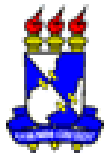
Herança e Polimorfismo (Universal)

Prof. Alberto Costa Neto
alberto@ufs.br

Linguagens de Programação



Departamento de Computação
Universidade Federal de Sergipe

Conteúdo

- Polimorfismo Paramétrico
- Polimorfismo de Inclusão
- Vinculação dinâmica
- Herança
- Interfaces

Polimorfismo

- Tipos:

Ad-hoc	*Coerção*
	Sobrecarga (Overloading)
Universal	*Paramétrico*
	Inclusão

Universal::Paramétrico

- Permite definir abstrações que operam uniformemente sobre uma família de tipos
- São baseados em Tipos Parametrizados:
  - Tipos que possuem outro(s) tipo(s) como parâmetro(s)
- Em Pascal: "**file** of τ"
  - **file** of Char
  - **file** of Real
  - etc...

Universal::Paramétrico

- Em linguagens monomórficas, entretanto, apenas tipos parametrizados *built-in* são providos
  - Programador não pode definir novos tipos parametrizados

em Pascal

```
type IntPar  = record pri, seg : Integer end;
     RealPar = record pri, seg : Real end;
```

```
type Par(T)  = record pri, seg : T end;
     IntPar  = Par(Integer);
     RealPar = Par(Real);
```

Universal::Paramétrico

- Parametrização das estruturas de dados e subprogramas com relação ao tipo do elemento sobre o qual operam
- Subprogramas específicos para cada tipo do elemento

```
int identidade (int x) {
    return x;
}
```

- Subprogramas genéricos

```
T identidade (T x) {
    return x;
}
```

T → T

Tipo como T→ T é chamado de **Politipo** porque pode derivar uma família de muitos tipos

U x T→ T indica que parâmetros podem ser de tipos distintos e que o tipo retornado será o tipo do segundo

Polimorfismo paramétrico

- Uma abstração paramétrica (módulo genérico) é uma abstração sobre uma declaração
  - tem corpo de uma declaração
  - pode ser parametrizada
  - uma instância produz vínculos elaborando o corpo
- Aparece de diversas formas nas linguagens
  - Ada: pacotes genéricos (*generic package*)
  - *Templates* (modelos) em C++
  - *Generics* em Java (a partir da versão 1.5)

Função Paramétrica em C++

```
template <class T>
T identidade (T x) {
    return x;
}
class tData {
    int d, m, a;
}
```

```
main( ) {
    int x;
    float y;
    tData d1, d2;

    x = identidade(1);
    y = identidade(2.5);
    d2 = identidade(d1);
}
```

Função Paramétrica em C++

- Limitação dos tipos devido a **operações** contidas na função paramétrica em C++

```
template <class T>
T maior(T x, T y) {
    return x > y ? x : y;
}
class tData {
    int d, m, a;
}
```

```
main( ) {
    tData d1, d2, d3;
    printf("%d", maior(3, 5));
    printf("%f", maior(3.1, 2.5);
    // d3 = maior(d1, d2);
    // Não sobrecarregou o
    // operador >
}
```

Tipo parametrizado com *template*

```
template <class Elemento>
class Pilha {
   private:
      Elemento[] elementos;
      int tam_max;
      int topo;
   public:
      Pilha(int size){...}
      void desempilhar( ) {...}
      void empilhar(Elemento e) {...}
      Elemento topo( ) {...}
      int vazia( ) {...}
}
```

```
Pilha<int> pi(20);
pi.empilhar(12);
....
Pilha<Figura> pf(15);
...
if (pf.vazia()) { .... }
pf.empilhar(...);
...
```

Generics em Java

```
public interface List<E> {
   void add(E x);
   Iterator<E> iterator();
}
```

```
public interface Iterator<E> {
   E next();
   boolean hasNext();
}
```

```
List<Integer> listaInt = new LinkedList<Integer>();
listaInt.add( new Integer(0) );
Integer x = listaInt.iterator().next();
```

Um pacote genérico em ADA

```
generic
    tam_max : Positive  -- valor parâmetro
    type Elemento is private; -- tipo parâmetro
package Pilha is
    procedure desempilhar( );
    procedure empilhar(e: in Elemento);
    function topo( ) return Elemento;
    function vazia( ) return Boolean;
end Pilha;
```

