

# Subprogramas

---

Marco A L Barbosa  
malbarbo.pro.br

Departamento de Informática  
Universidade Estadual de Maringá



Este trabalho está licenciado com uma Licença Creative Commons - Atribuição-CompartilhaIgual 4.0 Internacional.  
<http://github.com/malbarbo/na-lp-copl>

# Conteúdo

Fundamentos

Questões de projeto

Ambientes de referência local

Métodos de passagens de parâmetros

Parâmetros que são subprogramas e fechamentos

Questões de projeto referentes a funções

Subprogramas sobrecarregados e genéricos

Operadores sobrecarregados definidos pelo usuário

# Fundamentos

- Abstração de processo
  - Subprogramas
  - Elementos fundamentais dos programas
- Abstração de dados
  - Capítulos 11 e 12

- Características gerais dos subprogramas
  - Cada subprograma tem um único ponto de entrada
  - Toda unidade de programa chamadora é suspensa durante a execução do subprograma chamado
  - O controle sempre retorna para o chamador quando a execução do subprograma chamado termina

- Definições básicas
  - Uma **definição de subprograma** descreve a interface e as ações do subprograma
  - Uma **chamada de subprograma** é a solicitação explícita para executar o subprograma
  - Um subprograma está **ativo** se depois de chamado, ele iniciou a sua execução, mas ainda não a concluiu

- Definições básicas
  - O **cabeçalho do subprograma** é a primeira parte da definição
    - Especifica o tipo (função, procedimento, etc)
    - Especifica o nome
    - Pode especificar a lista de parâmetros

- Exemplos de cabeçalhos
  - Fortran: `Subrotine Adder(parameters)`
  - Ada: `procedure Adder(parameters)`
  - Python: `def adder(parameters):`
  - C: `void adder(parameters)`
  - Lua (funções são entidades de primeira classe)
    - `function cube(x) return x * x * x end`
    - `cube = function (x) return x * x * x end`



- Definições básicas
  - O **perfil dos parâmetros** é o número, a ordem e o tipo dos parâmetros formais
  - O **protocolo** é o perfil dos parâmetros mais o tipo de retorno (em caso de funções)
  - Declarações (protótipos em C/C++)
    - A declaração especifica o protocolo do subprograma, mas não as ações
    - Necessário em linguagens que não permitem referências futuras (**forward references**) a subprogramas

- Existem duas maneiras de um subprograma acessar os dados para processar
  - Acesso direto as variáveis não locais
  - Passagem de parâmetros

- Os parâmetros no cabeçalho do subprograma são chamados de **parâmetros formais**
- Os parâmetros passados em uma chamada de subprograma são chamados de **parâmetros reais**

- Vinculação entre os parâmetros reais e os parâmetros formais
  - A maioria das linguagens faz a vinculação através da posição (**parâmetros posicionais**): o primeiro parâmetro real é vinculado com o primeiro parâmetro formal, e assim por diante
- Os parâmetros posicionais funcionam bem quando o número de parâmetros é pequeno
- Existem outras formas?

- Parâmetros de palavras-chave (Ada, Fortran 95, Python)
  - Exemplo em Python

```
def soma(lista, inicio, fim):  
    ...  
soma(inicio = 1, fim = 2, lista = [4, 5, 6])  
soma([4, 5, 6], fim = 1, inicio = 2)
```

# Parâmetros

- Parâmetros com valor padrão (Python, Ruby, C++, Fortran 95, Ada)

- Exemplo em Python

```
def compute_pay(income, exemptions = 1, tax_rate):  
    ...  
pay = compute_pay(20000.0, tax_rate = 0.15)
```

- Exemplo em C++

```
float compute_pay(float income,  
                  float tax_rate,  
                  int exemptions = 1) { ... }  
pay = compute_pay(20000.0, 0.15);
```

