Functional Programming

#OddU 2020-06-24

Oddball

A strangely human digital agency

https://github.com/oddballteam/oddu-functional-programming

How is **Functional Programming** different than **Object Oriented?**

Oddball

A strangely human digital agency

What is Object Oriented Programming (OOP)

Imperative, Structure, and State Based

built using

Abstraction, Inheritance, Encapsulation, and Polymorphism

Looks For:

Logical Order, Hierarchy, and Representation

Thinking in OOP

What are the ‘Things’

and

How do I represent them

What is Functional Programming (FP)

Declarative Software

built using

‘Pure Functions’

Avoids:

Shared State, Mutable Data, and Side Effects

Thinking in FP

‘What am I doing’

not

‘How do I build it’

Change The Approach

Imperative Approach

I need a Cart,
The Cart need a Customer,
The Cart needs Products,
Products need Variations,
It should return a subtotal

vs

Declarative Approach

I should be able to add products to a list (cart),
I should be able to associate a list with a customer,
I should be able to total a given list of products,
I should be able to update a product on the list

Thinking in FP

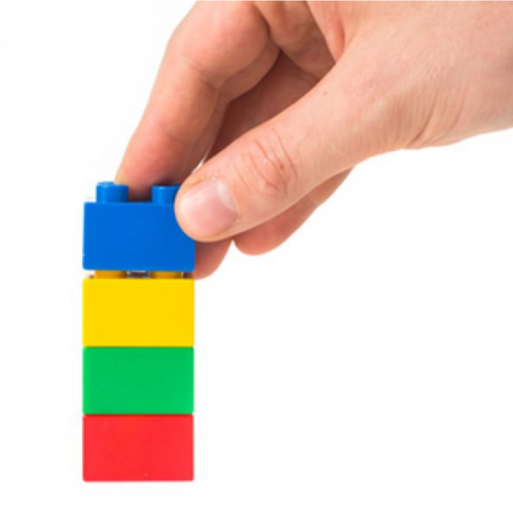
Do one thing, and do it well (tested)

doOneThingWell(withSomething);

This should always work because

it does not rely on external influences

Thinking in FP



Think small.

Reusable small ‘black boxes’ that return

testable values

passed to the next ‘black box’

Oddball
A strangely human digital agency

Terminology and Examples

https://github.com/oddballteam/oddu-functional-programming

Mutability

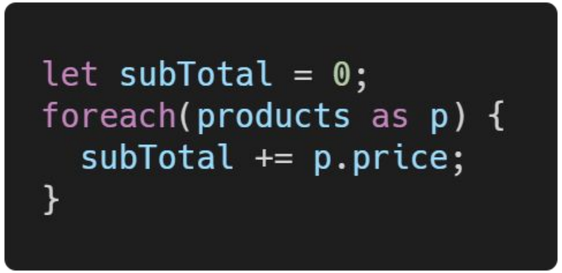
Mutables CAN be changed and can cause unpredictable results.

Immutables CAN’T be changed and are predictable/constant.

Avoid mutants!!!

Mutations are untrustworthy and unpredictable

Counters and loops are a smell test…



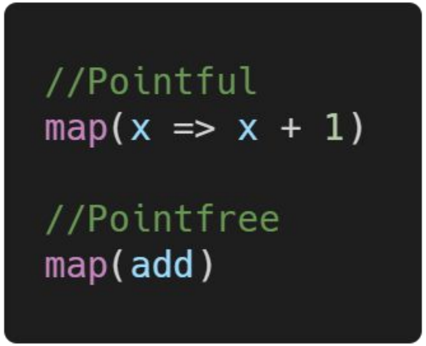
```
let subTotal = 0;
foreach(products as p) {
  subTotal += p.price;
}
```

Pure Functions

- A function where output is derived solely from it’s input.
- It must never modify external data or state.
- It will always work, and always return the same response with the same input regardless of environment, runtime, session, user state, etc.
- This makes it prime for testing!

Pointfree

- Pointfree code doesn't explicitly mention it's arguments, even though they exist and are being used.
- Pointful code does explicitly mention it's arguments, and how they are being used.



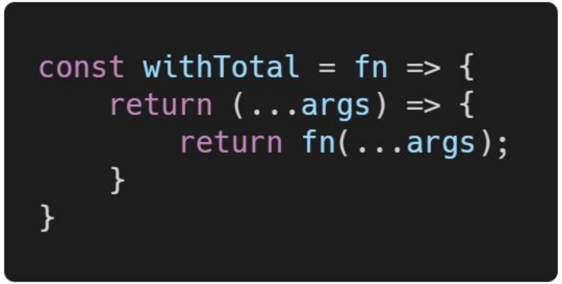
```
//Pointful
map(x => x + 1)

//Pointfree
map(add)
```

Point~~less~~free Example:

Oddball
A strangely human digital agency

Higher Order Functions

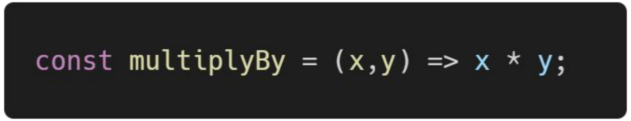
- Accepts function as argument
- Returns function



```
const withTotal = fn => {
    return (...args) => {
        return fn(...args);
    }
}
```

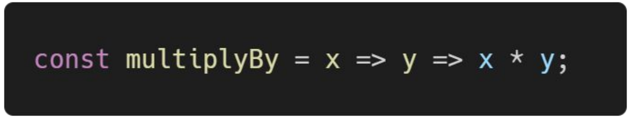
Currying

- To transform a function with multiple arguments into a sequence of nesting functions



```
const multiplyBy = (x,y) => x * y;
```

vs



```
const multiplyBy = x => y => x * y;
```

Partial Application

- A partial application is a function which has been applied to some, but not yet all of its arguments.

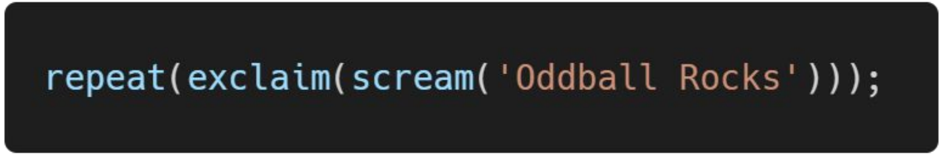
In other words, it’s a function which has some arguments *fixed* inside its closure scope. A function with some of its parameters fixed is said to be *partially applied*.

Didn’t get that? Don’t worry - demo is up next!

Oddball
A strangely human digital agency

Composition

- Combining multiple simple/pure functions to build more complex ones



```
repeat(exclaim(scream('Oddball Rocks')));
```

Ok… Now What?

First - Don’t be a purist.

Today:

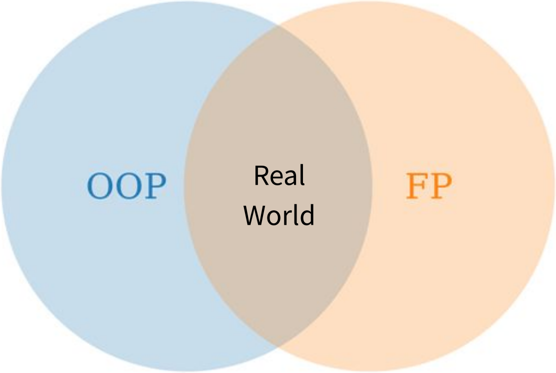
Write new code avoiding mutations

Later:

Start using Higher Order Functions and Currying

Eventually:

Write an app without state or classes



Oddball

A strangely human digital agency