Implementando Pacote Genérico

```
package body Pilha is
    elementos : array (1.. tam_max) of Elemento;
    topo : Integer range 0..tam_max;

    procedure desempilhar( );
        ... -- implementação aqui
    procedure empilhar(e: in Elemento);
        ... -- implementação aqui
    function topo( ) return Elemento;
        ... -- implementação aqui
    function vazia( ) retrurn Boolean;
        ... -- implementacão aqui
end Pilha;
```

Instanciando o Pacote genérico

```
package pilha_calculadora is
                    new Pilha (20, Float);

type Transaction is record ... end record;
package pilha_auditoria is
                    new Pilha(300,Transaction);
```

Reflexão

- Existe alguma relação entre polimorfismo e checagem de tipo dinâmica ?
- Considere a seguinte função numa **linguagem hipotética dinamicamente tipada:**

```
function soma(a) is
var i,s;
begin
   s:=0;
   for i:=1 to a.length do
      s:=s+a.element(i);
      return s;
end;
```

- É genérica?
- Nestas linguagens, toda variável é polimórfica!

Vinculação Dinâmica

- Linguagens **dinamicamente tipadas** são muito flexíveis, porém inseguras (Ex. Smalltalk)
- A **vinculação dinâmica** é flexível, mas ineficiente pois exige informações e testes em tempo de execução
  - Java: todos os métodos usam esta abordagem
  - C++ e Pascal: apenas os declarados **virtual**

Reflexão

- Como possibilitar código polimórfico sem abdicar das checagens estáticas de tipos?

- Resposta:

Polimorfismo de Inclusão

com *vínculo dinâmico*

(Polimorfismo + Subtipos + Vínculo Dinâmico)

Paradigma: Orientação a Objetos

Polimorfismo de Inclusão

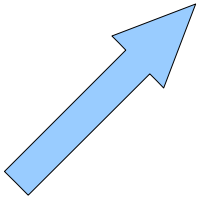
- Característico das linguagens OO
- Elementos dos subtipos são também elementos do supertipo (daí vem o nome inclusão)
- Abstrações formadas a partir do supertipo podem também envolver elementos dos seus subtipos

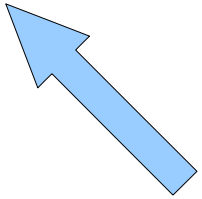
Princípio da Substituição (Liskov)

Se S é um **subtipo** de T, então objetos do tipo T podem ser **substituídos** por objetos do tipo S, **sem alterar qualquer propriedade desejável do programa** (correção, tarefa executada, etc)

Polimorfismo de Inclusão

```
class Figura {
  private float x,y;
  void move(int dx,dy) { x+=dx; y+=dy }
  abstract float area( );
  ...
}
```





```
class Circulo extends Figura {
   private float raio;
   float area( ) {
      return PI*raio*raio;
   }
   ...
}
```

```
class Retangulo extends Figura
   private float alt, comp;
   float area( ) { return alt*comp; }
   ...
}
```

Em Java, extends é um mecanismo de herança mas também define uma relação de subtipos

Polimorfismo de Inclusão

```
Figura[] figs = new Figura[10];
....
float areaTotal = 0;
for (int i=0; i<10; i++)
   areaTotal += figs[i].area( );
```

- figs é uma variável polimórfica pois pelo princípio da substituição figs[i] pode referenciar uma Figura, um Círculo ou um Retângulo
- Na chamada figs[i].area( ), a escolha do método é feita dinamicamente dependendo de que tipo de figura está sendo referenciada por figs[i].

Ampliação

- Ocorre quando uma instância de uma subclasse é atribuída a uma variável da superclasse (o inverso não é possível)

```
Pessoa p, *q;
Empregado e, *r;
q = r;
// r = q;  -- Não é ampliação
// p = e; -- Não é um apontador (faria cópia)
// e = p; -- Não é ampliação
```

Estreitamento

- É o inverso de ampliação. Só é possível quando o objeto sendo estreitado é uma instância do subtipo (ou de seus descendentes)
- Pode causar erros se isso não for garantido
- Java exige conversão explícita, lançando uma ClassCastException quando é inválido
- Pode-se checar o tipo usando **instanceof**

