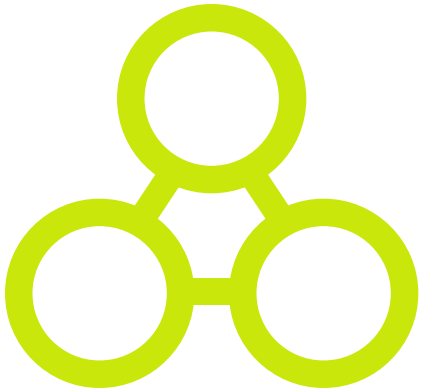


nemo

ontology & conceptual modeling research group

Linguagens de Programação
7 – Polimorfismo

Vítor E. Silva Souza

(vitor.souza@ufes.br)

http://www.inf.ufes.br/~vitorsouza

Departamento de Informática

Centro Tecnológico

Universidade Federal do Espírito Santo

Conteúdo do curso

- Introdução;
- Amarrações;
- Valores e tipos de dados;
- Variáveis e constantes;
- Expressões e comandos;
- Modularização;
- Polimorfismo;
- Exceções;
- Concorrência;
- Avaliação de linguagens.

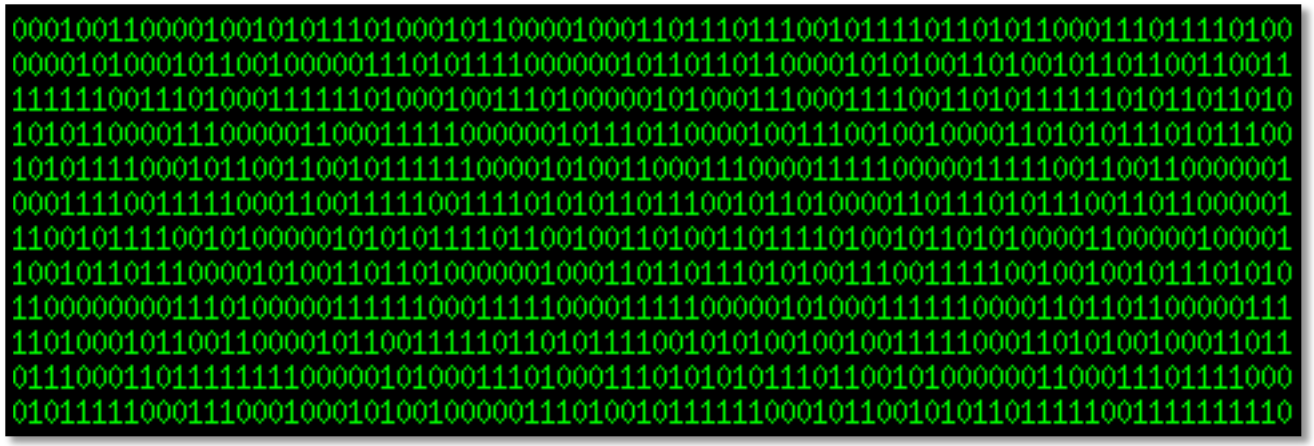
- Estes slides foram baseados em:
  - Slides do prof. Flávio M. Varejão;
  - Livro “Linguagens de Programação – Conceitos e Técnicas” (Varejão);
  - Livro “Linguagens de Programação – Princípios e Paradigmas, 2a edição” (Tucker & Noonan).

Tipos de dados

- Definem um conjunto de valores e as operações aplicáveis sobre esses valores;
- Servem fundamentalmente para oferecer informações relevantes aos programadores e aos compiladores (ou interpretadores) sobre os dados usados pelos programas.

Linguagens de máquina

- Não tipadas porque nessas linguagens existe um único tipo representado pela palavra da memória;
- Programar significa representar tudo (caracteres, números, ponteiros, estruturas de dados, instruções e programas) como cadeias de bits.
  - Se um dado é usado numa operação ADD, pressupõe-se ser um número…



Linguagens de máquina

- Necessário uma interpretação externa para identificar de maneira clara o que está sendo representado nos programas;
- Mesmo em linguagens não tipadas, o conceito de tipos de dados surge naturalmente com a atividade de programação:
  - Números, caracteres e instruções precisam ser tratados de maneira distinta pelo programador.

Linguagens de máquina

- Impossibilidade de evitar violações na organização dos dados:
  - Não há impedimento para o programador efetuar a atribuição de uma instrução a algo que representava um número;
  - Ambos são cadeias de bits, o único tipo realmente reconhecido por essas linguagens.

Linguagens de alto nível

- São tipadas:
  - Cada LP define um sistema de tipos com um conjunto próprio de tipos de dados;
- Sistemas de tipos facilitam o processo de organização dos dados:
  - Oferecendo facilidades para a definição e o uso dos dados;
  - Fornecem garantias de tratamento apropriado.

Linguagens de alto nível

- Linguagens que incluem o tipo booleano facilitam o programador:
  - Criar variáveis desse tipo: não é necessário definir a representação de um valor booleano;
  - Usar essas variáveis: as operações boolenas já estão definidas;
  - Garantir que essas variáveis não serão usadas em operações inapropriadas: por exemplo, adição com um valor inteiro.

Sistemas de tipos

- Servem para descrever de modo mais adequado os dados:
  - Aumenta a legibilidade e a redigibilidade dos programas;
- Evitam que programas executem operações incoerentes:
  - Somar um inteiro e um caractere.

Verificação de tipos

- Atividade de garantir que operandos de um certo operador são de tipos compatíveis;
  - Subprogramas & parâmetros == operadores & operandos;
- Verificação a priori de tipos é normalmente recomendada:
  - Evita a realização de operações sem sentido;
- LPs podem possibilitar ou não a realização de uma ampla verificação de tipos:
  - C adota uma postura fraca;
  - Ada e Java procuram realizar uma verificação extremamente ampla de tipos.

Verificação de tipos

- Feita antes da execução das operações;
- Em geral, o quanto antes melhor:
  - Erros de tipos são identificados mais cedo;
  - Nem sempre isso é possível ou desejável (ex.: ligação dinâmica em LPs OO);
- Algumas LPs possuem brechas no seu sistema de tipos que impedem a realização de algum tipo de verificação
  - Registro variante em Pascal e Modula-2.

Verificação estática de tipos

- Todos os parâmetros e variáveis devem possuir um tipo fixo identificável a partir da análise do texto do programa;
- Tipo de cada expressão pode ser identificado e cada operação pode ser verificada em tempo de compilação:
  - C faz poucas verificações, reduz confiabilidade;
  - Modula-2 faz muitas verificações, reduz redigibilidade.

Verificação dinâmica de tipos

- Em tempo de execução;
- Somente os valores dessas LPs têm tipo fixo:
  - Cada valor tem associado a ele uma *tag* indicando o seu tipo;
- Uma variável ou parâmetro não possui um tipo associado:
  - Pode designar valores de diferentes tipos em pontos distintos da execução;
  - Verificação dos tipos de operandos imediatamente antes da execução da operação;
- Lisp, Basic, APL e Smalltalk.

Verificação de tipos mista

- Maior parte das verificações de tipos em tempo de compilação;
- Algumas verificações em tempo de execução:
  - Programas em LPs orientadas a objetos podem precisar verificar a conversão de tipos de objetos durante a execução;
- C++, Ada e Java.