# Parâmetros

- Número variável de parâmetros (C/C++/C#, Python, Java, JavaScript, Lua, etc)
  - Exemplo em C#

```
public void DisplayList(params int[] list) {  
    foreach (int next in list) {  
        Console.WriteLine("Next value {0}", next);  
    }  
}  
  
int[] list = new int[6] {2, 4, 6, 8, 10, 12};  
DisplayList(list);  
DisplayList(2, 4, 3 * x - 1, 17);
```

# Parâmetros

- Número variável de parâmetros (C/C++/C#, Python, Java, JavaScript, Lua, etc)
  - Exemplo em Python

```
def fun1(p1, p2, *p3, **p4):  
    print('p1 =', p1, 'p2 =', p2,  
          'p3 =', p3, 'p4 =', p4)
```

```
> fun1(2, 4, 6, 8, mon=68, tue=72, wed=77)  
p1 = 2 p2 = 4 p3 = (6, 8)  
p4 = {'wed': 77, 'mon': 68, 'tue': 72}
```



- Existem duas categorias de subprogramas
  - Procedimentos
  - Funções

- **Procedimentos** são coleções de instruções que definem uma computação parametrizada
  - Produzem resultados para a unidade chamadora de duas formas: através das variáveis não locais e alterando os parâmetros
  - São usados para criar novas instruções (sentenças)

- **Funções** são baseadas no conceito matemático de função
  - Produz um valor, que é o efeito desejado
  - São usadas para criar novos operadores
  - Uma função sem efeito colateral é chamada de **função pura**

## Questões de projeto

## Questões de projeto

- As variáveis locais são alocadas estaticamente ou dinamicamente?
- Definições de subprogramas podem aparecer em outra definição de subprograma?
- Quais métodos de passagem de parâmetros são usados?

## Questões de projeto

- Os tipos dos parâmetros reais são checados em relação ao tipo dos parâmetros formais?
- Se subprogramas podem ser passados como parâmetros e podem ser aninhados, qual é o ambiente de e referenciamento de um subprograma passado?
- Os subprogramas podem ser sobrecarregados?
- Os subprogramas podem ser genéricos?

**Ambientes de referência local**

- Variáveis locais
  - São definidas dentro de subprogramas
  - Podem ser estáticas ou dinâmicas na pilha
    - Vantagens e desvantagens (capítulo 5 e seção 9.4.1)



- Subprogramas aninhados
  - Permite que um subprograma esteja visível apenas no subprograma que ele é utilizado
  - Por algum tempo estava presente em linguagens descendentes de Algol
  - Comum em linguagens funcionais e em linguagens mais recentes

## **Métodos de passagens de parâmetros**

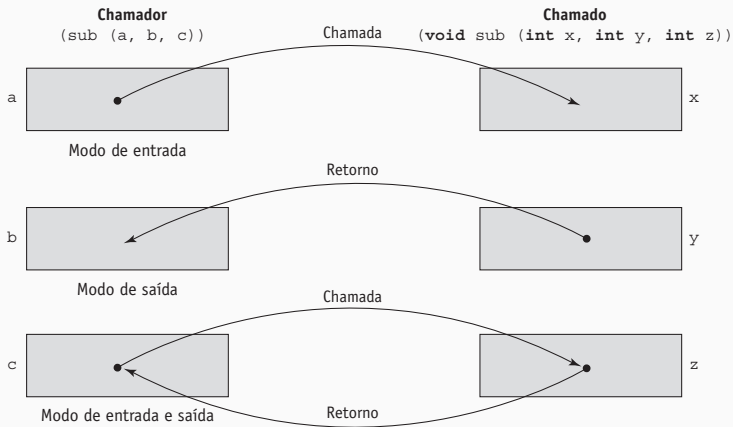
# Métodos de passagens de parâmetros

- Um **método de passagem de parâmetro** é a maneira como os parâmetros são transmitidos para (ou/e do) subprograma chamado
- Os parâmetros formais são caracterizados por um de três modelos semânticos
  - Podem receber dados dos parâmetros reais (modo de entrada)
  - Podem transmitir dados para os parâmetros reais (modo de saída)
  - Podem fazer ambos (modo de entrada/saída)

