

Programação Funcional

λ

Controlando complexidade & corretude

O que é programação funcional?

“Paradigma de programação onde computações são representadas por **funções** ou **expressões puras**, evitando **efeitos colaterais** e **dados mutáveis** e que utiliza amplamente **composição de funções** e **funções de primeira classe**”

Paradigma

Funções Puras

1. Determinística
2. Não existem efeitos colaterais

Determinística

Função Impura



```
let PI = 3.14;  
  
const calculateArea = (radius) =>  
  radius * radius * PI;
```

✓ Não existem efeitos colaterais

? Determinística

Imagine que $\pi = 42$

Para o mesmo parâmetro,
estamos retornando
resultados diferentes



```
let PI = 3.14;  
calculateArea(10); // 314.0  
  
let PI = 42;  
calculateArea(10); // 420
```

Função Pura



```
const calculateArea = (radius, pi) =>  
  radius * radius * pi;
```

✓ Não existem efeitos colaterais

✓ Determinística

Não existem efeitos colaterais

Função Impura



```
let counter = 1;
```

```
const increaseCounter = () => {  
  counter++;  
  return counter;  
}
```

```
counter; // 1  
increaseCounter(); // 2  
counter; // 2
```

✗ Determinística

? Não existem efeitos
colaterais

✓ Não existem efeitos colaterais

✓ Determinística

Função Pura

```
const counter = 1;

const increaseCounter = () =>
  counter + 1;

counter; // 1
increaseCounter(); // 2
counter; // 1
```

Alguns benefícios

0. Função previsível
1. Facilmente testável
2. Melhor entendimento da funcionalidade do código

Imutabilidade

0. Inalterável ao longo do tempo ou incapaz de ser alterado
1. Evitar mudanças de estado para reduzir complexidade

“OO makes code understandable by encapsulating moving parts. FP makes code understandable by minimizing moving parts.” - Michael Feathers

UrlSlugify

“ I will be a url slug ”



“i-will-be-a-url-slug”

0. lowercase
1. trim
2. replace

Mutabilidade em 3 níveis



```
class UrlSlugify
  attr_reader :text

  def initialize(text)
    @text = text
  end

  def slugify!
    text.downcase!
    text.strip!
    text.gsub!( ' ', '-' )
  end
end

UrlSlugify.new( ' I will be a url slug  ').slugify! # "i-will-be-a-url-slug"
```

Compondo funções s/ mutabilidade



```
const string = " I will be a url slug  ";
```

```
const slugify = string =>  
  string  
    .toLowerCase()  
    .trim()  
    .split(" ")  
    .join("-");
```

```
slugify(string); // i-will-be-a-url-slug
```

Compondo funções s/ mutabilidade



```
(defn slugify
  [string]
  (-> string
    (clojure.string/trim)
    (clojure.string/lower-case)
    (clojure.string/replace #" " "-")))

(slugify " I will be a url slug ") ;; "i-will-be-a-url-slug"
```

Funções como valor

**Poder de abstração: Retornando
funções como valor**

Curry? Closures?
Partial Application?



```
const add = arg1 =>  
  arg2 => arg1 + arg2;
```

```
const increment = add(1);  
increment(2); // 3
```

```
const add60 = add(60);  
add60(100); // 160
```


**Poder de abstração:
função como argumento**



```
(defn double-sum  
  [a b]  
  (* 2 (+ a b)))
```

```
(defn double-subtraction  
  [a b]  
  (* 2 (- a b)))
```



```
(defn double-operator  
  [f a b]  
  (* 2 (f a b)))
```

0. Recebe função como valor
1. Evalua função



```
(double-operator + 2 2) ;; 8  
(double-operator - 4 2) ;; 4
```

- Passa a função +
- Passa a função -

Funções de alta ordem

Funções de alta ordem

0. Funções que recebem função como argumento
1. Funções que retornam função como valor

Filter

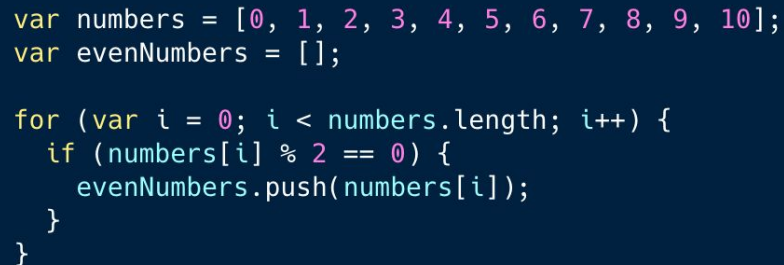
Map

Reduce

Filter

Filter

Dado uma lista de números,
queremos filtrar os pares.



```
var numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10];  
var evenNumbers = [];  
  
for (var i = 0; i < numbers.length; i++) {  
  if (numbers[i] % 2 == 0) {  
    evenNumbers.push(numbers[i]);  
  }  
}
```

