

EXISTE UM MUNDO
ALÉM DE OOP E
PROCEDURAL

@mracos

**OU SOBRE PROGRAMAÇÃO
FUNCIONAL**

PARADIGMAS

(DE PROGRAMAÇÃO)

- são as diferentes formas de "pensar" em programação

PARADIGMAS

PROCEDURAL (IMPERATIVO)

- receitinha de bolo
- um passo a passo

linguagens: Pascalzin, Ctop

PARADIGMAS

PROCEDURAL

```
customer = {name: "marcos", balance: 100, products: []}  
seller = {name: "julia", balance: 0, product: :water}  
  
# saque  
customer_money = 10  
customer[:balance] -= customer_money  
  
# compra  
seller[:balance] += customer_money  
customer[:products] << seller[:product]
```

PARADIGMAS

OOP (ORIENTAÇÃO A OBJETO)

- objetos que tentam "abstrair" a vida real
- estado (atributos) e ações (métodos) com objetos interagindo consigo

linguagens: java, python, C++, JavaScript, PHP, Ruby, etc.

PARADIGMAS

OOP

```
customer = Customer.new("marcos", 100)
seller = Seller.new("julia", 0, :water)

seller.set_product_price(10)
customer.buy(seller)
```

PARADIGMAS

- declarativo
- lógico
- simbólico
- ...

PARADIGMAS

MISTO (N DE QUEIJO)

- maioria das linguagens na real

PARADIGMA FUNCIONAL

FUNCIONAL

- lambda cálculo (modelo teórico) (1930)
- lisp e serviu de inspiração para linguagens mais novas
- IPL |> APL |> FP |> ML |> ... |> Haskell (foi tipo uma implementação referência)
- bastante embasamento acadêmico/matemático
 - type theory (em linguagens fortemente tipadas)

FUNCIONAL

- trata a computação como uma avaliação de funções matemáticas
 1. tenta chegar mais perto do conceito acadêmico de funções
 2. modelagem matemática
 - função identidade: $f(x) = x$
 - função quadrática: $f(x) = x^2$

FUNCIONAL

- enfatiza a aplicação de funções
 - ou seja, a tua aplicação/sistema funciona operando em cima de dados com funções trabalhando em conjunto
 - maior enfoque em **transformações de dado** ao invés de **mudança de estado**

clone_com_peruca(pessoa)

o
/|\
/ \

->

V-----V
| função de |
| clonagem |
| + peruca |
V-----V

o
/|\
/ \
^

+

\|/
/o\
/|\
/ \

<-----

retorna nov
com peruca

|_ antigo se mantém

FUNCIONAL

- imutabilidade
- pureza de funções
- aridade
- lazy/strict evaluation

FUNCIONAL

IMUTABILIDADE

- enfatiza não manter estado ou dados mutáveis (!!)
- ### 1. imutabilidade

```
defmodule Example do
  def add_four_to_array(array) do
    array ++ [4]
  end
end

arr = [1, 2, 3]
new_arr = Example.add_four_to_array(arr)

IO.puts(arr) # [1, 2, 3]
IO.puts(new_arr) # [1, 2, 3, 4]
```


FUNCIONAL

FUNÇÕES PURAS

funções não deveriam ter efeitos colaterais

1. "imutabilidade" no resultado (resultado só depende dos argumentos passados)

```
sum = fn (x, y) -> x + y end  
sum.(1, 2) # SEMPRE vai ser 3
```

```
sum_with_api = fn (x, y) -> x + get_y_from_api(y) end  
sum.(1, 2) # não me dá certeza, pode ter erro etc...
```

FUNCIONAL

- acaba sendo mais declarativo
 - gente define pequenas funções que a gente sabe exatamente o que faz

FUNCIONAL

ARIDADE

quantos parâmetros uma função tem

- `Example.add_four_to_array(array)` tem aridade de 1
- `Example.add_x_to_array(array, x)` aridade seria 2

Se escreve como

`Example.add_four_to_array/1`

FUNCIONAL

STRICT/LAZY EVALUATION

1. avaliação preguiçosa / avaliação apressada (lazy evaluation)
 - **lazy evaluation:** Só computa o valor/código se realmente necessário
 - **strict evaluation:** Computa o valor de qualquer forma

FUNCIONAL

STRICT/LAZY EVALUATION

```
# demora um senhor tempo
list = 1..1000000
filtered_list = Enum.filter(list, &(rem(&1, 2)))
Enum.take(filtered_list, 2) # [2, 4]

# não demora nada
lazy_list = 1..1000000
filtered_lazy_list = Stream.filter(lazy_list, &(rem(&1, 2) == 0))
Enum.take(filtered_lazy_list, 2) # [2, 4]
```

FUNÇÕES

- como se comportam em um paradigma funcional?

