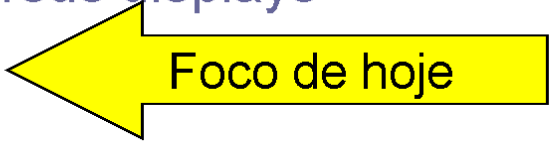
Abstrações de unidades
(Subprogramas – Parte II)

Modelos de Linguagens de Programação

Contextualização

- Abstração de processos: subprogramas
  - Abstração procedimental (comandos)
  - Abstração funcional (expressões ou valores)
- Definição e declaração de subprogramas
- Controle de chamadas
  - Registro de ativação e pilha
  - Cadeia estática e dinâmica
- Acesso aos dados que devem ser processados:
  - Acesso à variáveis não locais
    - encadeamento estático versus displays
  - Passagem de parâmetros



Questões principais

- **Como passar dados a um subprograma?**
  - Acesso direto a variáveis não locais:
    - vantagem: mais rápido do que parâmetros
    - desvantagem: redução de confiabilidade
  - Passagem de parâmetros:
    - vantagens: flexibilidade, parametrização (aplicação do mesmo código a diferentes valores)
    - desvantagem: mais lento
- Como transmitir computação (e não só dados)?
  - passagem de subprogramas

Subprogramas: questões de projeto

- Como se dá a associação entre parâmetros reais e formais?
- Quais são os modos de passagem oferecidos?
- Há verificação de tipos entre parâmetros formais e reais?
- Subprogramas também podem ser passados como parâmetros?
  - Se sim:
    - qual é o ambiente de referência dos mesmos?
    - há verificação de tipos de seus argumentos?

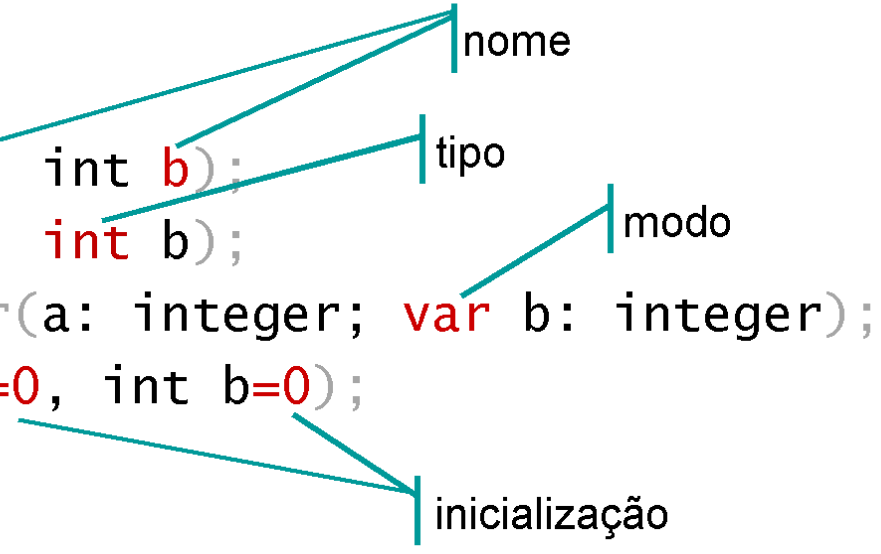
Passagem de parâmetros: conceitos

- Parâmetros formais:
  - Lista de parâmetros que o subprograma recebe (e seus respectivos tipos)
  - **Nomes locais** para a manipulação dos dados
  - Exemplo: int soma(int a, int b);
- Parâmetros reais:
  - **Argumentos** usados na chamada ao subprograma
  - Valores (reais) passados à unidade chamada (argumentos)
  - Exemplos: soma(10, 2);
    soma(a, 2*a);
- Tipo (da abstração):
  - Tipo de resultado da execução da sub-rotina (função)
  - Exemplo: int soma();

Parâmetros formais

- Elementos de um parâmetro formal:
  - Nome do parâmetro
  - Tipo do parâmetro
  - Modo de passagem do parâmetro (opcional, quando há mais de um tipo)
  - Inicialização do parâmetro (opcional, quando permitido)

- Exemplos:
  - `int soma(int a, int b);`
  - `int soma(int a, int b);`
  - `Procedure somar(a: integer; var b: integer);`
  - `int soma(int a=0, int b=0);`



Parâmetros formais x variáveis locais

```
void troca_int(int* u,int* v)
{
    int t;

    t = *u;
    *u = *v;
    *v = t;
}

:

main() {
    int  a = 3;  int  b = 5;
    troca (&a, &b);
}
```

Parâmetros formais

Variáveis locais

- Parâmetros formais:
  podem ser associados a uma expressão pré-existente ou a uma posição transmitida através da chamada
- Variáveis locais:
  são sempre associadas a uma (geralmente nova) posição de memória na entrada do bloco (ambiente)

OBS: ambos são (normalmente) alocados na pilha!

