Polimorfismo

Modelos de Linguagens de Programação

Visão geral

- Polimorfismo se refere à possibilidade de se criar código capaz de operar sobre valores de tipos diferentes [Varejão 2004]
- Toda a linguagem de programação possui algum tipo de polimorfismo
- Quanto mais polimórfico é o sistema de tipos, maior é a possibilidade de se criar código reutilizável [Varejão 2004]

Conceitos relacionados

- Operadores e operandos:
  - podem ser subprogramas (funções, métodos, procedimentos)
  - podem ser parâmetros
- Verificação de tipos:
  - garante que os operandos de um operador sejam compatíveis
  - pode ser estática (compilação) ou dinâmica (execução)
- Tipos compatíveis:
  - adequados para a operação designada pelo operador
  - convertidos de forma implícita para valores adequados

Tipagem estática

- Vantagens:
  - programas mais confiáveis
  - depuração simplificada
  - maior legibilidade
- Desvantagens:
  - menor redigibilidade (temos que especificar os tipos)
  - desfavorece o reuso, pois muitos algoritmos e estruturas de dados (e.g., coleções) são inerentemente genéricos (i.e., independentes do tipo dos valores manipulados).

Tipagem dinâmica

- Vantagens:
  - retardar a verificação de tipos ou oferecer tipagem dinâmica:

```
// Função em lisp que retorna o segundo elemento de uma lista qualquer:
(defun segundo (l)
   (car(cdr l)))
// pode ser utilizada em qualquer lista:
(segundo (1 2 3))
(segundo (“abacate” “pêssego” “banana”))
```

  - aumenta redigibilidade, reuso

Tipagem dinâmica

- Desvantagens:
  - menor eficiência (*overhead*, checagem feita a todo o momento)
  - redução de legibilidade
  - maior consumo de memória (e.g., informação do tipo é mantida; código de verificação fica em memória)

Verificação de tipos: observações

- Muitas linguagens atuais realizam a **maior parte** das verificações em **tempo de compilação**, mas deixam algumas a serem feitas durante a execução (e.g., C++, Java)
  - Linguagens fortemente tipadas devem possibilitar a detecção de todo e qualquer erro de tipo, de maneira estática ou dinâmica [Cardelli 1991]
- Assim, combinam eficiência e flexibilidade, aumentando a confiabilidade

Sistemas de tipos

- Podem ser:
  - Monomórficos
  - Polimórficos

Tipos monomórficos

- Sistema de tipos exige com que as constantes, variáveis e programas sejam definidos com/para um tipo específico
- E como ficam os algoritmos e estruturas genéricos?
  - necessário criar um tipo conjunto para cada tipo de elementos e operações correspondentes para cada tipo conjunto
  - grande redundância, redigibilidade reduzida
  - conjuntos não podem ser compostos por elementos de tipos diferentes
- Pascal e Modula-2 seguem este padrão (monomórfico), mas Pascal tem subprogramas que aceitam vários tipos (e.g., read, readln, write, writeln, eof)