FUNÇÕES

- lambdas (funções anônimas)
(lambda lambda jovem nerdsss)
- closures

FUNÇÕES

LAMBDA

são funções sem um nome

```
# funções não anônimas
defmodule Multiply do
  def by_two(value) do
    value * 2
  end
end
```

```
Enum.map([1, 2, 3], &Multiply.by_two/1) # [2, 4, 6]
```

```
# funções anônimas
Enum.map([1, 2, 3], fn x -> x * 2 end) # [2, 4, 6]
```


FUNÇÕES

CLOSURES

funções que mantêm o escopo em que foram definidas (!!)

```
x = 10  
sum_ten = fn (y) -> x + y end
```

```
sum_ten.(10) # 20
```

```
x = 20  
sum_ten.(10) # 10
```

FUNÇÕES

CLOSURES

- a função foi definida quando x era 10
- mudou a definição de x para 20
- a função continuou com a definição de quando x era 10
 - Porque ela é uma closure
- consistência irmãos!

FUNÇÕES

- **first class** (funções de primeira classe)
- **funções de alta ordem** (high order)
- **curry**

FUNÇÕES

FIRST CLASS FUNCTIONS

funções podem ser guardadas em variáveis,
retornadas de funções e passadas como parâmetro

```
# podem ser guardadas em variáveis
multiply_by_two = fn (x) -> x * 2 end
multiply_by_two.(2) # 4

# podem ser passadas como argumento
multiply_by = fn (x, function) -> function.(x) end
multiply_by.(2, &multiply_by_two/1) # 4
```

FUNÇÕES

HIGH ORDER FUNCTIONS

- é uma função que retorna outra função

```
hello_to_name = fn
  (name) -> fn -> "Hello, #{name}" end
end

greet_marcos = hello_to_name("Marcos")
greet_marcos.() # Hello, Marcos
```

FUNÇÕES

CURRY (NÃO É A COMIDA RS)

Uma função que retorna outra função de menor aridade com algum argumento já "*preenchido*"

```
print_if_debug = fn
  (value, debug) -> if (debug), do: IO.puts(value)
end

always_print = fn (value) -> print_if_debug.(value, true) end

print_if_debug.(123, true) # 123
print_if_debug.(123, false) # nil

always_print.(123) # 123
```

FUNÇÕES

PATTERN MATCH

Um dado que "casa" com o outro

- = é um operador de pattern match

```
variavel = 4  
4 = variavel
```

```
[1, 2, 3, 4] = [1, 2, 3, variavel]  
[primeiro, 2, 3] = [1, 2, 3]
```

```
# só é chamada caso o primeiro parâmetro seja 1  
sum_only_with_one = fn (1, y) -> 1 + y end
```

```
#  
sum_only_with_one.(1, 2) # 3  
sum_only_with_one.(2, 3) # erro, função não encontrada
```

FUNÇÕES

PATTERN MATCH

Funções podem ser definidas com base nos seus parâmetros

```
print_if_debug = fn (value, true) -> IO.puts(value)

print_if_debug.(123, true) # 123
print_if_debug.(123, false) # Erro, função não encontrada
```


FUNCTIONAL PROGRAMMING?

FUNCTIONAL PROGRAMMING?

THE GOOD FELLAS

- imutabilidade
 - sem surpresas
 - mais explícito

FUNCTIONAL PROGRAMMING?

THE GOOD FELLAS

- sucinta, mais fácil de ler
- de grátis building blocks para paralelismo | concorrência
- "força" composibilidade (de funções)
- geralmente devs "melhores" (buscam além do que precisam)
 - código mais maintainable

FUNCTIONAL PROGRAMMING?

THE BAD FELLAS

- outra forma de pensar -> curva de aprendizado
- alta memória
- outra format de pensar
- (pra algumas coisas) too much mathy

LHES APRESENTO

ELIXIR

(eu tentei)

Não é sobre elixir, é sobre os conceitos lá atrás :)

ELIXIR

PQ?

1. pq?

1. feita na vm da erlang (tolerância a falhas de grátis, concorrência, maturidade)
2. feita por um br !!!!!
3. tooling MUITO bom (ferramentas que permeiam a linguagem)
4. sintaxe (só sintaxe) inspirado em ruby (<3)
5. comunidade mt top

Questions?

Dicas

- *prependar* é sempre preferível a *appendar* uma lista
- listas em elixir/erlang são linked lists, ou seja, adicionar no início da lista é sempre $O(1)$ (mais rápido) que no final $O(n)$ (velocidade depende do tamanho da lista)
- dora o lance de ser imutável e ter que copiar toda a lista pq foi no final

énós! 

LEARN MORE

- <https://elixir-lang.com>
- <https://elixirschool.com/pt/>
- <https://elixirforum.com/>

- talk jose valim
- <https://medium.com/making-internets/functional-programming-elixir-pt-1-the-basics-bd3ce8d68f1b>
- <https://medium.com/@cameronp/functional-programming-is-not-weird-you-just-need-some-new-patterns-7a9bf9dc2f77>
- <https://speakerdeck.com/arthurbragaa/introducao-a-programacao-funcional-com-elixir>