# Métodos de passagens de parâmetros

- Existem dois modelos conceituais sobre como os dados são transferidos na passagem de parâmetros
  - O valor real é copiado
  - Um caminho de acesso é transmitido

# Métodos de passagens de parâmetros



**Figura 9.1** Os três modelos semânticos de passagem de parâmetros quando movimentações físicas são usadas.

# Métodos de passagens de parâmetros

- Modelos de implementação
  - Passagem por valor
    - Quando um parâmetro é passado por valor, o valor do parâmetro real é utilizado para inicializar o parâmetro formal correspondente (modo de entrada)
    - A passagem por valor geralmente é implementada por cópia, mas pode ser implementada transmitindo-se o caminho de acesso
    - Vantagem: rápido para valores escalares
    - Desvantagem: memória extra e tempo de cópia (para parâmetros que ocupam bastante memória)

- Modelos de implementação
  - Passagem por resultado
    - É uma implementação do modo de saída
    - Quando um parâmetro é passado por resultado, nenhum valor é transmitido para o subprograma
    - O parâmetro formal funciona como uma variável local
    - Antes do retorno do subprograma, o valor é transmitido de volta para o parâmetro real

- Modelos de implementação
  - Passagem por resultado
    - Mesmas vantagens e desvantagens da passagem por valor
    - Colisão de parâmetros reais
    - Momento da avaliação do endereço dos parâmetros reais



## Métodos de passagens de parâmetros

- Exemplo passagem por resultado em C#

```
void Fixer (out int x, out int y) {  
    x = 17; y = 35;  
}  
...  
Fixer(out a, out a);  
...  
void DoIt(out int x, out int index) {  
    x = 17; index = 42;  
}  
...  
sub = 21;  
DoIt(out list[sub], out sub);
```

- Modelos de implementação
  - Passagem por valor-resultado (por cópia)
    - É uma implementação do modo de entrada/saída
    - É uma combinação da passagem por valor e passagem por resultado
    - Compartilha os mesmos problemas da passagem por valor e passagem por resultado

- Modelos de implementação
  - Passagem por referência
    - É uma implementação do modo de entrada/saída
    - Ao invés de copiar os dados, um caminho de acesso é transmitido (geralmente um endereço)
    - Vantagens: eficiente em termos de espaço e tempo
    - Desvantagens: acesso mais lento devido a indireção, apelidos podem ser criados

## Métodos de passagens de parâmetros

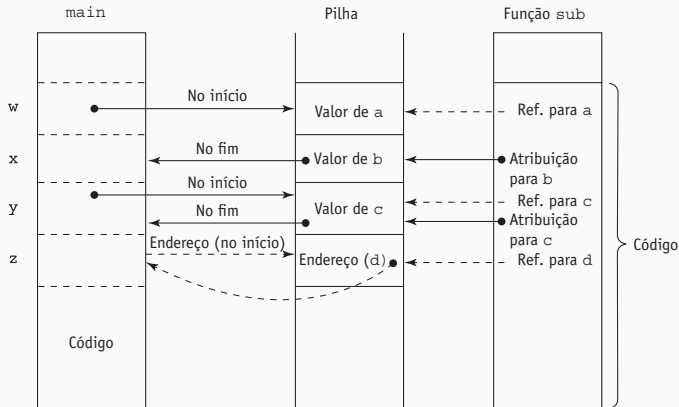
- Exemplo passagem por referência em C++

```
void fun(int &first, int &second){...}  
...  
fun(total, total);  
fun(list[i], list[j]);  
fun1(list[i], list);  
int *global;  
void main() {  
    sub(global);  
}  
void sub(int *param) {...}
```

