Práticas de Desenvolvimento de Software

#

Aula 05

Paradigmas e linguagens de programação

If all you have is a hammer, everything looks like a nail.

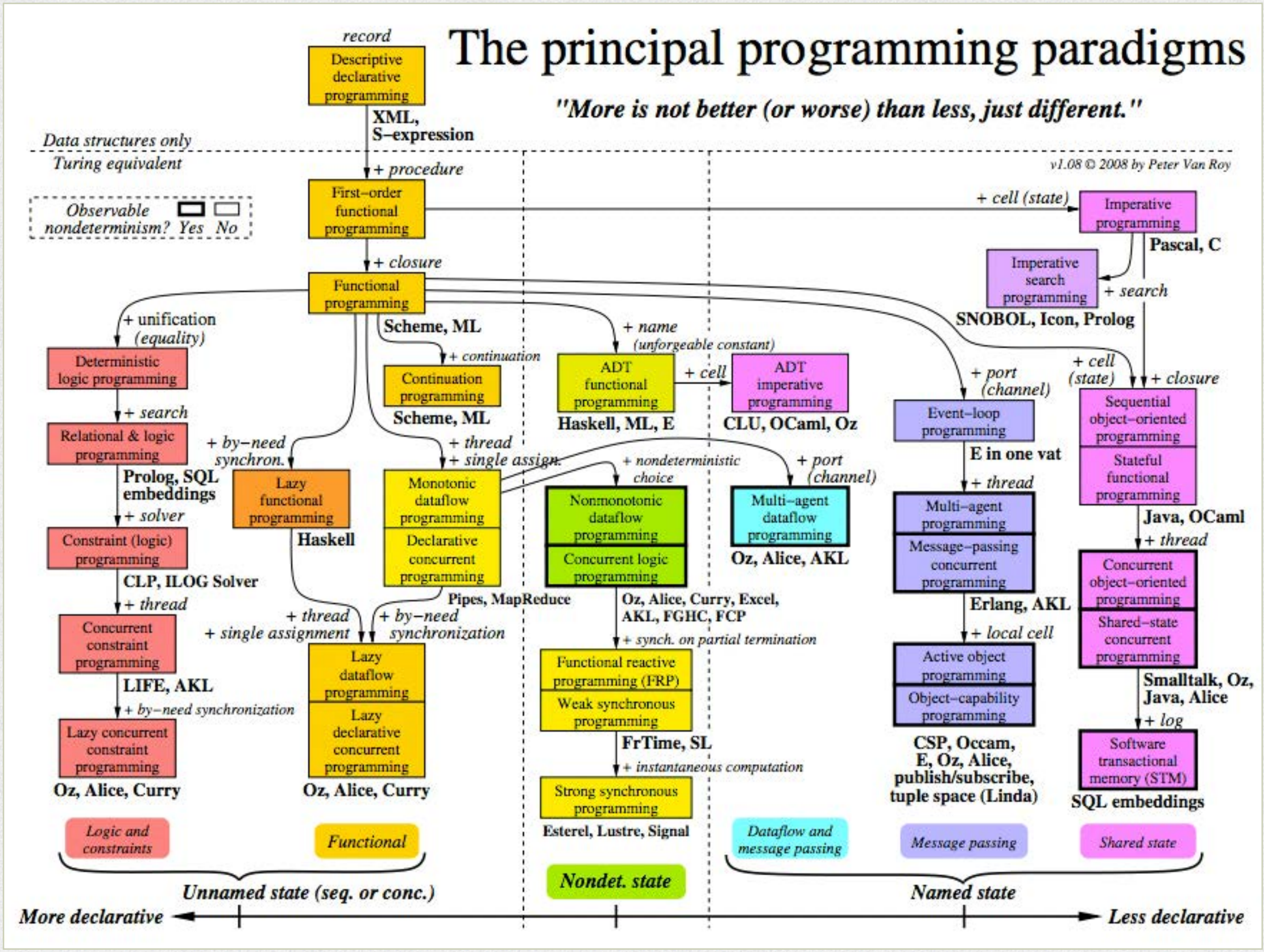
http://www.flickr.com/photos/juniorvelo/4490511204

Paradigma de programação

"Um paradigma de programação é uma forma de conceituar o que significa realizar computação e como tarefas a serem realizadas no computador devem ser estruturadas e organizadas."