Filter

Usando função filter para filtrar os números pares de uma maneira declarativa



```
const even = n => n % 2 == 0;  
const ns = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10];  
ns.filter(even); // [0, 2, 4, 6, 8, 10]
```

Map

Map

Pensando em
transformação de listas



```
people = [  
  { name: 'TK', age: 26 },  
  { name: 'Kaio', age: 10 },  
  { name: 'Kazumi', age: 30 }  
]
```

Map

Dado uma lista de pessoas com nome e idade, queremos uma lista de sentenças.


```
const people = [
  { name: "TK", age: 26 },
  { name: "Kaio", age: 10 },
  { name: "Kazumi", age: 30 }
];

var sentences = [];

for (var i = 0; i < people.length; i++) {
  const sentence = people[i].name + " is " + people[i].age;
  sentences.push(sentence);
}
```

Map

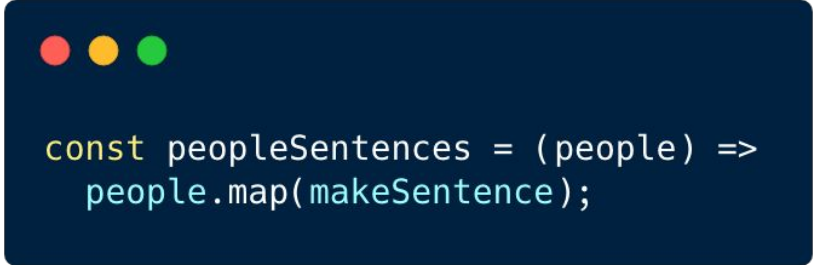
Pausa para o refactoring...



```
const makeSentence = (person) =>  
  `${person.name} is ${person.age} years old`;
```

Map

Usando função de alta ordem,
programação declarativa e
imutabilidade




```
const peopleSentences = (people) =>  
  people.map(makeSentence);
```

Reduce

Reduce

Juntar uma coleção de elementos
e colocar em um “produto final”



```
const orders = [  
  { productTitle: "Product 1", amount: 10 },  
  { productTitle: "Product 2", amount: 30 },  
  { productTitle: "Product 3", amount: 20 },  
  { productTitle: "Product 4", amount: 60 }  
];
```

Reduce

“produto final” == soma de todos
os elementos da lista

(Pensando imperativamente)

```
const orders = [
  { productTitle: "Product 1", amount: 10 },
  { productTitle: "Product 2", amount: 30 },
  { productTitle: "Product 3", amount: 20 },
  { productTitle: "Product 4", amount: 60 }
];

let totalAmount = 0;

for (var i = 0; i < orders.length; i++) {
  totalAmount += orders[i].amount;
}

console.log(totalAmount); // 120
```

Reduce

Usando reduce para gerar o
“produto final”


```
const shoppingCart = [
  { productTitle: "Product 1", amount: 10 },
  { productTitle: "Product 2", amount: 30 },
  { productTitle: "Product 3", amount: 20 },
  { productTitle: "Product 4", amount: 60 }
];

const sumAmount = (currentTotalAmount, order) =>
  currentTotalAmount + order.amount;


const getTotalAmount = (shoppingCart) =>
  shoppingCart.reduce(sumAmount, 0);

getTotalAmount(shoppingCart); // 120
```

Probleminha




```
const stock = {  
  'coffee': {  
    label: '3 corações'  
    count: 7,  
  },  
  'water': {  
    label: 'Bonafont'  
    count: 2,  
  },  
  'juice': {  
    label: 'Ades'  
    count: 3,  
  }  
};  
  
const beverages = ['coffee', 'water', 'juice'];
```



```
const stock = {
  'coffee': {
    label: '3 corações'
    count: 7,
  },
  'water': {
    label: 'Bonafont'
    count: 2,
  },
  'juice': {
    label: 'Ades'
    count: 3,
  }
};

const beverages = ['coffee', 'water', 'juice'];
```



```
const buildBeverages = (beverages, stock) =>
  beverages.map(
    itemType => do(
      stock[itemType].label,
      stock[itemType].count
    )
  );
```



```
const getStockItem = stock =>  
  itemType => do(  
    stock[itemType].label,  
    stock[itemType].count  
  );
```

```
const buildBeverages = (beverages, stock) =>  
  beverages.map(getStockItem(stock));
```

Composição

“The act of breaking a complex problem down into smaller problems, and composing simple solutions to form a complete solution to the complex problem.”

Composição de funções

- Função como unidade que resolve um pequeno problema
- Composição de funções que resolvem problemas complexos

Composição de funções

Aplicar uma função no resultado de outra função

$$\begin{aligned} &\rightarrow f() \rightarrow g() \rightarrow \\ &(g \circ f)(x) \\ &g(f(x)) \end{aligned}$$

- Pomo Lambda
- Luhn Algorithm
- Reselect

Controlando complexidade

com programação funcional

0. Decomposição de problemas complexos
1. Abstração de estruturas de controle de fluxo
2. Determinístico. Completamente previsível