Linguagens fortemente tipadas

- Conceito muito em voga nos anos 80 e 90;
- Devem possibilitar a detecção de todo e qualquer erro de tipo em seus programas;
- Verificação estática tanto quanto o possível e a verificação dinâmica quando for necessário;
- Algol-68 é fortemente tipada;
- Ada e Java são quase fortemente tipadas.

Verificação estática de tipos

- Considerando o exemplo abaixo:

```
int par (int n) {
  return (n % 2 == 0);
}
int main() {
  int i = 3;
  par (i);
}
```

- Ações do compilador:
  - Operandos de % devem ser inteiros;
  - Os operandos de == devem ser de um mesmo tipo;
  - O tipo de retorno de par deve ser inteiro;
  - O parâmetro real de par deve ser inteiro.

Verificação dinâmica de tipos

- Efetuadas antes da execução das operações:

```
(defun mult-tres (n)
   (* 3 n))
(mult-tres 7)
(mult-tres "abacaxi")
```

- Maior flexibilidade para produzir código reutilizável:

```
(defun segundo (l)
   (car(cdr l)))
(segundo (1 2 3))
(segundo ((1 2 3) (4 5 6)))
(segundo ("manga" "abacaxi" 5 6))
```

- Menor eficiência computacional (verificação em tempo de execução, armazenamento do tipo em memória).

Verificação dinâmica de tipos

- Em Python:

```
def mult_tres(n):
    print "resultado=",3*n

mult_tres(7)
mult_tres("Abacaxi")
```

- Resultado:

```
21
AbacaxiAbacaxiAbacaxi
```

E, como veremos mais à frente, isso tem tudo a ver com polimorfismo!

Inferência de tipos

- LPs estaticamente tipadas não devem necessariamente exigir a declaração explícita de tipos:

```
par (n) {
  return (n % 2 == 0);
}
main() {
  i = 3;
  par (i);
}
```

Qual o tipo do argumento n? E qual o tipo de retorno de par()?

Inferência de tipos

- Um sistema de inferência de tipos pode ser usado para identificar os tipos de cada entidade e expressão do programa:
  - Aumenta redigibilidade, mas pode reduzir legibilidade e facilidade para depuração dos programas;
  - Compiladores são bem mais exigidos;
- Haskell e ML.

```
-- Em Haskell, esta linha é opcional:
igualA1 :: Char -> Bool

-- O compilador infere a definição acima a partir de:
igualA1 c = c=='1'
```

Conversão implícita de tipos

- Situações nas quais se aplica um operando de tipo diferente do esperado por uma operação:
  - C é extremamente liberal;
  - Modula-2 é extremamente rigorosa;
  - Java e Pascal adotam postura intermediária;
- Necessário estabelecer regras nas quais a conversão é possível.

Regras para conversão implícita de tipos

- Específicas para cada tipo de conversão;
- Baseadas no conceito de inclusão de tipos:
  - Permitida se os valores do tipo do operando também são valores do tipo esperado pela operação;
- Baseadas em equivalência de tipos:
  - Estrutural: o conjunto de valores do tipo do operando é o mesmo do tipo esperado pela operação (compara-se as estruturas dos dois tipos);
  - Nominal: equivalentes se e somente se possuem o mesmo nome.

Equivalência estrutural

- Se T e T’ são primitivos, então T e T’ devem ser idênticos:
  - inteiro inteiro
- Se T e T’são produtos cartesianos e T = A x B e T’= A’ x B’, então A A’ e B B’:
  - inteiro x booleano inteiro x booleano
- Se T e T’são uniões e T = A + B e T’= A’ + B’; então A A’ e B B’ ou A B’ e B A’:
  - inteiro + booleano booelano + inteiro
- Se T e T’são mapeamentos e T = A B e T’ = A’ B’; então A A’ e B B’:
  - inteiro booleano inteiro booleano

Estrutural vs. nominal

```
typedef float quilometros;
typedef float milhas;
quilometros converte (milhas m) {
  return 1.6093 * m;
}
main() {
  milhas s = 200;
  quilometros q = converte(s);   // ambas
  s = converte(q);               // estrutural apenas
}
```

- C adota estrutural neste caso (tipos primitivos).

Estrutural vs. nominal

```
typedef struct {
  float f;  int i; char c;
} A;

typedef struct {
  float f;  int i; char c;
} B;

int main(int argc, char *argv[]) {
  A e1;
  B e2;
  e1.f=1.0; e1.i=2; e1.c='a';
  // assigning to 'A' from incompatible type 'B'
  e2=e1;
  printf("%f %i %c",e2.f, e2.i, e2.c);
}
```

- Nominal passa a ser preferida para tipos compostos.

Sistemas de tipos monomórficos

- Todas constantes, variáveis e subprogramas devem ser definidos com um tipo específico;
  - Pascal e Modula-2;
- Simples mas com baixa reusabilidade e redigibilidade;
- Muitos algoritmos e estruturas de dados são inerentemente genéricos:
  - Algoritmo para ordenação independe parcialmente do tipo do elemento a ser ordenado (tipo do elemento precisa ter operação de comparação);
  - Conjuntos e suas operações são totalmente independentes do tipo dos elementos.

Sistemas de tipos monomórficos

- Nenhuma LP é totalmente monomórfica;
- Pascal:
  - read, readln, write, writeln e eof;
  - Vetores, conjuntos, arquivos;
- Pascal e Modula-2:
  - Operadores (como +) atuam sobre diversos tipos;
- Linguagens monomórficas exigem que se crie representação e operações distintas para cada tipo de elemento:
  - Lista de inteiros, lista de cadeia de caracteres, etc.

Sistemas de tipos polimórficos

- Favorecem a construção e uso de estruturas de dados e algoritmos que atuam sobre elementos de tipos diversos;
  - Subprogramas polimórficos são aqueles cujos parâmetros (e, no caso de funções, o tipo de retorno) podem assumir valores de mais de um tipo;
  - Tipos de dados polimórficos são aqueles cujas operações são aplicáveis a valores de mais de um tipo;

Sistemas de tipos polimórficos

- Em C:
  - `void*` possibilita a criação de variáveis e parâmetros cujos valores podem ser ponteiros para tipos quaisquer;
  - Permite a construção de estruturas de dados genéricas;
  - Funções com lista de parâmetros variável também são polimórficas.
- Em Java:
  - Hierarquias polimórficas (ex.: Forma, Quadrado, Círculo, Triângulo, ...);
- Em LPs dinamicamente tipadas (ex.: Python):
  - Tipagem dinâmica permite passar qualquer valor como argumento. Verificação em tempo de execução.

Tipos de polimorfismo

- Classificação de Cardelli e Wegner:

<table>
<tr><td rowspan="4">Polimorfismo</td><td rowspan="2">Ad-hoc</td><td>Coerção</td></tr>
<tr><td>Sobrecarga</td></tr>
<tr><td rowspan="2">Universal</td><td>Paramétrico</td></tr>
<tr><td>Inclusão</td></tr>
</table>

Polimorfismo ad-hoc

- Se aplica apenas a subprogramas;
- Aparentemente proporciona reuso do código de implementação do subprograma, mas na realidade não o faz;
- São criados subprogramas específicos para operar sobre cada tipo admissível;
- Somente proporciona reuso do código de chamada dos subprogramas.