Parâmetros reais (argumentos de chamada)

- Elementos concordam com os respectivos parâmetros formais em:
  - número, tipo e ordem
  - modo de passagem (valor, referência...)

- Exemplos:

Dado que:

```
void soma(int a=0, int b=0);
Procedure somar(a: integer; var b: integer);
```

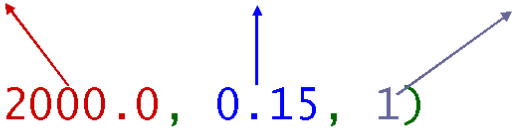
```
soma(x, y);    // x e y são inteiros
soma(10);      // coloca 0 em b (por padrão)
somar(10, b); // b é passado por referência
```

Associação de parâmetros

- Associação (vinculação/amarração) de parâmetros é a correspondência entre os parâmetros formais e os parâmetros reais
- Regra geral: (a) Parâmetros posicionais!
  - a associação se dá pela posição (mesma ordem, mesmo número, mesmo tipo!)
  - exemplo:

```
float f_pagto(float r, float imp, int isen)

f_pagto(2000.0, 0.15, 1)
```



Associação de parâmetros

- (b) Nomeados (ou por palavras-chave):
  - Exemplo em Ada: (também válido em Fortran 90)
    - definição: `proc1(in out a, b, c)`
    - chamada:
      ```
      proc1(a => x, b => y, c => z)
      proc1(x, c => z, b => y)
      ```
  - Exemplo em Visual Basic:
    ```
    ShowMsg(msg:=x, btDefault := 1);
    ```

Associação de parâmetros

- (c) Omissão de argumentos (por valores-*default*):
  - Ada, Fortran90, C++
  - Exemplo em C++ (devem aparecer por último):
    - definição: `float f_pagto(float r, float imp=0.25, int isen=1)`
    - chamada:
```
calc = f_pagto(2000.0, 0.15)
calc = f_pagto(2000.0)
```

Número variável de argumentos: Java

```
public static void main(String[] args) {
   soma(1.0);
   soma(1.0, 2.0);
   soma(1.0, 2.0, 3.0, 4.0, 50.0);
   soma(1.0, 2.0, 3.0, 4.0, 50.0, 100, 200, 3300);
}

public static void soma(double...ds){
   double resultado = 0;
   for(double d:ds) resultado += d;
   System.out.println("Resultado: "+ resultado);
}
```

Número variável de argumentos: C

```
void soma(const int nargs, ...){
    va_list argp; double resultado = 0;
    va_start(argp, nargs);
    for(int n = 0; n<nargs; n++)
       resultado+=va_arg(argp, double);
    va_end(argp);
    cout << "Resultado: " << resultado << endl;
}
main(){
   soma(2, 1.0, 2.0);
   soma(5, 1.0, 2.0, 3.0, 4.0, 50.0);
   return 0;
}
```

Semântica de parâmetros formais

- A especificação da associação (semântica) entre os parâmetros reais e os formais pode:
  1. determinar as operações que uma sub-rotina pode realizar com seus parâmetros formais
     (ou seja, restringir o que pode ser feito com os parâmetros)
  2. determinar os efeitos destas operações nos parâmetros reais correspondentes

Semântica de parâmetros formais

- **Modelos semânticos de passagem de parâmetros:**
  - **Modo entrada (in mode)**: o valor real não pode ser modificado no corpo do subprograma

```
a ------- valor -------> x
```

  - **Modo saída (out mode)**: o valor real não pode ser consultado no corpo do subprograma

```
b <------- valor ------- y
```

  - **Modo entrada/saída (in-out mode)**: o valor real pode ser consultado e modificado no corpo do subprograma

```
c <------- valor -----> z
```

Inicialização de parâmetros formais

- Modelos de passagem de parâmetros:
  1. por valor: semântica de modo entrada
  2. por resultado: semântica de modo saída
  3. por valor-resultado: semântica de modo entrada/saída
  4. por referência: semântica de modo entrada/saída
  5. por nome: semântica de modo entrada/saída