Tipos polimórficos

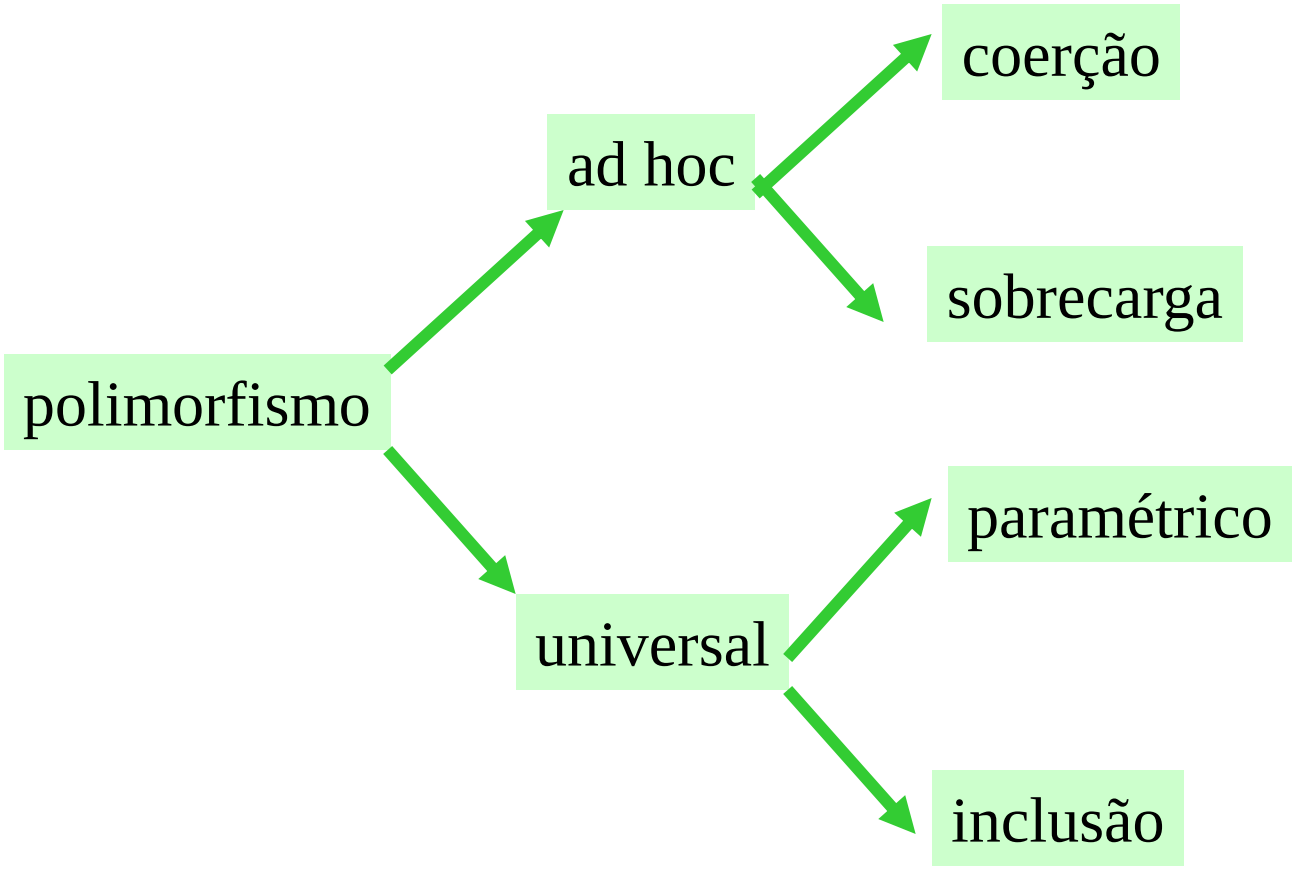
- O sistema de tipos favorece a construção e o uso de estruturas de dados e algoritmos que atuam sobre elementos de tipos diversos
- Tipos de dados polimórficos
  - operações aplicáveis a valores de mais de um tipo
  - e.g., tipo void* em C, permitindo criação de estruturas e algoritmos genéricos
- Subprogramas polimórficos
  - parâmetros ou tipo de retorno podem assumir mais de um tipo
  - e.g., funções com lista de parâmetros variável em C (printf, por exemplo)

Polimorfismo: definições

- Possibilidade de criar código capaz de operar (ou aparentar operar) sobre valores de tipos distintos [Varejão 2004]
- Propriedade de um sistema de tipos que permite escrever abstrações que operam uniformemente sobre uma família de tipos correlatos [Watt 1990]

- OBS: em um sistema polimórfico, cada objeto da linguagem pode pertencer a mais de um tipo [Jazayeri 1998]

Tipos de polimorfismo



Polimorfismo ad hoc

- Polimorfismo aparente
- Ocorre quando um símbolo ou identificador é associado a diferentes trechos de código que atuam sobre diferentes tipos
- Quem lê o código, pensa que ele denota um único trecho de código polimórfico
- Reuso aparente, mas na prática não o faz!
- Subtipos:
  - por coerção
  - por sobrecarga

Ad hoc por coerção

Conversão implícita, podendo ser:

- **Por ampliação:** valor de conjunto mais restrito é convertido para um de conjunto mais amplo
  - mais comum e aceito na maioria das linguagens
  - e.g., int para float
- **Por estreitamento:** tipo mais amplo para o mais restrito
  - pode haver perda de informação (restrito pode não representá-lo corretamente)
  - e.g., float para int (perde a parte fracionária)
- **Nem por ampliação nem por estreitamento:**
  - pode haver perda de informação
  - e.g., int para unsigned (o domínio e a quantidade de valores são os mesmos, mas os conjuntos de valores são diferentes)

Ad hoc por coerção

Exemplo por ampliação:

```
void foo(float a) { }
main () {
    long num;
    foo(num); // chamada com long
}
```

- Aparentemente, a função é capaz de receber floats ou longs. Mas, de fato, ela só funciona com floats (o compilador se encarrega de embutir código para efetivar a conversão)

- Atenção: coerções dão a entender que determinada operação pode ser realizada com operandos de tipos diferentes, mas, na verdade, isso não ocorre: **ocorrem chamadas implícitas à funções de conversão**!