Coerção

Sobrecarga

Polimorfismo universal

- Se aplica a subprogramas e estruturas de dados;
  - Estrutura de dados pode ser criada incorporando elementos de tipos diversos;
  - Mesmo código pode ser executado e atuar sobre elementos de diferentes tipos;
- Proporciona reuso de código tanto na chamada quanto na implementação;
- Considerado o verdadeiro polimorfismo.

Paramétrico

Inclusão

Coerção (ad-hoc)

- Conversão implícita de tipos:

```
void f (float i) { /* ... */ }
int main() {
   long num;
   f (num);
}
```

  - Função `f()` aparenta lidar com `float` e `long`;
  - De fato, `f()` lida apenas com `float`;
  - Compilador se encarrega de embutir código para transformar `long` em `float`;
- C possui tabela de conversões permitidas;
- LPs dinamicamente tipadas não fazem coerção em atribuições.

Coerção (ad-hoc)

- Ampliação:
  - Tipo de menor conjunto de valores para tipo de maior conjunto;
  - Operação segura pois valor do tipo menor necessariamente tem correspondente no tipo maior;
- Estreitamento:
  - Tipo de maior conjunto de valores para tipo de menor conjunto;
  - Operação insegura pois pode haver perda de informação;
- Nem sempre é ampliação ou estreitamento:
  - De `int` para `unsigned` em C.

Coerção (ad-hoc)

- Dão a entender que determinada operação ou subprograma pode ser realizada com operandos de tipos diferentes mas não é isso o que ocorre:

```
int main() {
  int w = 3;
  float x = 0;
  float y = 5.6;
  x = x + y;     // x = somafloat(x, y)
  x = x + w;     // x = somafloat(x, intToFloat(w))
  w = x + y;     // w = floatToInt(somaFloat(x, y))
}
```

Coerção (ad-hoc)

- Ocorre com grande frequência em atribuições e expressões em C:

```
int main() {
  int i;
  char c = ‘a’;
  float x;
  i = c;
  c = i + 1;
  x = i;
  i = x / 7;
}
```

Coerção (ad-hoc)

- Deve se ter cuidado com perdas:

```
int main(int argc, char *argv[]) {
  int i;
  float j=2.5;
  float k=1234.5*10000*10000;
  i=j;

  // i=2 j=2.500000
  printf("i=%d\tj=%f\n",i,j);
  i=k;

  // i=-2147483648 k=123449999360.000000
  printf("i=%d\tk=%f\n",i,k);
}
```

Coerção (ad-hoc)

- Existem opiniões controversas a seu respeito:
  - Maior redigibilidade por não demandar chamada de funções de conversão;
  - Menor confiabilidade pois podem impedir a detecção de certos tipos de erros por parte do compilador.

```
int main() {
  int a, b = 2, c = 3;
  float d, e;
  d = a * b;    // Desnecessário: d = (float)(a * b), :)
  e = b * d;    // Se o desejado fosse: e = b * c ?   :(
}
```

Coerção (ad-hoc)

- Ada e Modula-2 não admitem coerções;
- C adota uma postura bastante permissiva;
- Java busca um meio termo só admitindo a realização de coerções para tipos mais amplos.

```
byte a, b = 10, c = 10;
int d;

d = b;
c = (byte) d;
a = (byte) (b + c);
```

Sobrecarga (ad-hoc)

- Quando identificador ou operador é usado para designar duas ou mais operações distintas;
- Aceitável quando o uso do operador ou identificador não é ambíguo:
  - A operação apropriada pode ser identificada usando-se as informações do contexto de aplicação.

Sobrecarga (ad-hoc)

- Operador "**–**" de C pode designar:
  - Negações inteiras:
    - (int → int ou short → short ou long → long)
  - Negações reais:
    - (float → float ou double → double)
  - Subtrações inteiras:
    - (int x int → int)
  - Subtrações reais:
    - (float x float → float)
- E muitas outras (não listamos todos os tipos numéricos).

Sobrecarga (ad-hoc)

- Sugere que determinada operação ou subprograma pode ser realizada com operandos de tipos diferentes mas não é isso que ocorre:

```
int main() {
  int a = 2, b = 3;
  float x = 1.5, y = 3.4;
  a = a + b;                  // a = somaint (a, b);
  x = x + y;                  // x = somafloat (x, y);
}
```

Sobrecarga (ad-hoc)

- Modula-2 e C:
  - Embutem sobrecarga em seus operadores;
  - Programadores não podem implementar novas sobrecargas de operadores;
  - Não existe qualquer sobrecarga de subprogramas;
- Pascal:
  - Existem subprogramas sobrecarregados na biblioteca padrão (ex.: `read()` e `write()`);
  - Programadores não podem implementar novas sobrecargas de subprogramas.

Sobrecarga (ad-hoc)

- Java:
  - Embute sobrecarga em operadores e em subprogramas de suas bibliotecas;
  - Só subprogramas podem ser sobrecarregados pelo programador;
- Ada, C++ e Python:
  - Adotam a postura mais ampla e ortogonal;
  - Realizam e permitem que programadores realizem sobrecarga de subprogramas e operadores;
  - Não admitem a criação de novos operadores;
  - Sintaxe e precedência não pode ser alterada.

Sobrecarga em C++

```
class UmValor {
  int v;

public:
  UmValor() { v = 0; }
  UmValor(int j) { v = j; }

  const UmValor operator+(const UmValor& u) const {
     return UmValor(v + u.v);
  }

  UmValor& operator+=(const UmValor& u) {
     v += u.v;
     return *this;
  }

// Continua...
```

Sobrecarga em C++

```
  const UmValor& operator++() { // prefixado
     v++;
     return *this;
  }

  const UmValor operator++(int) { // posfixado
     UmValor antes(v);
     v++;
     return antes;
  }
};

// Código cliente (exemplo de uso):
// UmValor r(1), s(2), t;
// t += r + s;
// r = ++s;
// s = t++;
```

Sobrecarga em C++

- Útil na criação de objetos em diferentes contextos;
- Sintaxe esquisita para sobrecarga de `++` e `--`;
- Nem todos operadores podem ser sobrecarregados. Exceções são:
  - `::` (resolução de escopo);
  - `.` (seleção de membro);
  - `sizeof` (tamanho do objeto/tipo).

Sobrecarga (ad-hoc)

- Independente de Contexto:
  - Lista de parâmetros diferenciada em número ou tipo dos parâmetros;
  - Tipo de retorno não pode ser usado para diferenciação;
  - Java e C++;
- Dependente de Contexto:
  - Necessário apenas uma assinatura diferenciada;
  - Tipo de retorno pode ser usado para diferenciação;
  - Exige mais esforço dos compiladores;
  - Pode provocar erros de ambiguidade;
  - Ada.