- Modelos de implementação
  - Passagem por nome
    - É um método de passagem de parâmetro de modo de entrada/saída
    - Não corresponde a um único modelo de implementação
    - O parâmetro real substitui textualmente o parâmetro formal em todas as ocorrências do subprograma
    - Usando em meta programação

- Implementação
  - Na maioria das linguagens contemporâneas, a comunicação dos parâmetros acontece através da pilha
  - A pilha é inicializada e mantida pelo sistema de tempo de execução

# Métodos de passagens de parâmetros



Cabeçalho da função: `void sub (int a, int b, int c, int d)`

Chamada da função em main: `sub (w, x, y, z)`

(passando *w* por valor, *x* por resultado, *y* por valor-resultado, *z* por referência)

**Figura 9.2** Uma possível implementação e pilha dos métodos comuns de passagem de parâmetros.

# Métodos de passagens de parâmetros

- Exemplos de algumas linguagens
  - C usa passagem por valor, passagem por referência pode ser obtida usando ponteiros
  - Em C e C++, os parâmetro formais podem ter o tipo ponteiro para constante
  - C++ inclui um tipo especial de ponteiro, chamado de tipo referência

```
void fun(const int &p1, int p2, int &p3) { ... }
```

- Todos os parâmetro em Java são passados por valor. Como os objetos são acessados por referência, os parâmetros dos tipos objetos são efetivamente passados por referência



# Métodos de passagens de parâmetros

- Exemplos de algumas linguagens
  - Ada implementa os três modelos semânticos

```
procedure Adder(A : in out Integer;  
               B : in Integer;  
               C : out Float)
```

- Parâmetros no modo de saída podem ser atribuídos mas não referenciados
- Parâmetros no modo de entrada podem ser referenciados mas não atribuídos
- Parâmetros no modo de entrada/saída podem ser referenciados e atribuídos

- Exemplos de algumas linguagens
  - Em Ada 95, todos os escalares são passados por cópia e todos os valores de tipos estruturados são passados por referência
  - Fortran 95 é similar ao Ada
  - E as linguagens de scripts?

- Checagem de tipos dos parâmetros
  - As primeiras linguagens de programação (como Fortran 77 e C) não requeriam checagem dos tipos dos parâmetros
  - A maioria das linguagens atuais fazem esta checagem
  - E as linguagens de scripts?

# Métodos de passagens de parâmetros

- Checagem de tipos dos parâmetros
  - Em C89, o programador pode escolher

```
double sin(x) // sem checagem
    double c;
{ ... }
```

```
int count = 123;
double value = sin(count);
```

```
double sin(double x) // com checagem
{ ... }
```

# Métodos de passagens de parâmetros

- Arranjos multidimensionais como parâmetros
  - O compilador precisa saber o tamanho do arranjo multidimensional para criar a função de mapeamento
  - Em C/C++ o programador tem que declarar todos os tamanhos (menos do primeiro subscrito)

```
void fun(int matrix[][10]) {...}  
void main() {  
    int mat[5][10];  
    fun(mat);  
}
```

- Ada, Java e C# não tem este problema, o tamanho do arranjo faz parte do objeto

# Métodos de passagens de parâmetros

- Considerações de projeto
  - Eficiência
  - Transferência de dados em uma ou duas direções
- Estas considerações estão em conflito
  - As boas práticas de programação sugerem limitar o acesso as variáveis, o que implica em usar transferência em uma direção quando possível
  - Mas passagem por referência é mais eficiente para estruturas com tamanho significativo

**Parâmetros que são subprogramas e  
fechamentos**

## Parâmetros que são subprogramas

- Existem muitas situações que é conveniente passar um nome de subprograma como parâmetro para outros subprogramas.