  - São possibilidades existentes, mas não quer dizer que as linguagens utilizem todas elas!
  - As mais comuns são por valor e por referência

Passagem por valor (*call by value*)

- Segue semântica de modo entrada ("*read-only*")
- Valor do argumento é usado para inicializar o parâmetro formal (seu valor é copiado)
- A partir desse momento, os dois funcionam de forma independente
- O parâmetro formal pode ser visto como uma variável local
- Exemplo em C:

Definição:

```
int fat(int num){
   int result=1;
   for(int i=num; i>0; i--)
      result*=i;
   return result;
End;
```

Chamada: `fat(p);`

Passagem por valor (*call by value*)

1. Transferência física (cópia) do parâmetro real para o RA do subprograma chamado:
   - Desvantagens:
     - necessidade de espaço de armazenamento adicional (um para o real e outro para a cópia no subprograma chamado)
     - a própria operação de transferência física (*time-consuming*, se o parâmetro for muito grande)
   - Vantagens:
     - funcionamento independente (proteção do parâmetro real)
     - parâmetro real não precisa ser uma variável

```
int fat(int num){
   int result=1;

   for(int i=num; i>0; i--)
      result*=i;

   return result;
End;
```

```
int p = 10;
fat(p);
```

Passagem por resultado

- Segue semântica de modo saída ("*write-only*"):
  - Serve portanto, para o retorno de valores
  - Parâmetro real tem que ser uma variável!
- Nenhum valor é transmitido para o parâmetro formal (não há inicialização)
- O parâmetro formal correspondente se comporta como uma variável local, que somente pode receber valores
- Ao término da sub-rotina, seu valor é passado de volta para o parâmetro real (que deve ser uma variável)!

Passagem por resultado

- Desvantagens / problemas:
  - operações de armazenamento e cópia extras (para levar o resultado de volta)
  - problema de imprevisibilidade:

```
Dado sub(x, y),
  sub(p1, p1)        → chamada com colisão - qual seria o resultado?
  sub(p1[index])     → se index for global e mudar dentro da sub-rotina?
  sub(10)            → o que acontece se o argumento é uma constante?
```

Passagem por valor-resultado

- Também chamada de passagem por cópia
- Semântica de entrada/saída:
  - Valores reais são transferidos e inicializam os formais
  - Na finalização, o valor do parâmetro formal é transmitido de volta
- Combinação de passagem por valor e por resultado
- Possui as mesmas desvantagens de passagem por valor e por resultado:
  - Armazenamento múltiplo
  - Tempo de cópia (na entrada e na saída!)

Passagem por referência (*call by reference*)

- Semântica de modo entrada/saída ("r*ead/write*")
- Transmite somente caminho de acesso (endereço), de forma transparente:
  - O próprio argumento é usado como parâmetro formal (que funciona como um *alias* para o parâmetro real correspondente): não há cópia
  - Formal e real referenciam o mesmo objeto, mudanças realizadas em um afetam no outro
  - Na maioria das linguagens, o parâmetro passado tem que ser um L-value, ou seja, não pode ser uma expressão (contra-exemplo: Fortran, que cria variável temporária)
- Vantagem: eficiência de tempo e espaço
- Exemplo em Pascal:

```
Procedure Teste(Var num : integer);
Begin
  num := 10;
End;
```

Exemplo em C++:

```
foo(int& num)
{
   num = 10;
}
```

Passagem por referência (*call by reference*)

- Desvantagens:
  - Acesso mais lento (endereçamento por indireção / desreferenciamento implícito)
  - Efeitos colaterais: alteração (por vezes indesejada) de valores no ambiente de chamada
  - Criação de apelidos (*aliases*)

Exemplo de aliases em Pascal

```
procedure confusa (var m, n: integer);
begin
  n:=1;
  n:= m + n;  // 'm' e 'n' são aliases para 'i'!
end;

{ ... }

i:= 4;
confusa(i,i); // Qual o valor de 'i' após a execução?
```

Exemplo de aliases em C

```
void fun(int *first, int *second) {…}

// ...

fun(&total,&total);
fun(&list[i], &list[j]);     // se i == j
fun(&list[i], list);         // se i == 0
```

OBS:

- Apesar de oferecer passagem por valor, ponteiros oferecem a mesma semântica de passagem por referência