Sobrecarga independente de contexto

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

// Não pode diferenciar por tipo de retorno:
// int f(void) { }

int main() {
    f();
    f(2.3);
    f(4, 5);
    f(2.2f, 7.3f);

    // Pode gerar ambiguidade por causa da coerção:
    // f(3, 5.1f);
    // f(1l, 2l);
}
```

Sobrecarga dependente de contexto

- Operador / designa:
  - Divisão real (float x float → float);
  - Divisão inteira (integer x integer → integer);
- Ambiguidade mesmo sem coerção;
- Sobrecarga de / (integer x integer → float):

```
function "/" (m, n : integer)  return float is begin
   return float (m) / float (n);
end "/";

n : integer; x : float;
x: = 7.0/2.0;          -- calcula 7.0/2.0 = 3.5
x: = 7/2;              -- calcula 7/2 = 3.5
n: = 7/2;              -- calcula 7/2 = 3
n: = (7/2) / (5/2);  -- calcula (7/2)/(5/2) = 3/2 = 1
x: = (7/2) / (5/2);  -- erro: ambiguidade (1.4 ou 1.5)
```

Sobrecarga (ad-hoc)

- Nem todos consideram uma característica desejável a possibilidade dos programadores sobrecarregarem os operadores;
- Programas podem ficar mais fáceis de serem lidos e redigidos com a sobrecarga;
- Aumenta a complexidade da LP;
- Pode ser mal utilizado, tendo efeito contrário a legibilidade;
- Java não inclui a sobrecarga de operadores por considerá-la capaz de gerar confusões e aumentar a complexidade da LP.

Paramétrico (universal)

- Parametrização das estruturas de dados e subprogramas com relação ao tipo do elemento sobre o qual operam:
  - Abstrações recebem um parâmetro implícito adicional especificando o tipo sobre o qual elas agem;
- Subprogramas específicos para cada tipo do elemento:

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

- Subprogramas genéricos:

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

Politipo

- Subprogramas genéricos possuem parâmetro tipo:
  - `T` é parâmetro tipo em `T identidade(T x);`
  - Tipo retornado por identidade será o mesmo usado na chamada:

```
x = identidade (3.2);  // x receberá um float
```

  - Tipo de identidade é sua assinatura `T → T`;
  - Tipo como `T → T` é chamado de politipo porque pode derivar uma família de muitos tipos;
- Não existe impedimento em se usar mais de um parâmetro tipo em um politipo:
  - `U x T → T` indica que parâmetros podem ser de tipos distintos e que o tipo retornado será o tipo do 2º.

Paramétrico em C++

- Uso de template:

```
template <class T>
T identidade (T x) {
  return x;
}

class tData {
  int d, m, a;
};
```

```
int main () {
  int x;
  float y;
  tData d1, d2;
  x = identidade (1);
  y = identidade (2.5);
  d2 = identidade (d1);
  // y = identidade (d2);
}
```

Paramétrico em C++

- Muitas funções são parcialmente genéricas:

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

```
int main () {
  tData d1, d2;
  printf ("%d", maior(3, 5));
  printf ("%f",
       maior(3.1, 2.5));
  // d1 = maior (d1, d2);
}
```

- Erro de compilação porque o operador > não está definido para a classe `tData`;
- Necessário sobrecarregar o operador > para a classe `tData`.

Paramétrico em C++

- É possível parametrizar classes:

```
template <class T, int tam>
class tPilha {
  T elem[tam];
  int topo;

public:
  tPilha(void) { topo = -1; }
  int vazia (void) { return topo == -1; }
  void empilha (T);
  void desempilha (void);
  T obtemTopo (void);
};
```

Paramétrico em C++

```
template <class T, int tam>
void tPilha<T, tam>::empilha (T el){
  if (topo < tam - 1)
     elem[++topo] = el;
}

template <class T, int tam>
void tPilha<T, tam>::desempilha (void){
  if (! vazia()) topo--;
}

template <class T, int tam>
T tPilha<T, tam>::obtemTopo (void) {
  if (! this->vazia()) return elem[topo];
}
```

Paramétrico em C++

```
class tData {
  int d, m, a;
};

int main () {
  tData d1, d2;
  tPilha <int, 3> x;
  tPilha <tData, 2> y;
  x.empilha (1);
  y.empilha (d1);
  x.empilha (3);
  y.empilha (d2);
  while (! x.vazia() || ! y.vazia()) {
     x.desempilha();
     y.desempilha();
  }
}
```

Implementação de template em C++

- Só possibilita a reutilização de código fonte;
- Não é possível compilar o código usuário separadamente do código de implementação;
- O compilador C++ necessita saber quais tipos serão associados ao template:
  - Faz varredura do código usuário;
  - Replica todo o código de implementação para cada tipo utilizado;
- Compilador necessita verificar se o tipo usado é compatível com as operações definidas nas implementações e saber o tamanho a ser alocado.

Paramétrico (universal)

- Ada:
  - Pacotes genéricos;
- Java:
  - Tipos genéricos a partir do Java 5.

```
public class Casulo<T> {
  private T elemento;
  public void colocar(T elem) {
     elemento = elem;
  }
  public T retirar() {
     return elemento;
  }
}
```

Tipos genéricos em Java

- Referência e construtor definem o tipo manipulado pela classe genérica;
- Compilador pode efetuar checagens de tipo;
- LP fica mais complicada quando se considera curingas, herança, etc.

```
Casulo<String> cs = new Casulo<String>();
cs.colocar("Uma string");
// Erro: cs.colocar(new Integer(10));
String s = cs.retirar();

Casulo<Object> co = new Casulo<Object>();
co.colocar("Uma string");
co.colocar(new Integer(10));
Object o = co.retirar();
```

Tipos genéricos em Java

- Tipo genérico pode ser limitado (funções/tipos parcialmente genéricos):

```
import java.util.Date;

public class Teste {
  static <T extends Comparable<T>> T maior(T x, T y) {
     return (x.compareTo(y) > 0) ? x : y;
  }

  public static void main(String[] args) {
     Date d1 = new Date(), d2 = new Date();
     System.out.println(maior(3, 5));
     System.out.println(maior(3.1, 2.5));
     System.out.println(maior(d1, d2));
  }
}
```

Inclusão (universal)

- Característico de linguagens orientadas a objetos;
- Uso de hierarquia de tipos para criação de subprogramas e estruturas de dados polimórficas;
- Ideia Fundamental:
  - Elementos dos subtipos são também elementos do supertipo;
  - Abstrações formadas a partir do supertipo podem também envolver elementos dos seus subtipos.

Inclusão (universal)

- S subtipo de T implica:
  - S formado por um subconjunto dos valores de T: todo valor de S é também um valor de T;
  - As operações associadas ao tipo T são aplicáveis ao subtipo S: S herda todas as operações do tipo T;
- Conceito de tipo implementado através de classes em linguagens orientadas a objetos.

Herança

- Subclasses herdam os atributos e métodos de uma classe e, portanto, implementam subtipos do tipo definido por essa classe;
- Herança associa à subclasse:
  - Uma representação inicial para os objetos dessa classe (os atributos herdados);
  - Um conjunto inicial de métodos aplicáveis aos objetos dessa classe (os métodos herdados);
- Subclasse pode conter atributos e métodos adicionais, especializando o estado e o comportamento dos objetos da subclasse.

Herança em Java

```
public class Produto {
  protected String nome;
  protected double preco;

  public Produto(String nome, double preco) {
     this.nome = nome;
     this.preco = preco;
  }

  public boolean ehCaro() {
     return (preco > 1000);
  }

  /* Métodos de acesso ... */
}
```

Herança em Java

```
public class Livro extends Produto {
  private String autor;
  private int paginas;

  public Livro(String nome, double preco,
               String autor, int paginas) {
     super(nome, preco);
     this.autor = autor;
     this.paginas = paginas;
  }

  public boolean ehGrande() {
     return (paginas > 200);
  }
}
```

Herança em Java

