FP beginners workshop (1/3)

- what is/why FP
- pure functions
- mutable state
- currying
- map / filter / reduce
- lenses
- composition

Next workshops:

Functional Reactive Programming && Algebraic Data Structures *_*

what is FP

- use side effects free functions as a basic unit of abstraction
- build on it to achieve every™ abstraction you need

why FP

- write generic, reusable, extensible code
- manage and separate state and side effects from logic
- don't learn syntax or patterns (couGoFh!), learn concepts

pure functions

no reference to external code, no side effects

```
// Person is a Int which represents health points
const jules = 120
const shoot = (person) => person - 25 // ✓ PURE
assert.equal(shoot(jules), 95)
```

PURE

pure functions

no reference to external code, no side effects

mutable state

```
// Person is a Int which represents health points
// People is a Object (String, Person)
const people = { MarsellusWallace: 80, MiaWallace: 60 }
const shoot = (name) => people[name] -= 25 // X IMPURE
shoot('MarsellusWallace')
assert.equal(people.MarsellusWallace, 55)
```

IMPURE + MUTABLE STATE

pure functions

no reference to external code, no side effects

mutable state

```
// Person is a Int which represents health points
// People is a Object (String, Person)
const people = { MarsellusWallace: 80, MiaWallace: 60 }
const shoot = (people, name) => {
  if (!name in people) throw "not found" // X IMPURE
  people[name] -= 25 // X IMPURE
}
shoot(people, 'MiaWallace')
assert.equal(people.MiaWallace, 35)
```

IMPURE + IMMUTABLE

pure functions

no reference to external code, no side effects

```
// Person is a Int which represents health points
// People is a Object (String, Person)
const people = { MarsellusWallace: 80, MiaWallace: 60 }
const shoot = (people, name) => {
  if (!name in people) return people
  return Object.assign({}, people,
        { [name]: people[name] - 25 }) // X PURE
}
const newPeople = shoot(people, 'MiaWallace')
assert.equal(people.MiaWallace, 60)
assert.equal(newPeople.MiaWallace, 35)
```

PURE + IMMUTABLE

is add10 pure?

```
const add5 = (a) => a+5
const add10 = (a) => add5(a)+5
```

is add10 pure? NO

```
let add5 = (a) => a+5
add5 = (a) => 3
const add10 = (a) => add5(a)+5
const newAdd10 = (add5, a) => add5(a) + 5
```

add10 references add5, which could change

strike the right balance between readability and purity

bonus slide: is add5 really pure? NO

```
const add5 = (a) => a+5
const b = {}
b.valueOf = Math.random
add5(b) //5.772593754043157
add5(b) //5.333966641329766
add5(b) //5.90834816009329
```

(sadpanda)

use a better language with guarantees

or don't hire trolls (or don't tell them)

currying

function with `n` arguments

-> function which accept `0..n` arguments

```
const { curry } = require('ramda')
const add = curry((a, b, c) => a + b + c)
add(1,2,3) === add(1,2)(3) === add(1)(2, 3) == add(1)(2)(
```

currying

function with `n` arguments

-> function which accept `0..n` arguments

```
const updateStatusWithDeps = curry(
  (db, mail, templates, status, person) => Promise.all([
    db.upsert({ [person]: status }),
    mail.send(templates[status], person, status)
  ]))

const db = ..., mail = ..., templates = ...

updateStatusWithDeps(db, mail, templates, 'killed', 'Bill
const updateStatus = updateStatusWithDeps(db, mail, templ
const markDead = updateStatus('killed')
//what if it were (db, mail, templates, person, status)?
//think about arguments position, data last

const people = ['O-Ren Ishii', 'Vernita Green',
                'Budd', 'Elle Driver', 'Bill']

Promise.all(people.map(markDead))
```

example

```
//Person is [String, Int] which represents name and healt
const people = [
  ['Jules Winnfield', 120],
  ['Vincent Vega', 120],
  ['Marsellus Wallace', 80],
  ['Mia Wallace', 60],
  ['Butch Coolidge', 100],
  ['Winston Wolfe', 200]
]

const render = () => { //hard to test
  var output = ''
  for (let i = 0; i < people.length; i++) {
    output += people[i][0] + ': ' +
              people[i][1] + ' life points'
  }
  console.log(output) // hard to test
}
//reference state outside the function
```

better

```
//Person is [String, Int] which represents name and healt
const people = [
  ['Jules Winnfield', 120],
  ['Vincent Vega', 120],
  ['Marsellus Wallace', 80],
  ['Mia Wallace', 60],
  ['Butch Coolidge', 100],
  ['Winston Wolfe', 200]
]

const show = (people) => {
  var output = ''
  for (let i = 0; i < people.length; i++) {
    output += people[i][0] + ': ' +
              people[i][1] + ' life points'
  }
  return output
}
```

map

applies a function to all elements of a List

```
//Person is [String, Int] which represents name and healt
//People is a List of Person
const people = [
  ['Jules Winnfield', 120],
  ['Vincent Vega', 120],
  ['Marsellus Wallace', 80],
  ['Mia Wallace', 60],
  ['Butch Coolidge', 100],
  ['Winston Wolfe', 200]
]

const showPerson = ([name, life]) =>
  name + ': ' + life + ' life points'
const show = (people) =>
  people.map(showPerson).join('\n')
```

