# Linguagens de Programação

Fabio Mascarenhas - 2013.1

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

# Objetos sem classes

- Um *objeto* tem duas visões: a de fora e a de dentro

- Visto de fora, um objeto é uma entidade opaca, para a qual podemos mandar *mensagens*; uma mensagem pode ter *argumentos*, que são outros objetos, e gera uma *resposta*, que também é um objeto → chamada de método

- Visto de dentro, um objeto tem um conjunto de *campos*, e um conjunto de métodos, que correspondem às mensagens que esses objetos podem responder

- O código de um método tem acesso aos campos do objeto, e também pode *clonar* o objeto, criando uma cópia do mesmo

# Proto

- *proto* é em essência uma linguagem imperativa, como MicroC
- Só que os valores de proto agora podem ser *números* ou *objetos*
  - Objetos têm campos, que são uma lista de endereços de memória
- Temos as mesmas operações de MicroC para números, mas não temos mais * e &
- A operação @ acessa um campo do objeto corrente

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

print(counter.inc(1));
print(counter.inc(4));
print(counter.dec(2))
```

# self

- A função `eval` para as expressões de *proto* precisa receber mais um parâmetro: o *objeto corrente*
- São os campos desse objeto que o operador `@` manipula; quando enviamos uma mensagem a outro objeto, o corpo do método é avaliado com o outro objeto como objeto corrente
- Vamos chamar o objeto corrente de `self`, e expor ele para o programa com uma primitiva com esse nome

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

print(counter.inc(1));
print(counter.inc(4));
print(counter.dec(2))
```

# new

- Dentro de um objeto, a primitiva new cria um *clone* do objeto
- Os campos do clone são inicializados com os mesmos *valores* dos campos do objeto original, mas em *endereços diferentes*
- O clone e o objeto original têm *identidades* diferentes: `self == new` é falso
- O que acontece se mudarmos a implementação de dec de `self.inc(-n)` para `counter.inc(-n)`?

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

print(counter.inc(3));
let c1 = counter.clone() in
  print(c1.inc(5));
  print(counter.inc(2));
  print(c1.dec(1));
  print(c1.inc(1));
  print(c1 == counter)
end
```

# Delegação

- Um objeto pode *delegar* a implementação de seus métodos para outro objeto
- Com isso temos uma espécie de *herança* da implementação do objeto para qual ele delega

```
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)
  end
end

c2.init(counter)
print(c2.inc(3))  -- 3
```

# 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

```
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
```