Linguagens de Programação

Fabio Mascarenhas - 2013.1

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

Recursão aberta

- Delegação permite reutilizar a implementação dos métodos do objeto `counter` na implementação dos métodos de `c2`
- Mas só com delegação não temos *recursão aberta*
- O programa ao lado imprime -3: a implementação de `dec` em `c2` delega para a implementação de `dec` em `counter`, que chama `self.inc`, só que `self` é o clone de `counter` que está em `@0` de `c2`
- Na recurão aberta, `self` seria `c2`, e o programa imprimiria -6

```
object counter(1)
  def inc(n)
    @0 = @0 + n
  end
  def dec(n)
    self.inc(-n)
  end
  def clone() new end
end

object c2(1)
  def init(c)
    @0 = c.clone()
  end
  def inc(n)
    (@0).inc(n*2)
  end
  def dec(n)
   (@0).dec(n)
  end
end

c2.init(counter);
print(c2.dec(3))  -- -3
```

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 (mas não o seu conteúdo), 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

```
object c1(1)
  def inc(n)
    @0 = @0 + n
  end
  def dec(n)
    self.inc(-n)
  end
  def clone()
    new
  end
end

object c2(0) extends c1
  def inc(n)
    @0 = @0 + n*2
  end
end

print(c1.inc(2)); -- 2
print(c2.dec(3))  -- -6
```

`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*

```
object c1(1)
  def inc(n)
    @0 = @0 + n
  end
  def dec(n)
    self.inc(-n)
  end
  def clone()
    new
  end
end

object c2(0) extends c1
  def inc(n)
    super.inc(n*2)
  end
end

print(c2.dec(3))  -- -6
```

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

```
object vazia(0)
  def imprime() 0 end
  def map(f) self end
end
object list(2)
  def init(h, t)
    @0 = h; @1 = t; self
  end
  def head() @0 end
  def tail() @1 end
  def cons(h, t) new.init(h, t) end
  def imprime()
    print(@0); (@1).imprime()
  end
  def map(f)
    self.cons(f.apply(@0), (@1).map(f))
  end
end

let l1=list.cons(1,list.cons(2, vazia)),
    l2=l1.map(object (0)
                def 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 método tracer.make constrói uma versão da “função” f que imprime seu argumento
- Com a recursão aberta, mesmo chamadas recursivas têm seus argumentos impressos

```
object fat(0)
  def apply(n)
    if n < 2 then 1
    else n * self.apply(n - 1) end
  end
end
object tracer(0)
  def make(f)
    object (0) extends f
      def apply(x)
        print(x); super.apply(x)
      end
    end
  end
end

(tracer.make(fat)).apply(5)
```

Classes

- Uma classe é um molde para construir objetos
- No mínimo, uma classe é quantos campos o objeto precisará ter, e quais métodos ele terá
- Se quisermos ter herança, a classe também pode ter uma *superclasse*
- Um objeto passa a ser um vetor de campos e uma classe, e a busca dos métodos é feita na classe (ou na sua superclasse)
- A linguagem *proto* não tem classes na sua sintaxe, mas elas estão lá implicitamente, no número de campos do objeto no seus métodos e no seu protótipo

Listas usando classes

- A primitiva `new` agora precisa do nome da classe que ela tem que instanciar
- Se chamamos um método de inicialização logo após `new` temos algo parecido com os *construtores* das linguagens OO
- 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 init(h, t)
    @0 = h; @1 = t; self
  end
  def head() @0 end
  def tail() @1 end
  def imprime()
    print(@0); (@1).imprime()
  end
  def map(f)
    (new cons).init(f.apply(@0),
                    (@1).map(f))
  end
end
class quadrado(0)
  def apply(o) o * o end
end

let l1=(new cons).init(1,
       (new cons).init(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 própriós 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 Ruby

- O programa ao lado é uma versão em Ruby do programa *classe* de dois slides atrás
- `imprime` e `map` são *class methods* da classe `Vazia`, e `apply` é um class method da classe Quadrado
- Notem que não instanciamos `Vazia` e `Quadrado`! Eles já são objetos que têm os métodos que precisamos (`imprime`, `map`, `apply`)
- `new` também é um *class method* que executa uma primitiva parecida com a `new` de *classe* e depois o método `initialize` do objeto recém-criado

```
class Vazia
  def Vazia.imprime() end
  def Vazia.map(f) self  end
end
class Cons
  def initialize(h, t)
    @h = h; @t = t
  end
  def imprime()
    puts(@h); @t.imprime()
  end
  def map(f)
    Cons.new(f.apply(@h),
@t.map(f))
  end
end
class Quadrado
  def Vazia.apply(x)
    x * x
  end
end

l1 = Cons.new(1, Cons.new(2,
Vazia))
l2 = l1.map(Quadrado)
l1.imprime()
l2.imprime()
```

tracer em Ruby

- Com metaclasses e *class methods* conseguimos implementar uma versão do objeto `tracer` do slide 6, coisa que não podemos fazer em *classe*
- Os métodos definidos com self são outro modo de definir class methods
- `Class.new` é uma maneira de definir uma classe anonimamente, usando uma superclasse
- Super chama o método *class method* `apply` na superclasse, que por sua vez chama o *class method* `apply` na subclasse, por recursão aberta

```
class Fat
  def self.apply(n)
    if n<2 then
      1
    else
      n * self.apply(n-1)
    end
  end
end

class Tracer
  def self.make(f)
    Class.new(f) do
      def self.apply(x)
        puts(x)
        super(x)
      end
    end
  end
end

puts(Tracer.make(Fat).apply(5))
```