### Exemplos

- A ação que deve ser realiza quando um evento ocorre (ex: clique de botão)
- A função de comparação utilizada por um subprograma de ordenação



## Parâmetros que são subprogramas

- Simples se apenas o endereço da função fosse necessário, mas existe uma questão que deve ser considerada
  - Qual é o ambiente de referenciamento usado na execução do subprograma passado como parâmetro?

- Um **fechamento** (*closure* em inglês) é um subprograma e o ambiente de referenciamento onde ele foi definido
- O ambiente de referenciamento é necessário pois o subprograma pode ser chamado em qualquer local

- Qual é o ambiente de referenciamento usado na execução do subprograma passado como parâmetro?
  - **Vinculação rasa:** o ambiente da instrução de chamada que ativa o subprograma passado - natural para linguagens com escopo dinâmico
  - **Vinculação profunda:** o ambiente da definição do subprograma passado - natural para linguagens com escopo estático
  - **Vinculação ad hoc:** o ambiente da instrução de chamada que passou o subprograma como parâmetro real

# Fechamentos

Exemplo usando a sintaxe de JavaScript

```
function sub1() {  
  var x;  
  function sub2() {  
    print(x);  
  }  
  function sub3() {  
    var x = 3;  
    sub4(sub2);  
  }  
  function sub4(subx) {  
    var x = 4;  
    subx();  
  }  
  x = 1;  
  sub3();  
}
```

Qual será o valor impresso na função sub2 quando ela for chamada em sub4?

- Vinculação rasa: 4 (x de sub4)
- Vinculação profunda: 1 (x de sub1)
- Vinculação ad hoc: 3 (x de sub3)

# Fechamentos

```
def somador(x):  
    def soma(n):  
        return x + n  
    return soma
```

```
>>> soma1 = somador(1)
```

```
>>> soma1(5)
```

```
6
```

```
>>> soma5 = somador(5)
```

```
>>> soma5(3)
```

```
8
```

```
>>> soma1.func_closure[0].cell_contents
```

```
1
```

```
>>> soma5.func_closure[0].cell_contents
```

```
5
```

**Questões de projeto referentes a funções**

- Efeitos colaterais são permitidos?
  - Funções em Ada podem ter apenas parâmetros de entrada, o que diminui as formas de efeitos colaterais

## Questões de projeto referentes a funções

- Qual tipo de valores podem ser retornados?
  - C/C++ não permite o retorno de arranjos e funções (ponteiros para arranjos e função são permitidos)
  - Ada, Python, Ruby, Lua permitem o retorno de qualquer tipo
  - Ada não permite o retorno de funções, por que função não tem tipo. Ponteiros para funções tem tipo e podem ser retornados



## Questões de projeto referentes a funções

- Quantos valores podem ser retornados?
  - A maioria das linguagens permitem apenas um valor de retorno
  - Python, Ruby e Lua permitem o retorno de mais de um valor

**Subprogramas sobrecarregados e genéricos**

## Subprogramas sobrecarregados

- Um **subprograma sobrecarregado** é um subprograma que tem o mesmo nome de outro subprograma no mesmo ambiente de referenciamento
  - Cada versão precisa ter um único protocolo
  - O significado de uma chamada é determinado pela lista de parâmetros reais (ou/e pelo tipo de retorno, no caso de funções)

- Exemplo da biblioteca padrão do Java

```
public class Arrays {  
    static void sort(byte[] a) { ... }  
    static void sort(byte[] a, int from, int to) { ... }  
    static void sort(short[] a) { ... }  
    static void sort(short[] a, int from, int to) { ... }  
}
```

## Subprogramas sobrecarregados

- Quando coerção de parâmetros são permitidas, o processo de distinção fica complicado. Exemplo e C++

```
int f(float x) { ... }
```

```
int f(double x) { ... }
```

```
int a = f(2); // erro de compilação
```

## Subprogramas sobrecarregados

- Subprogramas sobrecarregados com parâmetros padrões podem levar a uma chamada ambígua. Exemplo em C++

```
int f(double x = 1.0) { ... }  
int f() { ... }  
int a = f(); // erro de compilação
```