Sobre o uso de ponteiros em C

2. Passagem por valor com transmissão do caminho de acesso do parâmetro real ao chamador:
   - O uso de ponteiros (C/C++) transmite o caminho de acesso do parâmetro real ao chamador, simulando a semântica de passagem por referência, desde que os operadores de referenciamento (&) e desreferenciamento (*) sejam usados
   - Vantagem: velocidade, principalmente se o parâmetro real for muito grande (p.ex., vetores e matrizes)
   - Problema: como impor proteção contra escrita?
     - C permite isso com a palavra-chave *const* (**ponteiros *const* oferecem a eficiência da passagem por referência com a semântica da passagem por valor)**

Passagem por nome

- Semântica de modo entrada/saída ("*read-write*")
- O **parâmetro real** substitui textualmente o parâmetro formal em todas as suas ocorrências no subprograma
- O **parâmetro formal** representa uma função de acesso
- **Usa vinculação tardia**:
  - O parâmetro formal é vinculado a um valor ou endereço somente quando é atribuído ou referenciado
  - Vantagem: Maior flexibilidade
  - Desvantagem: menor velocidade
  - Desvantagem: difíceis de implementar e de depurar

Passagem por nome

- Exemplo em Algol-like:

```
procedure BIGSUB;
integer GLOBAL;
integer array LIST[1:2];
procedure SUB(PARAM);
   integer PARAM;                  // PARAM é LIST[GLOBAL] → LIST[1]
   begin
      PARAM  := 3;                 // LIST[GLOBAL] := 3 ➔ LIST[1] := 3
      GLOBAL := GLOBAL + 1;
      PARAM  := 5;                 // LIST[GLOBAL] := 5 → LIST[2] := 5
   end;
Begin
   LIST[1] := 2;
   LIST[2] := 2;
   GLOBAL  := 1;
   SUB(LIST[GLOBAL])
END;
```

Ao final, LIST = [3 , 5]

Observações gerais

- C utiliza somente passagem por valor
  - para modificar o valor de uma variável externa, dentro de uma função, utilize ponteiros (já que passam o endereço de memória)
  - no caso de *arrays*, devido a sua interoperabilidade com ponteiros, o seu endereço é que é passado
- Fortran trabalha com referências, mas os parâmetros reais não necessariamente precisam ser *L-values*. No caso de expressões, uma variável temporária é criada pelo compilador e sua referência é passada!
- Pascal utiliza passagem por valor, por *default*, mas permite ao programador especificar que deseja passagem por referência, utilizando a palavra-chave "Var"

Observações gerais

- Utilizar passagem por referência somente nos seguintes casos:
  - quando realmente necessitar modificar o valor do parâmetro real; (ou)
  - quando o dado original for muito grande (e a cópia demorar muito)

  - Neste caso, lembre-se de que:
    - a indireção necessita de desreferenciamento, que tem custo associado (podendo ser maior do que o de cópia, dependendo do caso)
    - A rotina pode modificar o valor original sem querer

  - Alternativa: algumas linguagens permitem especificar que o parâmetro é só de leitura e o compilador garante que ele não o está modificando
    - Modula-3 : palavra-chave READONLY
    - C : palavra-chave const
    - C++ : referências const

Observações gerais

- Smalltalk, Lisp, ML e Clu, devido a O.O., trabalham com referências a objetos:
  - um parâmetro real já é uma referência (logo, a passagem não é bem por referência, senão seria a referência da referência!)
  - trabalham, portanto, com *call-by-sharing* (Scott, 2000, p. 444):
    - referências: o endereço é passado (e o objeto fica compartilhado!)
    - objetos imutáveis, implementados como valores: passagem do valor
- Java funciona de forma similar:
  - referências: passagem *by sharing*
  - tipos primitivos: passagem por valor
- ADA trabalha com os modos *in* (*call by value*), *out* (*call by result*) e *in-out* (*call by value/result*)

Subprogramas como parâmetros

- Objetivo: passar um subprograma como argumento de/para outro subprograma
- Motivação: aplicar diferentes predicados de uma mesma forma
- Exemplos:
  - Função de busca em um banco de dados onde a pesquisa é feita da mesma forma, mas a seleção pode mudar
    - reescrever a função de seleção para cada pesquisa possível?
    - passar a função de seleção como parâmetro?
  - Funções de ordenação para diferentes elementos
  - Rotinas de *call-back*
- Questões de projeto:
  - Como passar argumentos e como fazer sua verificação de tipo?
  - Como definir o ambiente de referência do argumento?
- OBS: uma alternativa a isso é a implementação de subprogramas sobrecarregados e/ou genéricos e o uso de iteradores!

