Capítulo VII – Polimorfismo

"Se você tem uma maçã e eu tenho uma maçã e as trocamos então cada um de nós continuará a ter uma maçã. Mas, se você tem uma idéia e eu tenho outra idéia e as trocamos, então cada um de nós terá duas idéias."

George Bernard Shaw

Tipos de dados definem um conjunto de valores e as operações aplicáveis sobre esses valores. Eles servem fundamentalmente para oferecer informações relevantes aos programadores e aos compiladores (ou interpretadores) sobre os dados usados pelos programas.

Linguagens de máquina são consideradas como não tipadas porque nessas linguagens existe um único tipo, representado pela palavra da memória, o qual é uma cadeia de bits de tamanho fixo. Programar nessas linguagens significa, em última instância, representar tudo (caracteres, números, ponteiros, estruturas de dados, instruções e programas) como cadeias de bits. Assim, a análise de programas escritos em linguagens não tipadas não permite identificar de maneira clara o que está sendo representado. Esse tipo de informação é obtido apenas através de uma interpretação externa ao programa, a qual é normalmente mantida apenas na mente do programador.

É importante atentar que, mesmo em linguagens não tipadas, o conceito de tipos de dados surge naturalmente com a atividade de programação. Tão logo inicie a programar nessas linguagens, o programador começa a organizar os dados (as cadeias de bits) para usá-los de acordo com seus propósitos. Números, caracteres e instruções precisam ser tratados de maneira distinta pelo programador. A categorização das cadeias de bits em termos dos propósitos para os quais são usados corresponde a uma definição informal desses tipos.

O problema com essa abordagem é a impossibilidade de evitar violações na organização dos dados. Por exemplo, não há qualquer impedimento para o programador efetuar a atribuição de uma instrução a algo que anteriormente representava um número. Afinal, ambos são cadeias de bits, o único tipo realmente reconhecido por essas linguagens.

Linguagens de alto nível são tipadas. Cada qual define um sistema de tipos composto por um conjunto próprio de tipos de dados. Sistemas de tipos facilitam o processo de organização dos dados, oferecendo não somente facilidades para a definição e o uso dos dados, mas também para garantir o seu tratamento apropriado. Por exemplo, um sistema de tipos que inclui um tipo booleano facilita o programador a criar variáveis desse

tipo (não é necessário definir a representação de um valor booleano), o uso dessas variáveis (as operações boolenas já estão definidas) e também garantir que não serão usadas em operações inapropriadas (tal como uma adição com um valor inteiro).

O número e as propriedades dos tipos de dados, as facilidades e as garantias oferecidas variam de LP para LP, de acordo com a sofisticação do sistema de tipos. Uma decisão importante diz respeito ao grau de polimorfismo do sistema. Polimorfismo em LPs se refere a possibilidade de se criar código capaz de operar sobre valores de tipos diferentes. Em geral, quanto mais polimórfico é um sistema de tipos, maior é a possibilidade se criar código reutilizável em uma LP.

Esse capítulo tem por objetivo descrever os diferentes tipos de polimorfismo oferecidos por LPs. O capítulo começa apresentando uma breve introdução ao estudo dos sistemas de tipos. Em seguida, são apresentados e discutidos os diferentes tipos de polimorfismo. Especial atenção é dada ao tipo de polimorfismo que é intrisicamente associado a orientação a objetos.

7.1 Sistemas de Tipos

O uso de um sistema de tipos serve para descrever de modo mais adequado os dados, aumentando a legibilidade e a redigibilidade de programas. Além disso, a disciplina relacionada com o uso de tipos evita que programas executem operações incoerentes, tal como, somar um inteiro e um caracter. Neste sentido, LPs de alto nível diferenciam-se grandemente de LPs de baixo nível onde o único tipo é “byte”ou “word”.

Um conceito importante nesse contexto é o de verificação de tipos. Para se ter uma definição bem abrangente desse conceito vale lembrar que sempre se pode considerar subprogramas como operadores e seus parâmetros como operandos. Verificação de tipos é, portanto, a atividade de garantir que operandos de um certo operador são de tipos compatíveis. Um tipo compatível é um tipo cujos valores sejam adequados para a realização da operação designada pelo operador ou que pode ser convertido implicitamente em um tipo cujos valores sejam adequados.

Verificação a priori de tipos é normalmente recomendada pois evita a realização de operações sem sentido. Por exemplo, a verificação a priori de tipos deve analisar se os operandos de uma expressão aritmética são realmente de um tipo numérico.