# Subprogramas sobrecarregados

- Exemplos
  - C não permite subprograma sobrecarregados
  - Python, Lua e outras linguagens de scripts também não permitem
  - C++, Java, Ada e C# permitem (e incluem) subprogramas sobrecarregados
  - Ada pode usar o tipo de retorno da função para fazer distinção entre funções sobrecarregadas

- Vantagem
  - Aumenta a legibilidade
- Desvantagem
  - Dificulta a utilização de reflexão



## Subprogramas genéricos

- Um **subprograma polimórfico** recebe diferentes tipos de parâmetros em diferentes ativações
- Subprogramas sobrecarregados fornecem o chamado **polimorfismo *ad hoc***
- Linguagens que suportam programação OO oferecem o **polimorfismo de subtipo**, quando um parâmetro de um tipo recebe um valor de um subtipo
- Python e Ruby fornecem um tipo mais geral de polimorfismo (em tempo de execução)

## Subprogramas genéricos

- **Polimorfismo paramétrico** é fornecido por um subprograma que recebe parâmetros genéricos que são usados em expressões de tipos que descrevem os tipos dos parâmetros do subprograma
- Os subprogramas com polimorfismo paramétricos são chamados de **subprogramas genéricos**

- Em geral polimorfismo *ad hoc* e paramétrico são estáticos enquanto polimorfismo de subtipo é dinâmico
  - Estático é mais rápido mas pode gerar executáveis maiores (os subprogramas são monomorfizados)
  - Dinâmico é mais flexível mas mais lento

- Ada
  - Um versão do subprograma genérico é criado pelo compilador quando instanciado explicitamente em uma instrução de declaração
  - Precedido pela cláusula `generic` que lista as variáveis genéricas, que podem ser tipos ou outros subprogramas

## Subprogramas genéricos - Exemplo em Ada

```
generic
  type Index_Type is (<>);
  type Element_Type is private;
  type Vector is array (Index_Type range <>) of Element_Type;
  with function ">"(left, right : Element_Type) return Boolean is <>;
  procedure Generic_Sort(List : in out Vector);
  procedure Generic_Sort(List : in out Vector) is
    Temp : Element_Type;
  begin
    for Top in List'First .. Index_Type'Pred(List'Last) loop
      for Bottom in Index_Type'Succ(Top) .. List'Last loop
        if List(Top) > List(Bottom) then
          Temp := List(Top);
          List(Top) := List(Bottom);
          List(Bottom) := Temp;
        end if;
      end loop;
    end loop;
  end Generic_Sort;
```

## Subprogramas genéricos - Exemplo em Ada

```
type Int_Array is array(Integer range <>) of Integer;  
procedure Integer_Sort is new Generic_Sort(  
    Index_Type => Integer,  
    Element_Type => Integer,  
    Vector => Int_Array);
```

- C++
  - Versões do subprograma genérico são criados implicitamente quando o subprograma é chamado ou utilizado com o operador `&`
  - Precedido pela cláusula `template` que lista as variáveis genéricas, que podem ser nomes de tipos, inteiros, etc

## Subprogramas genéricos - Exemplo em C++

```
template <class Type>  
Type max(Type first, Type second) {  
    return first > second ? first : second;  
}
```



## Subprogramas genéricos - Exemplo em C++

```
struct P {};  
  
void test_max() {  
    int a, b, c;  
    char d, e, f;  
    P g, h, i;  
    // instanciação implícita  
    c = max(a, b);  
    f = max(d, e);  
    // instanciação explícita  
    float x = max<float>(1.2, 3);  
    // erro de compilação  
    float y = max(1.2, 3);  
    // erro de compilação, o operador > não foi definido  
    // para o tipo P  
    i = max(g, h);  
}
```