Exemplo

Fonte: Scott (2000)

```
Type person = record
    ...
    age: integer;
    ...
Threshold: integer
People: database

// main program
...
Threshold := 35
Print_selected_records(people, older_than, print_person)

Procedure print_selected_records(db: database; predicate, print_routine: procedure)
    line_length: integer

    if device_type(stdout) = terminal then line_length := 80
    else                                   line_length := 132

    foreach record r in db
       if predicate(r)             // se for um predicado (registro) válido
          then print_routine(r)    // imprime

Function older_than(p:person):boolean
    return p.age >= threshold

Procedure print_person(p: person)
    // call appropriate I/O routines to print record on stadard output.
    // Make use of non-local variable line_lenght to format data in columns
    ...
```

C / C++

Em C e em C++ não é possível passar uma função como parâmetro, mas seu endereço sim:

- O tipo de um ponteiro para a função é o protocolo dela
- Como o protocolo inclui todos os tipos de parâmetro, eles podem ser verificados!

Callback functions são comuns quando se utiliza Windows API, DirectX...

```
/* Library code */
int traverseWith(int array[], size_t length,
                 int (*callback)(int index,
                 int item, void *param),
                 void *param)
{
   int exitCode = 0;
   for (int i = 0; i < length; i++) {
     exitCode = callback(i, array[i], param);
     if (exitCode) { break; }
   } return exitCode;
}

/* APPLICATION CODE */
int search (int index, int item, void *param) {
   if (item > 5) {
      *(int *)param = index;
      return 1;
   } else { return 0; }
}

/* (in another function) */
   int index;
   int found;
   found = traverseWith(array, length,
                        search, &index);
   if (found) {
      printf("Item %d\n", index);
   } else { printf("Not found\n"); }
```

Fonte: Callback, Wikipedia

Algol 68 / Pascal

```
Procedure integral(function fun(x: real): real;
                   liminf, limsup: real;
                   var result: real);
...
Var funval: real;
Begin
   ...
   funval := fun(liminf);
   ...
End;
```

Os tipos dos parâmetros formais são incluídos na lista de parâmetros formais do subprograma recebedor, permitindo verificação estática!

Ambiente de referência em subprogramas passados como parâmetro

- Uma questão importante refere-se ao ambiente de referenciamento correto para executar o subprograma passado, ou seja, de onde vem os seus parâmetros?
- Opções:
  - *Deep binding* (vinculação profunda)
  - *Shallow binding* (vinculação rasa)
  - *Ad-hoc binding* (vinculação ad-hoc)

Ambiente de referência em subprogramas passados como parâmetro

- Se for dado por *Deep binding*:
  - "Vinculação profunda", usando *early binding*
  - A vinculação das variáveis é feita no momento em que o subprograma é chamado
  - O **ambiente de referência corresponde ao de onde o subprograma foi definido**, ou seja, é o local onde o subprograma foi textualmente definido (técnica utilizada na maioria das linguagens de **escopo estático** e que são estruturadas em blocos)
  - É normalmente implementado através de uma representação explícita do ambiente de referência, juntamente com uma referência à sub-rotina propriamente dita (lembrar do vínculo estático do RA)
  - Este conjunto (referência ao ambiente + referência ao código) é chamado de *closure* (clausura = confinamento)

Ambiente de referência em subprogramas passados como parâmetro

- Se for dado por *Shallow binding:*
  - "Vinculação rasa", usando amarração tardia (*late binding*)
  - A vinculação das variáveis é feita no momento em que são realmente utilizadas
  - O **ambiente de referência corresponde ao da instrução de chamada** do subprograma passado, ou seja, o subprograma onde ele é executado/chamado (lembrar do vínculo dinâmico)
  - Normalmente é a opção *default* em LPs com escopo dinâmico
  - Exemplo: Lisp 1.5
  - Mais detalhes, incluindo informações sobre como melhorar sua performance, em [BAKER, 1978]

Ambiente de referência em subprogramas passados como parâmetro

- Se for Ad-hoc binding:
  - Como o anterior, usa *late binding*
  - Mas o ambiente de referência é **definido pelo subprograma que passou a sub-rotina como parâmetro real**
  - Não há implementações práticas em linguagem alguma

