Linguagens de Programação

Fabio Mascarenhas - 2017.2

http://www.dcc.ufrj.br/~fabiom/lp

Herança

- Para ter recursão aberta vamos introduzir uma forma implícita de delegação, a *herança de implementação*
- Um objeto vai poder extender outro objeto, o seu *protótipo*; ele ganha o mesmo número de campos do protótipo, e pode ter campos adicionais
- Se não achamos um método no objeto então continuamos a busca no seu protótipo
- Uma vez encontrado o método a chamada é feita normalmente

```
fun counter(n)
  object (n)
    fun inc(n)
      @0 := @0 + n
    end
    fun dec(n)
      self.inc(-n)
    end
  end
end
fun hcounter(o)
  object () extends o
    fun inc(n)
      @0 := @0 + n * 2
    end
  end
end
let c1 = counter(0),
    c2 = hcounter(c1) in
  print(c1.inc(2));
  print(c2.dec(3))
end
```

`super`

- A chamada de um método tem duas partes: buscar o método e a chamada em si
- Se começarmos a busca no protótipo do objeto, mas fizermos a chamada com `self` sendo o próprio objeto, temos o comportamento de `super` nas linguagens OO
- `super` delega a implementação para o protótipo, mas mantém a recursão aberta; o programa ao lado imprime -6
- SELF e JavaScript implementam modelos OO parecidos com os de *proto*

```
fun counter(n)
  object (n)
    def inc(n)
      @0 := @0 + n
    end
    def dec(n)
      self.inc(-n)
    end
  end
end
fun hcounter(o)
  object () extends o
    def inc(n)
      super.inc(n*2)
    end
  end
end
let c1 = counter(0),
    c2 = hcounter(c1) in
  print(c1.inc(2));
  print(c2.dec(3)) -- -6
end
```

Objetos são de valores alta ordem

- Um objeto carrega a implementação de seus métodos consigo
- Programar com objetos se parece mais com a programação usando funções de primeira classe do que a programação imperativa tradicional, que usa apenas funções de primeira ordem
- Funções de alta ordem, tipos algébricos, casamento de padrões, tudo isso pode ser simulado em *proto* usando objetos sem nem mesmo usar herança

```
fun vazia()
  object ()
    fun imprime() 0 end
    fun map(f) self end
  end
end

fun cons(h, t)
  object (h, t)
    fun imprime()
      print(@0); @1.imprime()
    end
    fun map(f)
      cons(f.apply(@0), @1.map(f))
    end
  end
end

let l1 = cons(1, cons(2, vazia())),
    l2 = l1.map(object ()
                  fun apply(o) o * o end
                end) in
  l1.imprime(); l2.imprime()
end
```

Recursão aberta, de novo

- Herança e recursão aberta dão mais expressividade
- No programa ao lado, o a função tracer constrói uma versão da “função” f que imprime seu argumento
- Com a recursão aberta, mesmo chamadas recursivas têm seus argumentos impressos

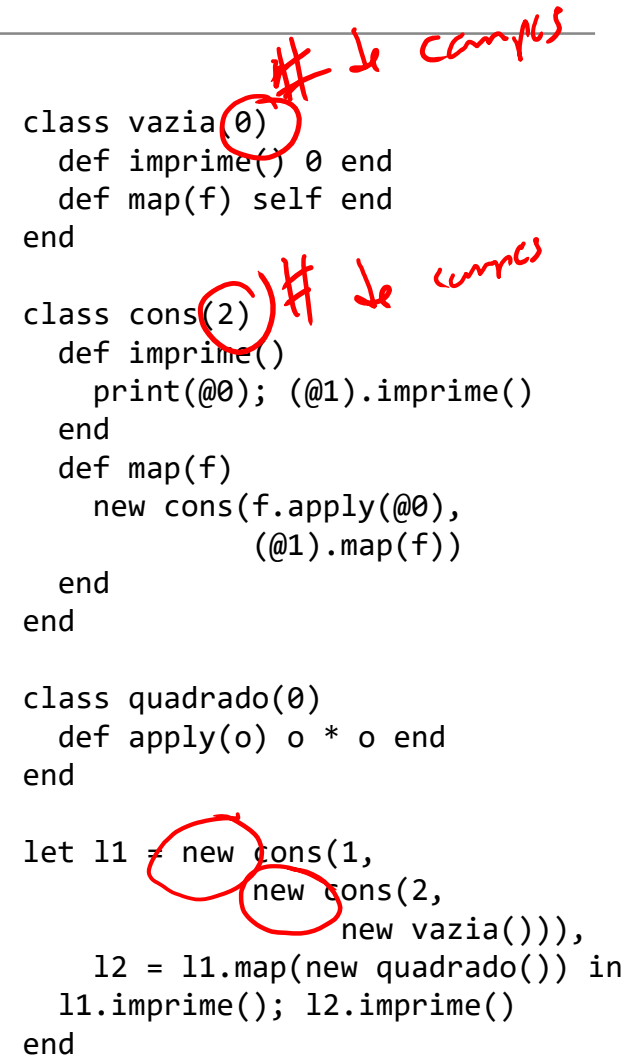
```
fun tracer(f)
  object () extends f
    fun apply(x)
      print(x);
      super.apply(x)
    end
  end
end

let fat = object ()
            fun apply(n)
              if n < 2 then
                1
              else
                n * self.apply(
                      n - 1)
              end
            end
          end in
  print(fat.apply(5));
  (tracer(fat)).apply(5)
end
```