## Subprogramas genéricos - Exemplo em C++

```
template <typename T>
void generic_sort(T list[], int n) {
    for (int top = 0; top < n - 2; top++) {
        for (int bottom = top + 1; bottom < n - 1; bottom++) {
            if (list[top] > list[bottom]) {
                T temp = list[top];
                list[top] = list[bottom];
                list[bottom] = temp;
            }
        }
    }
}
```

## Subprogramas genéricos - Exemplo em C++

```
struct P {};  
  
void test_generic_sort() {  
    int is[] = {10, 5, 6, 3};  
    generic_sort(is, 4);  
    // erro de compilação, o operador > não foi definido  
    // para o tipo P  
    P ps[] = {P(), P(), P()};  
    generic_sort(ps, 3);  
}
```

# Subprogramas genéricos

- Java
  - Adicionado ao Java 5.0
  - As variáveis genéricas são especificadas entre < > antes do tipo de retorno do método

## Subprogramas genéricos - Exemplo em java

```
public class Exemplo {  
    static<T extends Comparable<T>> T max(T a, T b) {  
        return a.compareTo(b) > 0 ? a : b;  
    }  
  
    public static void main(String[] args) {  
        System.out.println(max(3, 4));  
        System.out.println(max(4.0, 3.0));  
  
        // erro de compilação  
        System.out.println(max(4.0, 3));  
    }  
}
```

## Subprogramas genéricos - Exemplo em java

```
public class Exemplo {
    static<T extends Comparable<T>> void sort(T list[]) {
        for (int top = 0; top < list.length - 2; top++) {
            for (int bottom = top + 1; bottom < list.length - 1; bottom++) {
                if (list[top].compareTo(list[bottom]) > 0) {
                    T temp = list[top];
                    list[top] = list[bottom];
                    list[bottom] = temp;
                }
            }
        }
    }
}

...
```

## Subprogramas genéricos - Exemplo em java

...

```
public static void main(String[] args) {  
    Integer[] valores = {3, 5, 2, 5};  
    sort(valores);  
    System.out.println(Arrays.toString(valores));  
  
    Double[] valores2 = {3.0, 5.0, 2.0, 5.0};  
    sort(valores2);  
    System.out.println(Arrays.toString(valores2));  
  
    // erro de compilacao, T não pode ser primitivo  
    int[] valores3 = {3, 5, 2, 5};  
    sort(valores3);  
    System.out.println(Arrays.toString(valores3));  
}  
}
```

- Diferenças entre C++/Ada e Java
  - Os parâmetros genéricos precisam ser classes
  - Apenas uma cópia do método genérico para todas as instanciações
  - É possível especificar restrições sobre as classes que podem ser utilizadas como parâmetros genéricos
  - Parâmetro genéricos do tipo curinga (**wildcard**)



# Subprogramas genéricos

- C#
  - Adicionado ao C# 2005
  - Semelhante ao Java
  - Não tem suporte a tipo curinga
  - Uma versão para cada tipo primitivo

**Operadores sobrecarregados definidos pelo  
usuário**

# Operadores sobrecarregados definidos pelo usuário

- Operadores podem ser sobrecarregados pelo usuário em Ada, C++, Python, Ruby entre outras

- Exemplo Ada (produto escalar de dois arranjos)

```
function "*" (A, B : in Vector_Type) return Integer is
  Sum : Integer := 0;
  begin for Index in A'range loop
    Sum := Sum + A(Index) * B(Index);
  end loop;
  return Sum;
end "*";
```

- Exemplo C++ (produto escalar de dois arranjos)

```
int operator *(vector<int> &A, vector<int> &B) {  
    int sum = 0;  
    for (int i = 0; i < A.size(); i++) {  
        sum += A[i] * B[i];  
    }  
    return sum;  
}
```

- Robert Sebesta, Concepts of programming languages, 9<sup>a</sup> edição. Capítulo 9.