Ad hoc por coerção

- O compilador necessita de uma tabela de conversões permitidas
- Se a conversão é aplicável, ele inclui um código específico para realizá-la:

```
main () {
    int a = 3;
    float x, y = 2.1;
    x = x + y; // ser-se-ia: x = somafloat(x, y);
    x = x + a; // x = somafloat(x, intToFloat(a));
}
```

É como se a função de conversão fosse chamada automaticamente...

Ad hoc por coerção

- Tornam a linguagem mais simples em termos de escrita
- Podem impedir a detecção de certos tipos de erros, reduzindo a confiabilidade
- Algumas alternativas:
  - permitir todos os tipos (ampliação e estreitamento)
  - não permitir coerção
  - só permitir coerção por ampliação (caso de Java)

Ad hoc por sobrecarga

- Sobrecarga = overload
- Ocorre quando um identificador ou operador pode ser usado para designar duas ou mais operações distintas
- Somente aceito quando o seu uso não causa ambiguidades
- Exemplo de sobrecarga do operador “+” em C:

```
void main () {
   int a = 1, b = 2;
   float x = 1.2, y = 2.4;
   x = a + b; // soma de inteiros
   x = x + y; // soma de reais
}
```

Ad hoc por sobrecarga

- Dá a ideia de que a operação pode ser realizada com operandos de tipos diferentes (no exemplo, representa uma função que realiza tanto a operação de somar dois int quanto dois float)
- Na verdade, funções específicas são invocadas para cada uma destas operações:

```
main () {
    int a = 1, b = 2;
    float x = 1.2, y = 2.4;
    x = a + b; // x = somaint(a, b);
    x = x + y; // x = somafloat(x, y);
}
```

Ad hoc por sobrecarga

- C e Modula-2 embutem sobrecarga de operadores, mas os programadores não podem implementar novas sobrecargas
- Pascal é similar, mas oferece subprogramas sobrecarregados na biblioteca padrão, tais como read e write, por exemplo
- Java embute sobrecarga em operadores e programas de sua biblioteca, mas só permite que o programador sobrecarregue subprogramas (métodos)
- C++ e Ada permitem tanto sobrecarga de operadores quanto de subprogramas, mas:
  - Não é possível criar novos operadores
  - Nem a ordem de precedência nem a sintaxe podem ser alteradas
  - C++ impede a sobrecarga de **::** (resolução de escopo), **.** (seleção de membro) e **sizeof**

Ad hoc por sobrecarga

Exemplo em C++:

```
class complex {  // número complexo muito simplificado
   double re, im;
   public:
     //...
     complex operator+ (complex);
     complex operator* (complex);
};
```

Exemplo de uso:

```
complex d = a + b * c;
```

Ad hoc por sobrecarga

Observação:

- O uso do operador é apenas um atalho para uma chamada explícita da função operador
- Exemplo:

```
void f(complex a, complex b)
{
    complex c = a + b;              // atalho
    complex d = a.operator+(b); // chamada explícita
}
```

Ad hoc por sobrecarga

A sobrecarga de funções pode ser:

- **Dependente do contexto** (não usada), quando se distingue apenas pelo valor retornado pela função
- **Independente do contexto,** quando usa listas de parâmetros distintas
  - mais comum e usado
  - tipo de retorno não pode ser usado para diferenciar funções
  - exemplo:

```
void foo(void) { }
void foo(float) { }
void foo(int, int) { }
```

```
main() {
    foo ();
    foo(4.2);
    foo(3, 1);
}
```

Polimorfismo universal

- Polimorfismo verdadeiro
- Ocorre quando:
  a) estrutura de dados pode ser criada incorporando elementos de diversos tipos
  b) mesmo código pode ser aplicado sobre elementos de diferentes tipos
- Permite a programação genérica (definição de unidades genéricas)
- Tipos de polimorfismo universal:
  - paramétrico
  - inclusão