Exemplo comparativo

```
Procedure SUB1;
  var x: integer;
  procedure SUB2;
  begin
    write('x=', x);
  end;
  procedure SUB3;
  var x: integer;
  begin
   x := 3;
   SUB4(SUB2);
  end;
  procedure SUB4(SUBX);
  var x: integer;
  begin
   x := 4;
   SUBX;
  end;
Begin
  x := 1;
  SUB3;
End;
```

Deep binding

Ad-hoc

Shallow

- Qual o valor de x quando usamos:
  - Deep binding? (estrutura textual)
    - X = 1

      Porque o ambiente de referência é onde subx (sub2) está definida, ou seja, sub1
  - Shallow binding? (quem executa)
    - X = 4

      Porque o ambiente de referência é quem chamou subx, ou seja, sub4
  - Ad-hoc binding? (quem passa)
    - X = 3

      Porque o ambiente de referência é quem passou a sub-rotina como parâmetro, ou seja, sub3 (pois passou sub2 na chamada a sub4)

Closures

- Utilizadas em linguagens dinâmicas (e.g., Lisp, Ruby...)
- Similares a *Function pointers* em C, *Inner classes* em Java e d*elegates* em C#
- Mas com a diferença de poder referenciar (usar) variáveis visíveis no momento de sua definição (utilizar uma variável local da função externa)
- **Closures** são **blocos de código** que podem ser passados para uma função **+ conjunto de amarrações** do ambiente de onde eles vieram
- São funções de primeira ordem (classe) que podem utilizar variáveis do ambiente que as envolviam em tempo de criação

Exemplo de closure em Ruby

Fonte: Fowler (2004)

```
# PaidMore retorna uma função (closure) cujo
  comportamento  depende do argumento
# Proc.new utiliza uma função que manipula variáveis
  definidas fora do escopo dela
def paidMore(amount)
  return Proc.new {|e| e.salary > amount}
end
highPaid = paidMore(150) # dá um nome para a closure

john = Employee.new
john.salary = 200
print highPaid.call(john) # chama função (closure)
```

Exemplo em uma pseudo linguagem Java like

Fonte: Costanza (2007)

```
function createAdder(int n){
    int adder(int k) {
        n = n + k;
        return n;
    }
    return adder;
}
f = createAdder(5);
f(6); // resulta em 11 (5+6)
f(7); // resulta em 18 (11+7)
f2 = createAdder(6);
f2(6); // resulta em 12
```

- Adder é definido dentro do escopo de createAdder
- Adder refere-se a um parâmetro definido em createAdder
- **Mesmo quando retornado, tal associação (*binding*) continua existindo!**
- Os esquemas atuais de implementação de variáveis locais em linguagens estáticas não permitem o uso de closures (visto que as variáveis locais saem da pilha)

Leitura fortemente recomendada

- Baker, Henry G. **Shallow Binding in LISP 1.5**. Commun. ACM 21(7): 565-569 (1978). Disponível em: http://home.pipeline.com/~hbaker1/ ShallowBinding.html. Acesso em: 18/6/2007.
- Sebesta, R. W. **Subprogramas** (capítulo 9). In: Sebesta, R. W. Conceitos de Linguagens de programação. 5a edição. Porto Alegre: Bookman, 2003.
- Sebesta, R. W. **Implementando Subprogramas** (capítulo 10). In: Sebesta, R. W. Conceitos de Linguagens de programação. 5a edição. Porto Alegre: Bookman, 2003.
- Scott, Michael. **Parameter Passing** (chapter 8.3). In: Scott, Michael. Programming language Pragmatics. San Diego, CA: Academic Press, 2000.
- Oracle. **Arbitrary number of arguments**. In: The Java™ Tutorials. Disponível em: http://download.oracle.com/javase/tutorial/java/javaOO/ arguments.html#varargs. Acesso em: 08/06/2011.
- Oracle. **Using non-reifiable parameters with varargs methods**. In: The Java™ Tutorials. Disponível em: http://download.oracle.com/javase/tutorial/ java/generics/non-reifiable-varargs-type.html. Acesso em: 08/06/2011.
- Fowler, M. **Closure**, 2004. Disponível em: http://www.martinfowler.com/bliki/Closure.html. Acesso em: 02/04/2011.
- Costanza; Pascal. **Scope and Closures**, 2007. Disponível em: http://c2.com/cgi/wiki?ScopeAndClosures. Acesso em: 10/03/2011.