filter

returns elements of a List of elements which satisfies a predicate

```
//Person is [String, Int] which represents name and healt
//People is a List of Person
const people = [
  ['Jules Winnfield', 120],
  ['Vincent Vega', 120],
  ['Marsellus Wallace', 80],
  ['Mia Wallace', 60],
  ['Butch Coolidge', 100],
  ['Winston Wolfe', 200]
]
const dancers = ['Vincent Vega', 'Mia Wallace']

const showPerson = ([name, life]) =>
  name + ': ' + life + ' life points'
const canDance = ([name, life]) =>
  dancers.indexOf(name) !== -1 }
const showDancers = (people) =>
  people.filter(canDance).map(showPerson).join('\n')
```

reduce

```
//Person is [Int, [String]] which represents health
//points and a list of targets' names
const people = {
  'Jules': [120, []],
  'Vincent': [120, ['Pumpkin', 'Bonnie']],
  'Marsellus': [80, ['Gimp', 'Butch']],
  'Gimp': [60, ['Marsellus']],
  'HoneyBunny': [50, ['Vincent']],
  'Pumpkin': [50, ['Vincent', 'Jules']],
  'Butch': [100, ['Marsellus', 'Vincent', 'Gimp']],
  'Bonnie': [25, []]
}

const shoot = (people, name) => Object.assign({}, people,
  { [name]: [people[name][0] - 5, people[name][1]] })
```

reduce part 2

applies a function against an accumulator and each element of a List

```
const people = {
  'Vincent': [120, ['Pumpkin']],
  'Butch': [100, ['Marsellus', 'Vincent', 'Gimp']]
}

const concatTargets = (acc, name) =>
  acc.concat(people[name][1])
const getTargets = (people) =>
  Object.keys(people).reduce(concatTargets, [])

const shootTargets = (targets, people) =>
  targets.reduce(shoot, people)

const targets = getTargets(people)
const nextState = shootTargets(targets, people)
```

bonus slide: generating the next n states

```
const shootThese = curry(shootTargets)(targets)
const repeatShoot = (n) => range(0, n)
  .reduce(shootThese, people)

range(0, 10).map(repeatShoot)
```

composition

chain function execution, passing the result of one function to the next one

```
const { compose, add } = require('ramda')

const add5multiply2 = compose(multiply(2), add(5))
assert.equal(add5multiply2(4), 18)
```

composition example

```
const { compose, map, split, match,
        tail, prop, sort, head } = require('ramda')
const source = `
"Donny Donowitz" is dead with 82 confirmed kills
"Wilhelm Wicki" is alive, 92 confirmed kills
"Aldo Raine" is still alive with 134 confirmed kills`

const parseLine = compose(
  ([name, status, kills]) =>
    ({ name, status, kills: Number(kills) }),
  tail,
  match(/"(.*)".*(dead|alive).* ([0-9]+).*/))

const findHighestKills = compose(
  prop('name'), head,
  sort(prop('kills')),
  map(parseLine),
  tail, split('\n'))
```

bonus slide: compose with reduce

compose can be trivially implemented with reduce over a list of functions

```
const composeFrom = (reduce) => (...fns) =>
  (...args) =>
    fns[reduce]((res, fn) =>
      [fn(...res)], args)[0]
const composeRight = composeFrom('reduceRight')
const composeLeft = composeFrom('reduce')
const compose = composeRight
```

lenses

make you set and view a value through a getter and setter function

```
const { lens, lensProp, view, set, over, add } = require(
const magnifier = lens((v) => v*2, (nv, v) => nv)
assert.equal(view(magnifier)(3), 6)

const lensAge = lens((v) => v['age'],
  (nv, v) => Object.assign({}, v, { age: nv }))
const lensAge2 = lensProp('age')
assert.equal(view(lensAge)({ age: 5 }), 5)
assert.deepEqual(set(lensAge, 6)({ age: 5 }), { age: 6 })
assert.deepEqual(over(lensAge, add(1))({age: 5}),{age: 6}
```

another lens example

```
const { lens, lensPath, over, add } = require('ramda')

const people = {
  'Vincent': [120, ['Pumpkin']],
  'Butch': [100, ['Marsellus', 'Vincent', 'Gimp']]
}
const shoot = (people, name) =>
  over(lensPath([name, 0]), add(-5))(people)
```

Fin

slides here: http://framp.me/fp-workshop/1

Huh, hold on..

Workshop exercise:

You have a test file containing a 4*N lines of text.
The first 3 lines of every group of lines contain 3*M chars which represent numbers grouped in 3x3 cells drawn with pipes and underscores.

```
 _     _  _     _  _  _  _  _
| |  | _| _||_||_ |_   ||_||_|
|_|  ||_  _|  | _||_|  ||_| _|

 _     _  _     _  _  _  _  _
| |  | _| _||_||_ |_   ||_||_|
|_|  ||_  _|  | _||_|  ||_| _|
```

Write a parser able to convert the test file in plain numbers.
You can find everything here: https://github.com/framp/fp-workshop

Try to apply the concepts and techniques learnt today.

Inspired/Flat out copied from CodingDojo <3

Bonus exercise for George

(and whoever completed the previous one)

Given a letter `l` between A-Z you want to print a diamond such as:

```
l=C
  A
 B B
C   C
 B B
  A
```

Inspired/Flat out copied from CodingDojo <3