Universal paramétrico

- Abstrações de dados e de controle que atuam uniformemente sobre valores de vários tipos
- Característica principal: parametrização das **estruturas de dados** e **subprogramas** com relação ao tipo do elemento ao qual operam
- Recebem um parâmetro explícito ou implícito adicional que especifica o tipo sobre o qual elas agem
- Gabaritos ou *templates*

Universal paramétrico

- Exemplos:
  - ADA Generic Units
  - Tipos genéricos (como em ML, Python)
  - Templates C++
  - Java Generics

C++ template

```
template <class TipoQQ>
TipoQQ identidade (TipoQQ 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);
   y = identidade(d2); // erro: float vs tdata
}
```

C++ template

```
template <class TipoQQ>
TipoQQ maior (TipoQQ x, TipoQQ y) {
    return x > y ? x : y;
}
class tData { int d, m, a; };
main () {
  tData d1, d2;
  printf("%d", maior (3, 5));
  printf("%f", maior (3.1, 2.5));
     :    :    :
  d1 = maior (d1, d2); // Erro! Por que?
}
```

Java 1.5 Generics

```
:
List<Integer> myIntList = new LinkedList<Integer>();
myIntList.add(new Integer(0));
Integer x = myIntList.iterator().next();
:
List<Float> myFloatList = new LinkedList<Float>();
myIntList.add(new Float(0.0));
Float y = myFloatList.iterator().next();
:
```

Leia mais sobre este assunto em:

- http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf
- http://java.sun.com/j2se/1.5.0/docs/guide/language/generics.html

Java 1.5 Generics

- **Observações:**
  - Tipo parametrizado!
  - Elimina a necessidade de coerção explícita
  - Aumenta robustez: verificação estática de tipo
  - Aumenta legibilidade
  - Não cria múltiplas versões do código
  - Declaração é compilada para todos os tipos
  - Parâmetros formais possuem tipo genérico
  - Na invocação, os tipos dos parâmetros atuais são substituídos pelos tipos dos parâmetros formais
- Nota: algumas linguagens tradicionais já oferecem tipos parametrizados: File, Array e Set de Pascal, por exemplo. Mas eles são pré-definidos!

Universal por inclusão

- Característico de linguagens OO
- Uso de hierarquia de tipos para criar abstrações de dados e controle polimórficas
- Fundamentação:
  - elementos dos subtipos são também elementos do supertipo (por isso, inclusão)
  - abstrações formadas a partir do supertipo também podem envolver elementos dos subtipos
- Mecanismos:
  - Herança (associa à classe herdeira uma representação inicial e um conjunto inicial de métodos)
  - Sobre-escrita (*overhiding*) – polimorfismo por sobrecarga (especialização de métodos; implementação específica para o subtipo)

Universal por inclusão

- Polimorfismo por inclusão permite a criação de trechos de código polimórficos
- Métodos invocados através da mesma variável "referenciadora" de objetos se comportam de maneira diferente, de acordo com o tipo verdadeiro do objeto
- Exemplo: oficina apresentada em laboratório

```
public void manter(Veiculo v) {
   v.vistoria();
   v.conserto();
   v.limpeza();
}
```

  - Dependendo do tipo do veículo (automóvel ou motocicleta), vistoria, conserto e limpeza se comportam de maneiras diferentes

Universal por inclusão

- Mecanismos importantes:
  - Identificação dinâmica de tipos
  - *Upcasting* (ampliação)
  - *Downcasting* (estreitamento)
  - *Late binding* (amarração tardia ou dinâmica) ou *Early binding* (amarração estática)

Universal por inclusão

- Upcasting (ampliação)
  - Capacidade da instância de uma subclasse aparecer quando um membro de sua superclasse é solicitado (subclasse no lugar de superclasse)
  - É legal atribuir membros de uma classe (subtipos) a objetos (referências) dela
  - Exemplos:

```
Pessoa p = new Aluno(); // sendo Aluno subtipo de Pessoa!
```

```
// sendo Navio, Aviao, Onibus subtipos de Transporte
Transporte[] transportes = new Transporte[3];
transportes[0] = new Navio();
transportes[1] = new Aviao();
transportes[2] = new Onibus();
```