Timothy A. Budd, 1995 **(tradução livre)**

QUANTOS PARADIGMAS EXISTEM?



Concepts, Techniques, and Models of Computer Programming, Peter Van Roy, 2008.

QUANTOS PARADIGMAS EXISTEM?

a) Nenhum — Existem formas melhores de analisar linguagens de programação

b) 3 — Três principais — todos os outros são derivados destes

c) 4 — Perspectiva adotada nesta aula

d) 35 — Peter Van Roy (número aproximado)

e) 49 — Wikipedia (número aproximado)

OS 4 PARADIGMAS

- Imperativo
- Orientado a objetos
- Lógico
- Funcional

PARADIGMA IMPERATIVO

Sequência ordenada de passos que alteram o estado do programa

FIBONACCI.C (C)

```c
int fibonacci(int n) {
  int previous = 0, current = 0, i, tmp;

  for(i = 0; i < n; i++) {
    if (i == 0) {
       current = 1;
    }
    else {
      // Fib(n) = Fib(n-1) + Fib(n-2)
      tmp = current;
      current = current + previous;
      previous = tmp;
    }
  }

  return current;
}
```

FIBONACCI.C (C)

```c
int fibonacci(int n) {
  int previous = 0, current = 0, i, tmp;

  for(i = 0; i < n; i++) {
    if (i == 0) {
      current = 1;
    }
    else {
      // Fib(n) = Fib(n-1) + Fib(n-2)
      tmp = current;
      current = current + previous;
      previous = tmp;
    }
  }

  return current;
}
```

→ Subrotinas

→ Variáveis

→ Controle de fluxo (loop)

→ Controle de fluxo (if)

→ Mudança de estado (atribuição)

- Ordem determinada pelo programador
- Compilador tem poucas oportunidades de otimização

Visão geral

→ Sequência de passos que alteram o estado do programa

→ Manter a ordem dos passos é essencial

→ Modelo muito próximo do hardware

Construções

→ Variáveis

→ Atribuições

→ Controle de fluxo (if, while, for, etc)

→ Sub-rotinas

Características

→ Eficiente

→ Familiar

→ Difícil de entender e provar (formalmente)

→ Fácil de deixar desorganizado e difícil de depurar

Linguagens

→ Assembly, Fortran, COBOL, BASIC, Pascal, C

PARADIGMA ORIENTADO A OBJETOS

Objetos interagem através da passagem de mensagens e alteram seu estado interno

FIBONACCI.RB (RUBY)

```ruby
class Math

  def self.fibonacci(n)
    current, previous = 0, 0

    for i in 0...n do
      if i == 0
        current = 1
      else
        current, previous = current + previous, current
      end
    end

    current
  end

end
```

Importante: código não-idiomático!

FIBONACCI.RB (RUBY)

```ruby
class Math

  def self.fibonacci(n)
    current, previous = 0, 0

    for i in 0...n do
      if i == 0
        current = 1
      else
        current, previous = current + previous, current
      end
    end

    current
  end

end
```

Classes

Métodos

Variáveis (podem de instância)

Controle de fluxo (loop)

Controle de fluxo (if)

Mudança de estado

Olhando "de perto", é um código imperativo, porém mais estruturado e com os benefícios da modularização e encapsulamento.