```
public class Loja {
  public static void main(String[] args) {
     Produto p = new Produto("HD externo", 500);
     System.out.println(l.ehCaro());

     Livro l;
     l = new Livro("Linguagem de Programação",
          74.90, "Flávio Varejão", 334);

     System.out.println(l.ehCaro());
     System.out.println(l.ehGrande());
  }
}
```

Vantagens da herança

- Aumenta a reusabilidade do código:
  - Desnecessário redefinir os atributos e métodos da classe `Produto` na classe `Livro`;
- Herança é polimorfismo universal:
  - O método eh`Caro()`, usado para verificar se um `Produto` é caro, é também aplicado para verificar o mesmo em `Livro`.

Especificador de acesso para herdeiras

- Algumas situações requerem que classes herdeiras tenham acesso livre aos atributos da classe herdada;
- A alternativa de fazer esses atributos públicos ou criar métodos públicos de acesso pode não ser satisfatória porque torna esses atributos e métodos acessíveis para métodos de qualquer outra classe;
- Novo especificador de acesso para classes herdeiras: protected.

```
public class Produto {
  protected String nome;
  protected double preco;

// ...
```

Inicialização de atributos com herança

```
class Computador {
  public Computador() {
     System.out.println("Computador()");
     ligar();
  }
  public void ligar() { }
}

class Notebook extends Computador {
  private int codigo;
  public Notebook() {
     System.out.println("Notebook()");
     codigo = 12345;
  }
  public void ligar() {
     System.out.println("Código " + codigo);
  }
}
```

Inicialização de atributos com herança

- O construtor da superclasse é chamado antes do código receber seu valor:

```
public class Teste {
  public static void main(String[] args) {
     new Notebook();
  }
}

// Resultado:
// Computador()
// Código 0
// Notebook()
```

Sobrescrita de métodos (Java)

- Se um método herdado não satisfaz, podemos redefini-lo (sobrescrevê-lo):

```
public class Livro extends Produto {

  /* Definições anteriores... */

  // Livros acima de R$ 200 são caros!
  public boolean ehCaro() {
     return (preco > 200);
  }

}
```

Sobrescrita de métodos (Java)

- Métodos sobrescritos podem chamar sua versão na superclasse usando a palavra super:

```
public class Produto {
  /* ... */
  public void imprimir() {
     System.out.println(nome + "," + preco);
  }
}
public class Livro extends Produto {
  /* ... */
  public void imprimir() {
     super.imprimir();
     System.out.println(autor + "," + paginas);
  }
}
```

Sobrescrita extends Sobrecarga

- Dois métodos de mesmo nome na mesma classe == sobrecarga;
- Um método na superclasse sobrescrito na subclasse ==
  Dois métodos de mesmo nome ==
  Sobrecarga!
- Pode-se considerar, então, que a sobrescrita cria polimorfismo de sobrecarga;
  - Lembre-se que todo método tem um parâmetro implícito: o objeto no qual ele é chamado (this).

Identificação dinâmica de tipos

- Forma de identificar o tipo do objeto em tempo de execução:
  - Útil em situações nas quais o programador desconhece o tipo verdadeiro de um objeto;
  - Permite elaboração de trechos de código nos quais métodos invocados por um mesmo referenciador de objetos se comportam de maneira diferenciada.

Ampliação ou upcast

- Instância (ou instância direta):
  - Objetos de uma classe;
- Membro (ou instância indireta):
  - Todas as instâncias da classe e de suas subclasses;
- É permitido atribuir qualquer membro a uma referência à classe;
- Ampliação:
  - Termo usado para descrever o movimento de objetos na sua linha de ancestrais no sentido da subclasse para as superclasses.

Ampliação em Java

```
public class Pessoa {
  public void comprar(Produto p) { /* ... */ };

  public void ler(Livro l) { /* ... */ };

  public static void main(String[] args) {
     Produto hd = new Produto("HD externo", 500);
     Livro livro = new Livro("Linguagem de Programação",
          74.90, "Flávio Varejão", 334);
     Pessoa cliente = new Pessoa();

     cliente.comprar(hd);
     cliente.comprar(livro);

     cliente.ler(livro);     // cliente.ler(hd) não.
  }
}
```

Ampliação em C++

- Só através de ponteiros ou referências:

```
Produto p, *q;
Livro l, *r;
q = r;
// r = q;
// p = l;
// l = p;
```

  - Limitação é consequência do mecanismo de cópia de objetos utilizado pela operação de atribuição e para passagem de parâmetros por valor.

Amarração tardia de tipos

- Definição dinâmica do método a ser executado:
  - Depende do objeto que invoca o método.

```
public class Produto {
  public boolean ehCaro() {
     return (preco > 1000);
  }

  /* ... */
}

public class Livro extends Produto {
  public boolean ehCaro() {
     return (preco > 200);
  }

  /* ... */
}
```

Amarração tardia de tipos (em Java)

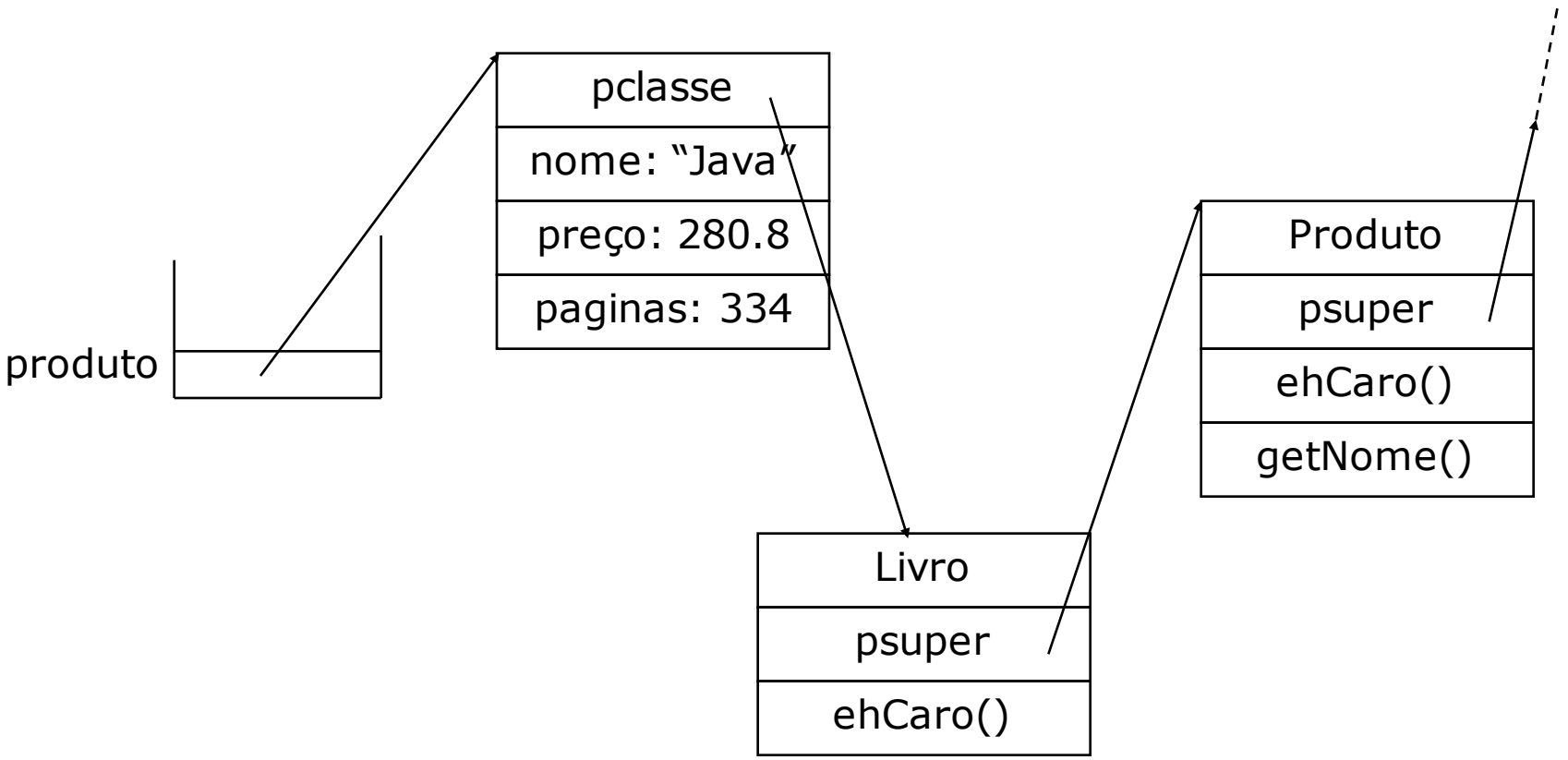
```
public class Teste {
  public static void main(String[] args) {
     Produto produto = new Livro("Java – Como Programar",
        280.8, "Deitel", 600);