Universal por inclusão

- Efeito colateral de Upcasting (ampliação):
  - Não ser possível utilizar os métodos específicos da subclasse
    (lembre-se: o tipo da variável é a classe pai e sua estrutura conhecida é o que ela oferece)

```
   :    :
public class Aluno extends Pessoa{
    private int matricula;
    public void setMatricula(int matricula)
    { this.matricula = matricula; }
    public int getMatricula()
    { return matricula; }
}
   :    :
public static void main(String argumentos[]){
   Pessoa p = new Aluno();
   p.setMatricula(2234);  // Erro (exception), pois o supertipo desconhece o método!
   p.setNome(new String("Nome"));
}
```

Universal por inclusão

- Downcasting (estreitamento):
  - Conversão de um objeto da superclasse para um da classe filha
  - Normalmente as linguagens (a) não o permitem ou (b) exigem que haja conversão explicita (caso do Java)
  - Exemplo (se tivermos um vetor que armazena Objects, ao retornar um elemento devemos fazer o *downcasting* para poder manipulá-lo corretamente):

```
:                :        :

LinkedList listaI = new LinkedList();
for (int i = 0; i < 10; i++)
    listaI.add(new Integer(i)); // upcast de Integer -> Object

:                :        :

for (int i=0; i < listaI.length; i++)
    Integer val1 = (Integer) listaI.getLast(); // downcast (explícito)

:                :        :
```

Atenção: Vetores, Listas e Arrays retornam Object por padrão, a não ser que você use Generics!

Universal por inclusão

- Atenção:
  - Em java, Vetores, Listas e Arrays retornam Object por padrão!
  - Todas as classes são filhas de Object (e herdam seus métodos), podendo sobrescrevê-los
  - Portanto, a linha abaixo é válida, pois Object tem toString:

```
System.out.println(listaI.getLast());
```

  - A linha seguinte também é válida em Java 1.5 (pois tem *auto-boxing* e *auto-unboxing*):

```
int n = listaI.getLast();
```

- **Autoboxing**:
  mecanismo que converte automaticamente um tipo primitivo para sua classe correspondente.
- **Auto-unboxing**:
  Inverso de autoboxing.

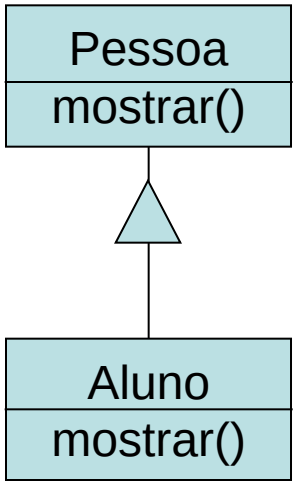
Universal por inclusão

- Amarração estática ou tardia
  - Considere o código abaixo:

```
class Pessoa { //...
  public mostrar(){
   System.out.println("Pessoa");
  }
}
class Aluno extends Pessoa{
  public mostrar(){
   System.out.println("Aluno");
  }
}
```

?????

?????



  - Considere a operação:

```
Pessoa p = new Aluno();
p.mostrar();
```

  - Qual método seria chamado (o "mostrar" de pessoa ou o de aluno)?

Depende do mecanismo de amarração utilizado!
Java usa amarração tardia para métodos de instância (logo, seria o de Aluno)