Aviso: código não-idiomático

Visão geral

→ Objetos interagem através da passagem de mensagens

→ Cada objeto possui seu próprio estado (variáveis) e comportamento que define como alterar o estado (métodos)

→ Modelo mais próximo do mundo real

Construções

→ Classes

→ Objetos (instâncias)

→ Hierarquia de classes

→ Passagem de mensagens

Características

→ Modularidade e encapsulamento

→ Código reutilizável e extensível

→ Difícil de entender e provar (formalmente)

→ Programas normalmente maiores (LOC)

→ Difícil de paralelizar e otimizar

Linguagens

→ Smalltalk, C++, Objective-C, Perl, Python, Java, Ruby, C#

PARADIGMA LÓGICO

Encontrar respostas a partir de fatos e regras de inferência

STARK-FAMILY.PL (PROLOG)

```
parent(eddard, arya).
parent(eddard, sansa).
parent(eddard, rickon).
parent(eddard, bran).
parent(eddard, robb).
parent(eddard, jon).

sibling(X,Y) :-
    parent(P,X), parent(P,Y), X\=Y.

child(C) :-
    parent(P,C).

%% Find all children
child(C).
%% Are Arya and Sansa siblings?
sibling(arya,sansa).
%% Are Arya and Lord Eddard siblings?
sibling(eddard,arya).
```

STARK-FAMILY.PL (PROLOG)

```
parent(eddard, arya).
parent(eddard, sansa).
parent(eddard, rickon).
parent(eddard, bran).
parent(eddard, robb).
parent(eddard, jon).
```

→ Fatos

```
sibling(X,Y) :-
   parent(P,X), parent(P,Y), X\=Y.

child(C) :-
   parent(P,C).
```

→ Regras de inferência

```
%% Find all children
child(C).
%% Are Arya and Sansa siblings?
sibling(arya,sansa).
%% Are Arya and Lord Eddard siblings?
sibling(eddard,arya).
```

→ Queries (pesquisas)

FIBONACCI.PL (PROLOG)

```
fib(0, 0).
fib(1, 1).
fib(N, NF) :-
    A is N - 1, B is N - 2,
    fib(A, AF), fib(B, BF),
    NF is AF + BF.
```

```
?- fib(10, X).
```

FIBONACCI.PL (PROLOG)

```
fib(0, 0).
fib(1, 1).
```

→ Fatos

```
fib(N, NF) :-
    A is N - 1, B is N - 2,
    fib(A, AF), fib(B, BF),
    NF is AF + BF.
```

→ Regra de inferência

```
?- fib(10, X).
```

→ Query

- Linguagem declarativa (!= imperativa)
- Ordem das declarações não importa
- O mecanismo de busca faz parte da linguagem

Visão geral

→ Baseado em lógica formal

→ Programa declarativo (foco em **o quê**, ao invés de **como**)

→ Uso em Inteligência Artificial e provas automáticas de teoremas

Construções

→ Declaração de fatos e regras

→ Mecanismo (interno) de busca

Características

→ Fácil de verificar formalmente

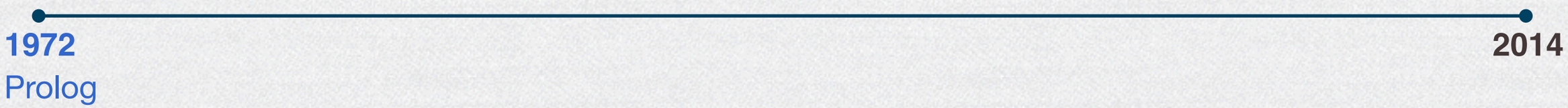
→ Fácil de paralelizar e otimizar

→ Estrutura não-familiar

→ Implementações não são padronizadas

Linguagens

→ Prolog



PARADIGMA FUNCIONAL

Sucessão de transformações (aplicação e redução de funções)

FIBONACCI.SCM (SCHEME)

```
(define (fib n)
  (if (< n 2) n (+ (fib (- n 1))
                   (fib (- n 2)))))


(fib 4)
```

FIBONACCI.JS (JAVASCRIPT)

```
fib = function(n) {
  fibInternal = function(current, next, remaining) {
    if (remaining == 0) {
      return current;
    }
    else {
      return fibInternal(next, current + next, remaining - 1);
    }
  };
  return fibInternal(0, 1, n);
}
```