     System.out.println(produto.getNome());

     // true ou false?
     System.out.println(produto.ehCaro());
  }
}
```

Amarração tardia/dinâmica é menos eficiente que a estática.

Amarração tardia de tipos



Amarração tardia vs. estática em C++

- Todas chamadas utilizam amarração tardia em Java;
- Em C++ o implementador da classe pode decidir se deseja o uso de amarração tardia ou não para cada método da classe:
  - Visa não comprometer a eficiência de execução desnecessariamente;
  - Uso da palavra `virtual`:

```
class Pessoa {
public:
  void ler() {}
  virtual void imprimir() {}
};
```

Amarração tardia de tipos

- Possibilita a criação de código usuário com polimorfismo universal:

```
class Forma {
  public void desenhar() {
     System.out.println("Forma");
  }
}
class Circulo extends Forma {
  public void desenhar() {
     System.out.println("Círculo");
  }
}
class Quadrado extends Forma { /* ... */ }
class Triangulo extends Forma { /* ... */ }
```

Amarração tardia de tipos

```
public class Teste {
  private static void desenha(Forma[] fs) {
     for (int i = 0; i < fs.length; i++)
        fs[i].desenhar();
  }

  public static void main(String[] args) {
     Forma[] formas = new Forma[] {
        new Circulo(), new Forma(),
        new Quadrado(), new Triangulo()
     };
     desenha(formas);
  }
}
```

Imagine um software de desenho carregando um arquivo de desenho salvo.

Classes abstratas

- Possuem membros, mas não possuem instâncias;
  - Membros são as instâncias de suas subclasses concretas (não abstratas);
  - Proibida a criação de instâncias dessas classes;
- Deve ser necessariamente estendida;
- Úteis quando uma classe, ancestral comum para um conjunto de classes, se torna tão geral a ponto de não ser possível ou razoável ter instâncias dessa classe.

Classes abstratas em Java

- Em Java, uso do especificador abstract:

```
// Formas genéricas não existem. Proibido instanciar.
abstract class Forma {
  public void desenhar() {
     System.out.println("Forma");
  }
}

// Causaria erro de compilação usar:
// Forma f = new Forma();
```

Métodos abstratos

- Métodos declarados na classe, mas não implementados:
  - A implementação desses métodos é deixada para as subclasses;
- Classes abstratas normalmente possuem um ou mais métodos abstratos;
- Se uma classe possui um método abstrato, ela deve necessariamente ser uma classe abstrata;
- Especificam protocolo entre a classe e suas subclasses;
- Devem ser implementados pelas subclasses concretas.

Métodos abstratos em Java

- Novamente, uso do especificador `abstract`:

```
// Formas genéricas não existem. Proibido instanciar.
abstract class Forma {
  // Como desenhar uma forma mesmo?
  public abstract void desenhar();
}

// Causaria erro de compilação definir classe concreta
// sem sobrescrever o método:
// class Losango extends Forma { }
```

Métodos abstratos em C++

- Uso da palavra chave `virtual` e do sufixo = 0:

```
class Militar {
public:
  virtual void operacao()=0;
  void imprime { cout << “Militar”; }
};

class Exercito : public Militar {
public:
  void operacao() { cout << "Marchar"; }
  void imprime() { cout << "Exercito"; }
};

class Marinha: public Militar {
public:
  void operacao() { cout << "Navegar"; }
  void imprime() { cout << "Marinha"; }
};
```

Classes abstratas puras

- Todos os métodos são abstratos;
- Usadas para disciplinar a construção de classes;
- Java define o conceito de interface para a sua implementação:
  - Não possuem atributos de instância;
  - Métodos são todos públicos e abstratos.

```
// Métodos são automaticamente públicos e abstratos.
interface Forma {
  void desenhar();
}
```

Estreitamento (*downcast*)

- Termo usado para descrever a conversão de tipos de objetos no sentido da superclasse para as subclasses;
- Não é completamente seguro para o sistema de tipos porque um membro da superclasse não é necessariamente do mesmo tipo da subclasse para a qual se faz a conversão;
  - Algumas LPs não permitem o estreitamento;
  - Outras exigem que ele seja feito através de uma operação de conversão explícita.

Estreitamento (*downcast*)

- Java somente permite a realização de estreitamento através de conversão explícita:
  - Caso a conversão seja feita entre classes não pertencentes a uma mesma linha de descendência na hierarquia, ocorrerá erro de compilação;
  - Caso a conversão seja na mesma linha de descendência, mas o objeto designado pela superclasse não seja membro da classe para o qual se faz o estreitamento, ocorrerá uma exceção em tempo de execução;
  - O operador `instanceof` permite testar dinamicamente se o objeto designado pela superclasse realmente é da classe para a qual se deseja fazer a conversão

Estreitamento em Java

```
class UmaClasse {}
class UmaSubclasse extends UmaClasse {}
class OutraSubclasse extends UmaClasse{}

public class Estreitamento {
  public static void main (String[] args) {
     UmaClasse uc = new UmaSubclasse();
     UmaSubclasse us = (UmaSubclasse) uc;
     OutraSubclasse os;
     // os = (OutraSubclasse) us;
     // os = (OutraSubclasse) uc;
     if (uc instanceof OutraSubclasse)
        os = (OutraSubclasse) uc;
  }
}
```

Estreitamento em C++

- Conversão explícita tradicional:
  - Feita em tempo de compilação sem qualquer verificação;
  - cast.
- Conversão explícita estática:
  - Feita em tempo de compilação;
  - Verifica se conversão ocorre em uma linha de descendência;
  - static_cast.

Estreitamento em C++

- Conversão explícita dinâmica:
  - Feita em tempo de execução (como Java);
  - Verifica se conversão é para o tipo correto, retorna nulo caso contrário;
  - Funciona apenas em classes com funções virtuais;
  - dynamic_cast;
- Teste dinâmico de tipos:
  - Feito em tempo de execução (como o instanceof);
  - Verifica qual o tipo referenciado por ponteiro
  - typeid.

Estreitamento em C++

