to a website.
  - Request top 10 results.
- Problem: Errors might return as a result
  - Solution: Maybe Functor to store **response**

```
let getTopTenSubRedditPosts = (type) => {
  let response
  try{
    response = JSON.parse(request(
          'GET',"https://www.reddit.com/r/subreddits/" + type + ".json?limit=10")
          .getBody('utf8'))
  }catch(err) {
    response = {
      message: "Something went wrong" ,
      errorCode: err['statusCode']
    }
  }
  return response
}
```

The **Maybe** Functor: Real World Example 6/6

```
let getTopTenSubRedditData = (type) => {
  let response = getTopTenSubRedditPosts(type);
  return MayBe.of(response)
        .map((arr) => arr['data'])
        .map((arr) => arr['children'])
        .map((arr) => arrayUtils.map(arr,
                        (x) => {
                          return {
                                  title : x['data'].title,
                                  url : x['data'].url
                          }
                        }
        ))
}
```

- In case of success? We receive the desired result!
- In case of error somewhere? `MayBe { value: null }`
  - Problem? Debugging is impossible!
    - Because there is **no separation** between **normal** response, and erroneous response!

The `Either` Functor

- Comes to assist us to **detect** the place of **failure**
  - Using **Maybe** we cannot detect the function that caused it to fail
    - Only manually – by checking each function in the pipe and its results
  - Using **Maybe** we only know that something has **failed**
    - By looking at the final result – we see null/undefined
- Done by **separating** a regular response from an erroneous response
  - Regular returned data is stored in one object
  - Exceptions caught are stored in another object

The `Either` Contract

- A Functor that can be one of two Functors:
  - Nothing, or Some
- The Nothing Functor:
  - Used to store the **error** itself alongside its **error message**.
  - **map does not execute the passed function on the data, returns data as is!**
- The Some Functor:
  - Used to store the **data** itself.
  - Acts as a regular Functor without additional functionality

The **Nothing/Some** Functors: Implementation 3/5

```
const Nothing = function(val) {
  this.value = val;
};

Nothing.of = function(val) {
  return new Nothing(val);
};

Nothing.prototype.map = function(f) {
  return this;
};
```

```
const Some = function(val) {
  this.value = val;
};

Some.of = function(val) {
  return new Some(val);
};

Some.prototype.map = function(fn) {
  return Some.of(fn(this.value));
};
```

The `Either` Functor: Real World Example 4/5

```
let getTopTenSubRedditPostsEither = (type) => {
  let response
  try{
    response = Some.of(JSON.parse(
       request('GET',"https://www.reddit.com/r/subreddits/" + type + ".json?limit=10")
       .getBody('utf8')))
  }catch(err) {
    response = Nothing.of({ message: "Something went wrong" ,
                            errorCode: err['statusCode']
    })
  }
  return response
}
```

- Instead of a returning a response then wrap it by **Maybe**
  - We either return **Some** or **Nothing**
  - The modified behavior of the map function in Nothing ensures we do not lose the stored message.

The `Either` Functor: Real World Example 5/5

```
let getTopTenSubRedditData = (type) => {
  let response = getTopTenSubRedditPosts(type);
  return response
        .map((arr) => arr['data'])
        .map((arr) => arr['children'])
        .map((arr) => arrayUtils.map(arr,
                          (x) => {
                            return {
                                    title : x['data'].title,
                                    url : x['data'].url
                            }
                          }
        ))
}
```

- In case of successful execution:
  - No difference between Maybe and Either.
  - Some === Maybe
- In case of a exception:

```
// using maybe:
↗ MayBe {value : null}
// using either:
↗ Nothing { value: { message: 'Something went wrong', errorCode: 404 } }
```

The **Monad** Functor

- Monad:
  - A functor that implements the **chain** contract
- The Chain Contract:
  - flatten **map** with context: `Monad(Monad(a)) => Monad(b)`
  - Basically, applies map on a, then flattens result.
- Implementation:
  - `join()`: returns the value stored in the Container
    - It **unwraps/flattens** the container.
  - `chain()`: applies `map()` then `join()`
    - Same as FlatMap!

The **Monad** Functor: join() Implementation

```
1  MayBe.prototype.join = function() {
2    return this.isNothing() ? MayBe.of(null) : this.value;
3  }
4
5  let joinExample = MayBe.of(MayBe.of(5))
6  ↗ MayBe { value: MayBe { value: 5 } }
7
8  joinExample.join()
9  ↗ MayBe { value: 5 }
10
11 joinExample.map((outsideMayBe) => {
12   return outsideMayBe.map((value) => value + 4)
13 })
14
15 ↗ MayBe { value: MayBe { value: 9 } }
16
17 joinExample.join().map((v) => v + 4)
18 ↗ MayBe { value: 9 }
```

The **Monad** Functor: chain() Implementation

```
1 MayBe.prototype.chain = function(f){
2     return this.map(f).join()
3 }
4
```

Use Case: Monad

- Counting the number of **comments** in a website following a specific query:
  - Website: Reddit.com
  - Search Query: Functional Programming

The **Comonad** Functor

- In contrary to a monad, comonad allows **revealing** the inner data.
- Comonad implements two contracts:
  - Extract, Extend
- The **Extract** Contract:
  - Extracts the value stored in the comonad
  - Returns the **value** as is
- The **Extend** Contract:
  - Applies a **function** f on the comonad value x
  - Returns a comonad containing y=f(x)

The **Comonad** Functor: Implementation

```
1  const CoIdentity = (v) => ({
2    val: v,
3    extract () {
4      return this.val
5    },
6    extend (f) {
7      return CoIdentity(f(this))
8    }
9  })
10
11 // using extract:
12 CoIdentity(1).extract() // 1
13
14 // using extend:
15 CoIdentity(1).extend((co) => co.extract() + 1) // CoIdentity(2)
```