Universal por inclusão

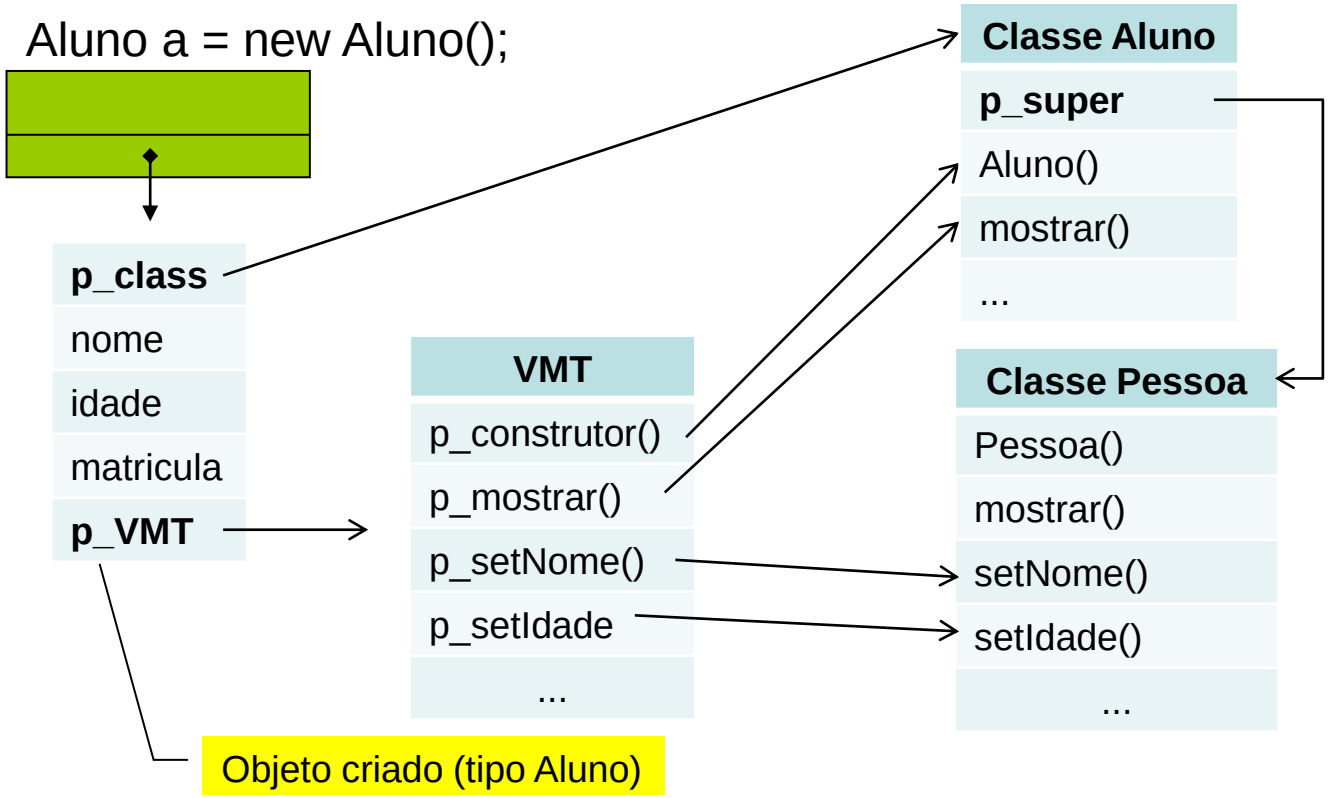
- Amarração tardia (*late/dynamic/virtual binding*)
  - Mecanismo que permite com que a decisão de qual método executar seja tomada em tempo de execução

Durante a compilação, não é possível saber o objeto que será referenciado por uma variável no momento da execução!

  - Permite com que as aplicações invoquem métodos especializados a partir de uma classe base comum
  - Oferece maior flexibilidade para a escrita de código reutilizável
  - Desvantagem: menor eficiência computacional (em C++, há cerca de 15% de custo extra na chamada do método)
  - Não funciona com atributos

Universal por inclusão

- Implementação de amarração tardia:
  - *C++*: virtual tables (vtable)
  - Java: virtual method dispatch / method tables (abaixo)



Universal por inclusão

- Amarração estática (early binding)
  - Em C++ os métodos são estaticamente amarrados por default (independente do tipo do objeto):

```
class Pessoa {
  public:
    void mostrar() { println (“pessoa”); }
}
class Aluno: public Pessoa {
   public:
    void mostrar() { println (“Aluno”); }
}
  :              :
Pessoa *p = new Aluno;
p->mostrar(); // chama o “mostrar” de Pessoa
```

- Atenção: Há linguagens que permitem com que o programador decida como a amarração funcionará

Universal por inclusão

- C++ permite amarração tardia se a palavra-chave virtual for usada antes do método:

```
class Pessoa {
 public:
    void ler() { }                  // estática
    virtual void imprimir() { } // tardia
}
```

Universal por inclusão

- Java permite amarração estática se a palavra-chave final for usada antes do método. No entanto, ele não pode mais ser sobreposto por uma classe-filha!

```
class Aluno {
   public void ler() { }                  // tardia
   final public void imprimir() { } // estática
 }
```

Observação: Em Java, todos os métodos de classe usam amarração estática!

Referências e leitura recomendada

- Jazayeri, M. **Programming language concepts**, 1998.
- Varejão, F. **Linguagens de Programação (Java, C e C++ e outras):** conceitos e técnica. Rio de Janeiro: Elsevier, 2004.
- Watt, D. **Programming languages concepts and paradigms**, 1990.