FIBONACCI.JS (JAVASCRIPT)

```
fib = function(n) {
  fibInternal = function(current, next, remaining) {
    if (remaining == 0) {
      return current;
    }
    else {
      return fibInternal(next, current + next, remaining - 1);
    }
  };
  return fibInternal(0, 1, n);
}
```

Atribuição de função

Funções internas

Recursão (tail recursion, neste caso)

- Ausência de variáveis

FIBONACCI.JS (JAVASCRIPT) — Recursão simples

```
fib = function(n) {
  if (n < 2) {
    return n;
  }
  else {
    return fib(n-1) + fib(n-2);
  }
};
```

```
fib(4)
fib(3) + fib(2)
fib(2) + fib(1) + fib(2)
fib(1) + fib(0) + fib(1) + fib(2)
1 + 0 + 1 + fib(2)
1 + 0 + 1 + fib(1) + fib(0)
1 + 0 + 1 + 1 + 0
3
```

```
fib(3): 1x
fib(2): 2x
fib(1): 3x
fib(0): 2x
```

→ Ineficiente!

FIBONACCI.JS (JAVASCRIPT) — Tail recursive

```
fib = function(n) {
  fibInternal = function(current, next, remaining) {
    if (remaining == 0) {
      return current;
    }
    else {
      return fibInternal(next, current + next, remaining - 1);
    }
  };
  return fibInternal(0, 1, n);
}
```

```
fib(4)
fibInternal(0, 1, 4) + fibInternal(x, y, z)
fibInternal(1, 1, 3)
fibInternal(1, 2, 2)
fibInternal(2, 3, 1)
fibInternal(3, 5, 0)
3
```

- Não é necessário fazer redução
- Mais eficiente no tempo e no espaço

Visão geral

→ Baseado no cálculo lambda

→ Aplicação e redução de funções

→ Ordem de execução implícita

Construções

→ Recursão ao invés de iteração

→ Funções como construções de “primeira classe”

→ Funções de ordem superior

→ Ausência de variáveis (estado) e controle de fluxo (ordem de execução)

Características

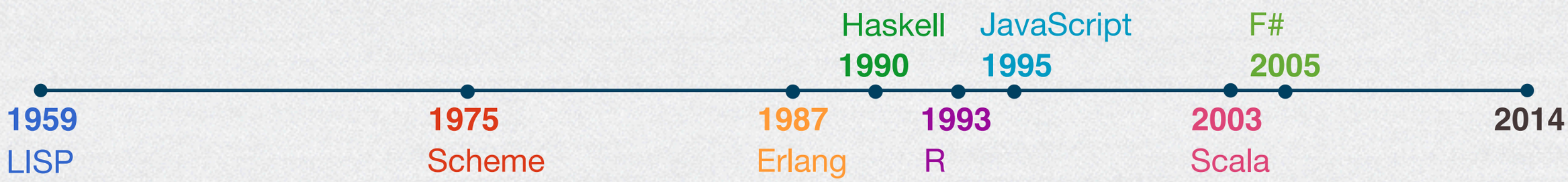
→ Fácil de verificar formalmente

→ Fácil de paralelizar e otimizar

→ Menos eficientes (?)

Linguagens

→ Lisp, Scheme, Erlang, Haskell, R, JavaScript, Scala, F#



FUNÇÕES DE "PRIMEIRA CLASSE"

⇾ Associar um identificador a uma função

⇾ Armazenar uma função em uma estrutura de dados (ex: lista)

RUBY

```ruby
greet = ->(name) { puts "Hi, #{name}!" }
greet["Jane"]
```

Lambda

JAVASCRIPT

```javascript
greet = function(name) { console.log("Hi, " + name + "!"); }
greet("Jane");
```

SCHEME

```scheme
(define greet (lambda (name) (print "Hi, " name "!")))
(greet "Jane")
```