Polimorfismo de Inclusão e Coerção

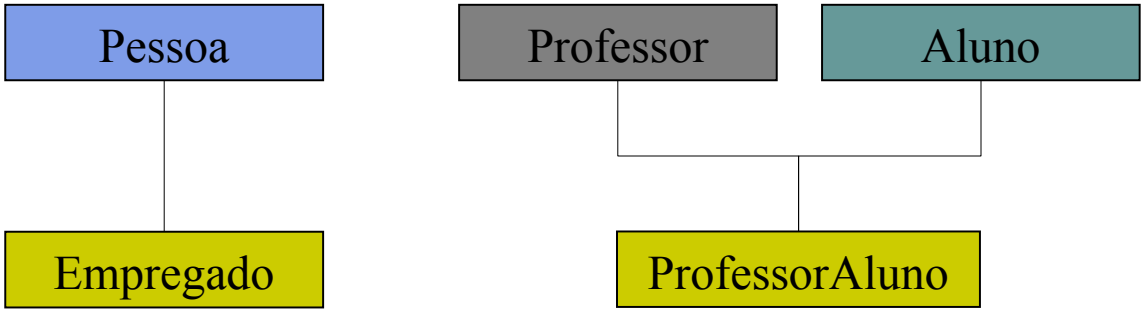
- Em alguns casos, polimorfismo de inclusão sozinho não é suficiente

```
class Pilha {
   private Object[] elems;
   ...
   void empilhar(Object o) {...}
   Object topo( ) {...}
   ...
}
```

```
Pilha p = ...;
p.empilhar("Jõao");
...
String s = (String) p.topo( );
```

Herança Múltipla

- Tipos de Herança
  - Herança simples: herda as características de uma única classe
  - Herança múltipla: herda as características de mais de uma classe
    - Útil para implementar alguns objetos no mundo real



Herança Múltipla

- Suporte das linguagens
  - C++ suporta herança múltipla
  - Smalltalk só suporta herança simples
  - Java suporta herança simples e uma forma restrita de herança múltipla (interfaces)

Problema 1: Colisão de Nomes

- Ocorre quando duas superclasses possuem o mesmo nome para atributos e métodos
- SELF (dialeto de Smalltalk) define uma lista de prioridades entre as classes base (superclasses)
- C++ detecta um erro de ambigüidade e exige o uso do operador de resolução de escopo

Colisão de Nomes

```
class Aluno {
    float nota;
public:
    void imprime();
};

class Professor {
    float salario;
public:
    void imprime();
};
```

```
class ProfessorAluno:
    public Professor, public Aluno {
};

main() {
    ProfessorAluno indeciso;
    // indeciso.imprime();
    // erro de compilação
}
```

Lidando com a Colisão de Nomes

```
class Aluno {
    float nota;
public:
    void imprime();
};

class Professor {
    float salario;
public:
    void imprime();
};
```

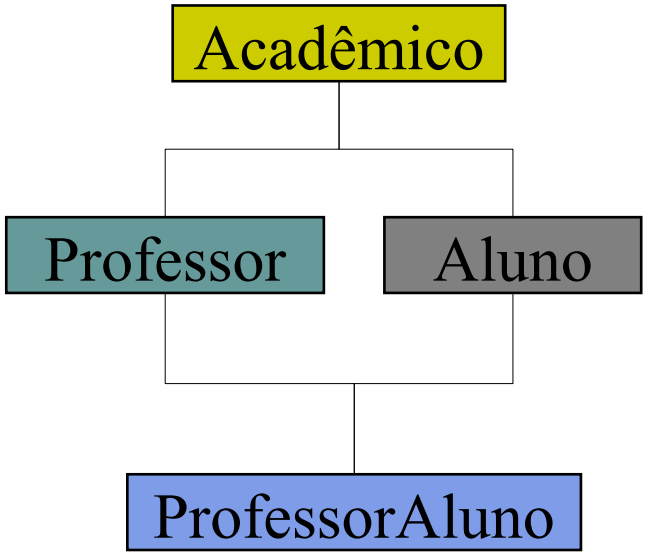
```
class ProfessorAluno:
    public Professor, public Aluno {
public:
    void imprime();
};

void ProfessorAluno::imprime() {
    Aluno::imprime();
}

main() {
    ProfessorAluno decidido;
    decidido.imprime();
}
```

Problema 2: Herança repetida

- Ocorre quando uma classe faz herança múltipla de classes descendentes de uma mesma classe
- Pode causar desperdício de memória
- Provoca colisão de nomes



