fp concepts:

- keeps functions and data separate

- avoids state change and mutable data

- treats functions as first class citizens

- keeps functions and data separate

oop groups data and functions together

```
class Student {
    constructor (name, evtAttn) {
        this.name = name
        this.evtAttn = evtAttn
    }
    addEvtAttn() {
        this.evtAttn += 1
    }
}
```

- keeps functions and data separate

fp stores data in simple constructs/structures like hashes or arrays with separate functions taking data as a parameter returning transformed data

```
const Peter = {
    name: "Peter",
    evtAttn: 33
}

function addEvtAttn(who) {
    const whoCopy = _.cloneDeep(who)
    whoCopy.evtAttn += 1
    return whoCopy
}
```

- keeps functions and data separate

```
instant data polymorphism
addEvtAttn created using fp style applies to any data having an 'evtAttn' field
```

```
const iot = {type: 'FDP', evtAttn: 35, venue: 'SSN, Dept. of IT'}
```

```
const meera = {name: 'Meera', designation: 'faculty', evtAttn: 66}
```

```
const  addIotAttn = addEvtAttn(iot)
```

```
const addMeeraEvtAttn = addEvtAttn(meera)
```

```
how about the oop style ?
```

- avoids state change and mutable data

```
var venue = "Seminar Hall"
```

```
...
venue = venue + " of CSE Dept."
```

```
...
venue = venue.toUppercase()

console.log(venue)
```

difficult to keep track of all the state changes

soln:

make all the data immutable

create a new variable to represent each change

easier to debug

- avoids state change and mutable data

```
const venue = "Seminar Hall"
...
const cseVenue = venue + " of CSE Dept."
...
const cseVenueFormatted = cseVenue.toUpperCase()

console.log(cseVenueFormatted)
```

- treats functions as first class citizens

assign functions to variables

flexibility and code reusability

```
const a = 10
const b = "SSN"
const f1 = function() { .. }
```

pass functions as arguments to other functions

```
f2(function() { .. })
```

return function from function

```
function f3() {
    ...
    return function() { .. }
}
```

closure

can be used to implement private variables

fp vs oop

```
class AdmittedStudentsList {
    constructor(students) {
        this.students = students
    }

    admitStudent(student) {
        this.students.push(student)
    }
}
```

```
function admitStudent(list, student) {
    return list.concat(student)
}
```

```
class AppliedStudent {
    constructor(name, qualifyingMarks) {
        this.name = name
        this.qualifyingMarks = qualifyingMarks
    }
}
```

```
var testAdmittedStudentsList = new AdmittedStudentsList([
    new AppliedStudent("Mohan", 98),
    new AppliedStudent("Anand", 96)
])
```

```
const testAdmittedStudentsList = [
   {"Mohan", 98},
   {"Anand", 96}
]
```

```
testAdmittedStudentsList.admitStudent(
    new AppliedStudent("Rakesh", 88)
)
```

```
const admittedSportsRakesh = admitStudent(testAdmittedStudentsList, {"Rakesh", 88})
```

assign fns to variables

```
const greet = function() {
    console.log("Hello")
}
```

```
greet()
```

```
const sq = fucntion(x) {
    return x * x
}
```

```
sq(2) //4
```

fn variable with parenthesis

```
var 3sq = sq(3)
3sq //9
```

function variable without parenthesis

```
var tsq = sq
tsq(3) //9
```

assign fns to variables

normal fn. definition

```
a()                    //ok
function a() {
    ..
}
a()                    //ok
```

function assigned to variable

```
a()                    //undefined
function a() {
    ..
}
a()                    //ok
```

fns. may behave differently depending on state/conditions

assign functions to variables

```
// var printLine = console.log

// printLine("hello1")
```

pass functions as arguments to other functions

```
function multiply(x,y) {
    return x * y;
}

function fmultiply(x, y, fn) {
    return fn(x,y)
}

// console.log(fmultiply(10,10, multiply))
console.log(fmultiply(10,10, function(x,y) { return x + y}))
```

return function from function

```
// function frtn() {
//     return function() {
//         console.log("hello")
//     }
// }
```

// also you can return named and multiple functions

```
//closure
//--------
```

```
// function frtn() {
//     i = 10
//     return function() {
//         console.log(i)
//     }

// }

// x = frtn()
// // x()
// x().i
```

Map

```
// var _ = require('lodash')

// var n = [10,20,30,40]
// var doubleN = _.map(n, function(n) {
//     return n * 2
// })
```

Filter

```
// var _ = require('lodash')

// var n = [10,21,31,41]
// var oddN = _.filter(n, function(item) {
//     return (!(item % 2 === 0))
// })
```

Reduce

```
var _ = require('lodash')

var stock1 = [
    {name:'item1', count: 20, unitCost: 600},
    {name:'item2', count: 2, unitCost: 85},
    {name:'item3', count: 200, unitCost: 60},
    {name:'item4', count: 15, unitCost: 25}
]

var stock1Count = _.reduce(stock1, function(ret_val, item) {
    return ret_val + item.count
},0)

var stock1Cost = _.reduce(stock1, function(ret_val, item) {
    return ret_val + item.unitCost
},0)
```

Every & Some

```
var all_odd = _.every(n, function(item) {
    return (!(item % 2 === 0))
})
```

```
var some_odd = _.some(n, function(item) {
    return (!(item % 2 ===0))
})
```

Callback

```
console.log("Before ..")


setTimeout( function() {
  console.log("processed..")
}, 2000)


console.log("After ..")
```

Partial Application

```
function mul(a, b, c) {
    return a * b * c
}

function parMul(func, a) {
    return function(b, c) {
        return func(a, b, c)
    }
}
```

```
var mul10 = parMul(mul, 10)
```

```
console.log(mul10(2, 2))
```

Recursion

```
function countDown(k) {
    console.log(k)
    if (k > 0) {
      countDown(k - 1)
    }
}
countDown(10)
```