FUNÇÕES DE ORDEM SUPERIOR

→ Passar uma função como argumento para outra função

→ Retornar uma função

RUBY

```ruby
def print_name(name)
  -> { puts name }
end
```

```ruby
print_name("Jane")[]
```

JAVASCRIPT

```javascript
function printName(name) {
   return function() { console.log(name); }
}
```

```javascript
printName("Jane")();
```

SCHEME

```scheme
(define print-name
   (lambda (name)
     (lambda () (print name))))
```

```scheme
((print-name “Bazinga"))
```

LAMBDAS E CLOSURES

⇾ Lambda: função anônima

⇾ Closure: função + ambiente (variáveis)

Lambda em Ruby

```ruby
def print_name(name)
  -> { puts name }
end
```

Closure em Ruby

```ruby
def print_name(first_name, last_name)
  full_name = "#{first_name} #{last_name}"
  -> { puts full_name }
end
```

`full_name` ✖

`print_name("Jane", "Doe")[]` ✔

A sua linguagem de programação consegue fazer isso?

Ou: a importância de construções funcionais

Baseado em: http://www.joelonsoftware.com/items/2006/08/01.html

Versão 1: Imperativa

LIVE

```ruby
def find_odd_numbers(arr = [])
  result = []
  for elem in arr do
    if elem % 2 != 0
      result << elem
    end
  end
  result
end

def find_even_numbers(arr = [])
  result = []
  for elem in arr do
    if elem % 2 == 0
      result << elem
    end
  end
  result
end

list = [1, 2, 3]
puts "1st approach: imperative style"
puts "Odd: #{find_odd_numbers(list)}"
puts "Even: #{find_even_numbers(list)}"
```

Versão 2: Passagem de funções como parâmetro

LIVE

```ruby
def is_odd?(x)
  x % 2 != 0
end

def is_even?(x)
  x % 2 == 0
end

def find_numbers(arr = [], filter_function)
  result = []
  for elem in arr do
    if filter_function.call(elem)
      result << elem
    end
  end
  result
end

list = [1, 2, 3]
puts "2nd approach: passing named functions as arguments"
puts "Odd: #{find_numbers(list, lambda(&method(:is_odd?)))}"
puts "Even: #{find_numbers(list, lambda(&method(:is_even?)))}"
```

Versão 3: Passagem de lambdas como parâmetro

LIVE

```ruby
def find_numbers(arr = [], filter_function)
  result = []
  for elem in arr do
    if filter_function.call(elem)
      result << elem
    end
  end
  result
end

list = [1, 2, 3]
puts "3rd approach: passing anonymous functions as arguments"
puts "Odd: #{find_numbers(list, ->(x) { x % 2 != 0 })}"
puts "Even: #{find_numbers(list, ->(x) { x % 2 == 0 })}"
```

Versão 4: Passagem de blocos para o método find_all

LIVE

```
puts "4th approach: passing lambdas to Ruby's default method"
puts "Odd: #{[1, 2, 3].find_all { |x| x % 2 != 0 }}"
puts "Even: #{[1, 2, 3].find_all { |x| x % 2 == 0 }}"
```

Vantagens da versão funcional:

⇾ Código mais legível

⇾ Código menor

⇾ Código mais declarativo (maior abstração)

⇾ Código mais paralelizável

⇾ Permite implementações como MapReduce, um dos modelos de programação mais importantes e inovadores da última década

Conclusão

⇀ O importante é não ser fundamentalista

⇀ Computacionalmente, todas as linguagens populares são equivalentes

⇀ Existem outros fatores mais importantes e não-técnicos na hora da escolha de uma linguagem:

Classes de problemas mais comuns

Familiaridade

Bibliotecas

Comunidade

Mercado (empresa e funcionário)

Legibilidade

Produtividade

EXERCÍCIOS!

Baixar a pasta **Exercises** (Week 4) do repositório da Infosimples, copiar esta pasta para o seu repositório (DevSoft2014-01) e resolver os 4 exercícios.

Entrega pelo GitHub, as usual.