Herança repetida

- Abordagens
  - Eiffel unifica os atributos em um só
  - C++ fornece um mecanismo especial, utilizando a palavra virtual na definição da classe

Herança repetida

```
class Academico {
    int i, m;
public:
    int idade() { return i; }
    int matricula() {
       return m;
    }
};

class Professor: virtual public
                        Academico {
    float s;
public:
    float salario() {return s;}
};
```

```
class Aluno: virtual public
                    Academico {
    float coef;
public:
    int coeficiente() {
       return coef;
    }
};

class ProfessorAluno:
    public Professor,
    public Aluno {};
```

Herança repetida

- A abordagem de C++ tem 2 problemas:
  - A especificação virtual não é feita na classe em que ocorreu a herança repetida
  - Reduz a expressividade, já que em alguns casos os atributos não deveriam ser compartilhados (ProfessorAluno deveria ter duas matrículas)
- Eiffel requer que sejam definidos quais atributos são compartilhados e repetidos (porém é mais trabalhoso)

Simulando Herança Múltipla

- Em linguagens que só suportam herança simples, é possível substituir a herança múltipla pela delegação
  - Atributo que referencia uma instância da superclasse
  - Métodos envoltórios que só fazem chamadas para os métodos de instância do atributo

Simulando Herança Múltipla

```
class Professor {
    String n = "Alberto";
    int matr = 12345;
    public String nome() { return n;}
    public int matricula() { return
    matr;}
}
```

```
class Aluno {
    String n = "Alberto";
    int matr = 54321;
    float coef = 8.0;
    public String nome() { return n; }
    public int matricula() { return matr; }
    public float coeficiente() {return coef; }
}
```

```
class ProfAluno extends Professor {
    Aluno aluno = new Aluno();
    public float coeficiente() {
       return aluno.coeficiente();
    }
    public int matriculaAluno() {
       return aluno.matricula();
    }
}
```

Desvantagens:

- Subtipagem múltipla deixa de existir
- Requer a implementação dos métodos envoltórios
- Desperdício de memória devido à duplicação de atributos

Interfaces em Java

- Java não permite herança múltipla
- Usa o conceito de interface como uma forma de implementar subtipagem múltipla
- Uma interface é semelhante a uma classe abstrata onde são definidos apenas
  - Protótipos dos métodos (assinatura)
  - Constantes

Interfaces em Java

- Uma classe pode implementar várias interfaces (subtipagem múltipla)
- Uma interface pode estender (herdar) várias interfaces
- Ao implementar uma interface, a classe deve implementar os métodos declarados na interface ou será obrigatoriamente uma classe abstrata

Interfaces em Java

```
interface Aluno {
    void estuda();
    void estagia();
}
```

```
class Graduando
    implements Aluno
{
    public void estuda() { }
    public void estagia() { }
}
```

Interfaces em Java

```
interface Cirurgiao { void opera(); }

interface Neurologista { void consulta(); }

class Medico { public void consulta() { } }

class NeuroCirurgiao extends Medico
      implements Cirurgiao, Neurologista {
   public void opera() { }
}
```

Interfaces em Java

```
public class Hospital {
    static void plantaoCirurgico(Cirurgiao x) { x.opera(); }
    static void atendimentoGeral(Medico x) { x.consulta() }
    static void neuroAtendimento(Neurologista x) { x.consulta() }
    static void neuroCirurgia(NeuroCirurgiao x) { x.opera() }

    public static void main(String[] args) {
       NeuroCirurgiao doutor = new NeuroCirurgiao();
       plantaoCirurgico(doutor);
       atendimentoGeral(doutor);
       neuroAtendimento(doutor);
       neuroCirurgia(doutor);
    }
}
```

Interfaces em Java

- Como não podem conter valores (apenas constantes que são únicas para todos os objetos) não têm os problemas de conflitos de nomes
- Como não implementam métodos, não há problema de herança repetida
- Interface é uma solução elegante para esses problemas e permite a subtipagem múltipla

Sugestões de Leitura

- Concepts of Programming Languages (Robert Sebesta)
  - Seções 9.8, 11.5 a 11.7 e Capítulo 12
- Programming Language Concepts and Paradigms (David Watt)
  - Seções 6.3, 8.1 e 8.2
- Linguagens de Programação (Flávio Varejão)
  - Capítulo 7