```
#include <typeinfo>
class UmaClasse {
public:
  virtual void temVirtual () {}
};

class UmaSubclasse: public UmaClasse {};
class OutraSubclasse: public UmaClasse {};
class OutraClasse {};

int main () {
  // Primeira parte do exemplo:
  UmaClasse* pc = new UmaSubclasse;
  OutraSubclasse* pos =
       dynamic_cast <OutraSubclasse*> (pc);
  UmaSubclasse* ps = dynamic_cast <UmaSubclasse*> (pc);
```

Estreitamento em C++

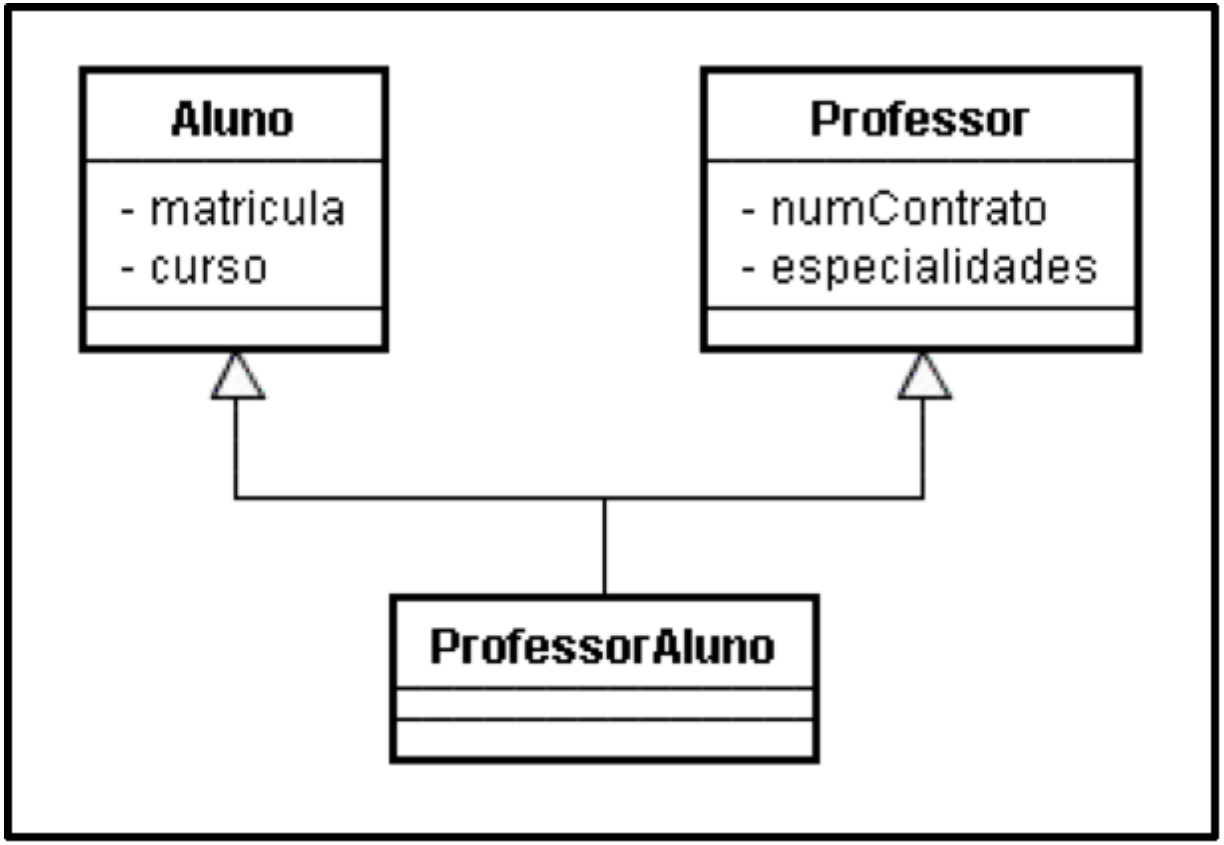
```
  // Segunda parte do exemplo:
  UmaSubclasse us;
  pc = static_cast <UmaClasse*> (&us);
  pc = &us;

  OutraClasse* poc = (OutraClasse*) pc;
  // OutraClasse* poc = static_cast <OutraClasse*> (pc);

  // Terceira parte do exemplo:
  if (typeid(pc) == typeid(ps))
     ps = static_cast<UmaSubclasse*>(pc);
  if (typeid(pc) == typeid(pos))
     pos = static_cast<OutraSubclasse*>(pc);
}
```

Herança múltipla

- Algumas vezes queremos herdar comportamento de duas classes:



Herança múltipla

- Algumas LPs não permitem herança múltipla:

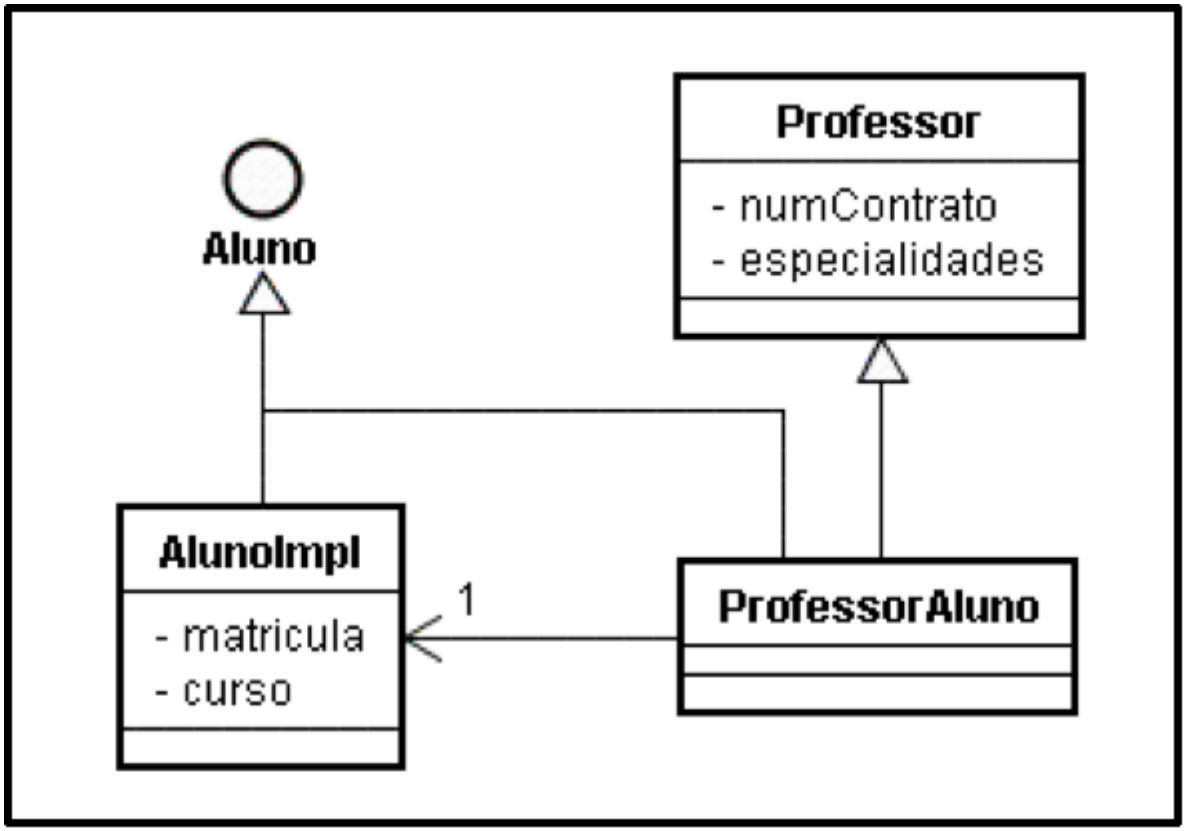
```
// Não permitido em Java, pois não tem herança múltipla:
class ProfessorAluno extends Professor, Aluno {