Classes

- Uma classe é um molde para construir objetos
- A linguagem *proto* não tem classes na sua sintaxe, mas elas estão lá implicitamente, no número de campos do objeto e nos seus métodos
- Na maioria das linguagens OO o conceito de classe é bem mais explícito
- Uma classe por si só apenas descreve objetos; para instanciá-los usa-se uma operação primitiva de instanciação

Listas usando classes

- A primitiva new precisa do nome da classe que ela tem que instanciar, e dos valores para os campos
- As classes não são valores, a única coisa que podemos fazer com uma classe é instanciá-la
- Em essência, esse é o modelo OO de Java; os campos e métodos estáticos são simplesmente variáveis globais e funções com regras de escopo específicas




```
class vazia(0)
  def imprime() 0 end
  def map(f) self end
end

class cons(2)
  def imprime()
    print(@0); (@1).imprime()
  end
  def map(f)
    new cons(f.apply(@0),
             (@1).map(f))
  end
end

class quadrado(0)
  def apply(o) o * o end
end

let l1 = new cons(1,
              new cons(2,
                   new vazia())),
    l2 = l1.map(new quadrado()) in
  l1.imprime(); l2.imprime()
end
```

Classes de primeira classe

- Em linguagens como Smalltalk e Ruby, classes também são objetos, que podem ter seus próprios métodos e campos
- A classe de uma classe é a sua *metaclasse*; se uma classe é subclasse de outra, então a sua metaclasse é subclasse da metaclasse da outra
- Metaclasses não precisam ser objetos, mas se forem todos podem ser instâncias de uma única classe
- Cada classe do sistema é uma instância única (um *singleton*) de sua *metaclasse*

Listas em JavaScript ES6

- O programa ao lado é uma versão em JavaScript do programa *classe* de dois slides atrás

- Notem que não instanciamos `Vazia` e `Quadrado`! Eles já são objetos que têm os métodos que precisamos (`imprime`, `map`, `apply`)

```
const Vazia = {
    imprime() {},
    map(f) { return this; }
};
const Quadrado = {
    apply(x) { return x * x }
}
class Cons {
    constructor (h, t) {
        this.h = h; this.t = t
    }
    imprime() {
        console.log(this.h); this.t.imprime()
    }
    map(f) {
        return new Cons(f.apply(this.h), this.t.map(f))
    }
}
const l1 = new Cons(2, new Cons(3, Vazia))
const l2 = l1.map(Quadrado)
l1.imprime()
l2.imprime()
```

tracer em JavaScript ES6

- Com objetos literais conseguimos implementar uma versão da função `tracer` do slide 6

- A função `Object.setPrototypeOf` muda o *protótipo* do objeto para ser outro objeto, fazendo o equivalente ao “extends” de proto

```
const Fat = {
    apply(n) {
        if(n < 2)
            return 1
        else
            return n * this.apply(n-1)
    }
}
function tracer(f) {
    const o = {
        apply(x) {
            console.log(x);
            return super.apply(x)
        }
    }
    Object.setPrototypeOf(o, f)
    return o
}
console.log(Fat.apply(5))
console.log(tracer(Fat).apply(5))
```