}
```

- Alternativas:
  - Composição;
  - Uso de métodos envoltórios;
  - Impede subtipagem múltipla.

Solução para herança múltipla em Java

- Escolha uma das classes concretas para herdar;
- Para as demais, crie interfaces e implemente-as;
- Use composição para fazer reuso.



Herança múltipla

- Algumas LPs dão suporte à herança múltipla:
  - Resolvem problema de subtipagem múltipla;
  - Não precisam de métodos envoltórios;
- Possíveis problemas:
  - Conflito de nomes;
  - Herança repetida.

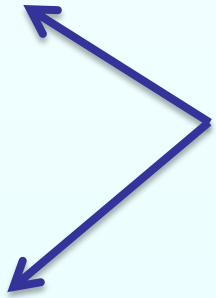
Herança múltipla

- C++, ao contrário de Java, permite a herança múltipla:

```
class Surf {
  time_t data;
  float notas[10];
public:
  void registrarNota(float nota);
};

class Vela {
  time_t data;
  float tempos[10];
public:
  void registrarTempo(float tempo);
};

class WindSurf : public Surf, public Vela {
  char competicao;
};
```



Subclasse herda ambos os métodos

Ambiguidades da herança múltipla

- Colisão de nomes:

```
class Surf {
  time_t data;
  float notas[10];
public:
  virtual void ordena();
};

class Vela {
  time_t data;
  float tempos[10];
public:
  virtual void ordena();
};

class WindSurf : public Surf, public Vela {
  char competicao;
};
```

Não há colisão de atributos se os mesmos forem privativos

Colisão de nomes gerando ambiguidade.

Ambiguidades da herança múltipla

- Programador deve resolver ambiguidade “na mão”:

```
int main () {
  WindSurf ws;
  ws.Surf::ordena();
}
```

- Ou:

```
void WindSurf::ordena() {
  // N = competição por nota; T = por tempo.
  if (competicao == 'N' )
     Surf::ordena();
  else
     Vela::ordena();
}
```

Ambiguidades da herança múltipla

- Herança repetida:

```
class Competicao {
  float premio;
public:
  virtual float getPremio() { return premio; };
};

class Surf: public Competicao { /* ... */ };
class Vela: public Competicao { /* ... */ };

// Herda getPremio() duas vezes:
class WindSurf : public Surf, public Vela { /* ... */ };

int main() {
  WindSurf ws;
  cout << ws.getPremio(); // request is ambiguous
}
```

Ambiguidades da herança múltipla

- Resolve-se usando a palavra-chave `virtual`:

```
class Competicao {
  float premio;
public:
  virtual float getPremio() { return premio; };
};

class Surf: virtual public Competicao { /* ... */ };
class Vela: virtual public Competicao { /* ... */ };

class WindSurf : public Surf, public Vela { /* ... */ };

int main() {
  WindSurf ws;
  cout << ws.getPremio(); // OK!
}
```

Obs.: herança virtual NÃO resolve colisão de nomes!

Metaclasses

- Classes cujas instâncias são outras classes:
  - Atributos são os métodos, instâncias e as superclasses das classes instâncias da metaclasse;
  - Os métodos normalmente oferecem serviços aos programas de aplicação, tais como, retornar os conjuntos de métodos, instâncias e superclasses de uma dada classe;
- Java possui uma única metaclasse, chamada de `Class`.

Metaclasse em Java

```
import java.lang.reflect.*;

class Info {
  public void informa(){}
  public int informa(Info i, int x) { return x; }
}

class MetaInfo {
  public static void main(String args[])
                                        throws Exception {
    Class info = Info.class;
    Method metodos[] = info.getMethods();
    Info i = (Info)info.newInstance();
    Method m = info.getDeclaredMethod("informa", null);
    m.invoke(i, null);
   }
}
```

Metaclasses

- Nível Único: todos objetos são vistos como classes e todas as classes são vistas como objetos;
- Dois Níveis: todos objetos são instâncias de uma classe, mas classes não são acessíveis por programas;
- Três Níveis: todos objetos são instâncias de uma classe e todas as classes são instâncias de uma única metaclasse;
- Vários Níveis: podem existir várias metaclasses.

Qual a forma de Java?

Composição vs. herança

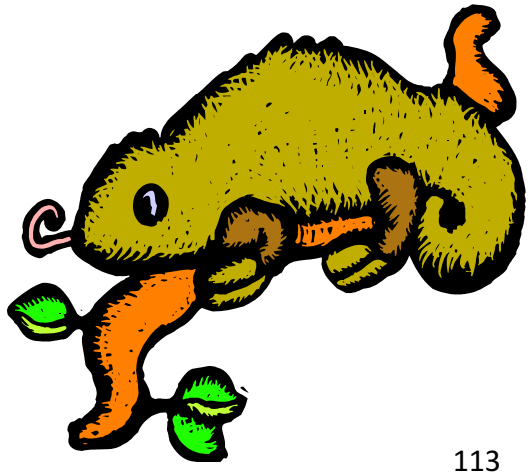
- Composição:
  - Quando se quer as características de uma classe, mas não sua interface;
  - Objeto é utilizado para implementar a funcionalidade da nova classe;
  - Relacionamento do tipo tem-um;
- Herança:
  - Além de usar as características, a classe herdeira também usa a interface da classe herdada;
  - Relacionamento do tipo é-um.

Estruturas de dados genéricas

- Polimorfismo paramétrico (C++, Java 5):
  - Compilador garante homogeneidade dos elementos;
- Polimorfismo por inclusão (Java 1.4):
  - Uso de `Object`;
  - Simplifica a LP;
  - Necessidade de estreitamento.
- Combinados, geram estruturas mais flexíveis e, porém, mais difíceis de entender:
  - Vide generics + curingas + herança em Java!

Conclusões

- Importância do sistema de tipos;
- Sistemas de tipos polimórficos aumentam possibilidade de reutilização;
- Destaque para o polimorfismo de inclusão, fundamental no paradigma OO.



http://nemo.inf.ufes.br/