7.1.1 Verificação de Tipos

LPs podem possibilitar ou não a realização de uma ampla verificação de tipos. Algumas LPs, tal como C, adotam uma postura fraca com relação a isso pois somente parte dos erros de tipos são verificados. Por exemplo, em C, é possível fazer um ponteiro para um *float* acessar uma área de memória onde está alocado um vetor de caracteres. Embora ofereça flexibilidade, esse tipo de característica, se usada inapropriadamente, pode gerar problemas desastrosos. Outras LPs, tais como ADA e JAVA, procuram realizar uma verificação extremamente ampla de tipos. A vantagem de se fazer isso é aumentar a produtividade do programador e a confiabilidade dos programas.

A verificação de tipos deve ser necessariamente realizada antes da execução das operações. De modo geral, elas devem ser feitas o quanto antes possível. Isso garante que os erros de tipos serão identificados mais cedo, trazendo significativos benefícios para todo o processo de desenvolvimento de programas. Contudo, nem sempre isso é possível ou desejável. Algumas LPs possuem brechas no seu sistema de tipos que impedem a realização de algum tipo de verificação. Além disso, em certas situações, pode ser conveniente retardar a verificação de tipos. Por exemplo, programas em LPs orientadas a objetos podem precisar converter tipos de objetos durante a execução. Nesses casos, a verificação dessa operação só pode ser realizada tardiamente.

LPs estaticamente tipadas executam a verificação de tipos em tempo de compilação. Todos os parâmetros e variáveis devem possuir um tipo fixo identificável a partir da análise do texto do programa. Desse modo, o tipo de cada expressão pode ser identificado e cada operação pode ser verificada pelo compilador. Em certas situações, LPs estaticamente tipadas podem ser excessivamente restritivas, reduzindo a redigibilidade dos programas. Exemplo de LP quase estaticamente tipada é MODULA-2, reconhecida pelas restrições que impõe à escrita dos programas. Cabe dizer que, embora a verificação de tipos em MODULA-2 seja toda feita em tempo de compilação, ela não pode ser considerada uma LP estaticamente tipada por causa do conceito de registro variante, o qual abre uma brecha para a ocorrência de violações do sistemas de tipos em tempo de execução[7.1].

LPs dinamicamente tipadas só executam a verificação de tipos em tempo de execução. Somente os valores dessas LPs têm tipo fixo. Normalmente, cada valor tem associado a ele um `tag` indicando o seu tipo. Uma variável ou parâmetro não possui um tipo associado, podendo designar valores

[7.1] Ver seção 3.2.2.2 desse livro.

de diferentes tipos em pontos distintos da execução. Esse fato impõe a realização da verificação dos tipos de operandos imediatamente antes da execução da operação. LISP, BASIC, APL e SMALLTALK são LPs dinamicamente tipadas.

Algumas LPs realizam a maior parte das verificações de tipos em tempo de compilação, mas deixam algumas verificações para serem feitas em tempo de execução. Exemplos de LPs que possuem essa característica são C++, ADA e JAVA.

Linguagens fortemente tipadas [CARDELLI, 1991] devem possibilitar a detecção de todo e qualquer erro de tipo em seus programas. Cardelli recomenda usar a verificação estática tanto quanto o possível e a verificação dinâmica quando for necessário nessas linguagens. ALGOL-68 é uma linguagem fortemente tipada. ADA e JAVA são aproximadamente fortemente tipadas.

É importante entender como verificações estáticas de tipos são efetuadas pelo compilador. Observe o exemplo 7.1 em C.

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

Exemplo 7. 1 - Verificação Estática de Tipos

O compilador de C efetua as seguintes verificações de tipo no código do exemplo 7.1:

1. Os operandos de % tem de ser inteiros, logo *n* também tem de ser inteiro. Como *n* e *2* são inteiros, a operação % é válida.
2. Os operandos de == devem ser de um mesmo tipo. Como o operando *0* e o resultado de % são inteiros, a operação também é válida.
3. O tipo de retorno de *par* deve ser inteiro. Como o resultado de == é inteiro, ele é compatível com o tipo retornado pela função.
4. O parâmetro formal de par é inteiro. Como o parâmetro real usado na chamada da função *par* é inteiro, ele é compatível com o tipo do parâmetro formal correspondente.

Em linguagens dinamicamente tipadas, as verificações de tipo são efetuadas imediatamente antes da execução das operações. O exemplo 7.2 em LISP mostra uma função que efetua a multiplicação de um número por 3.

```
(defun mult-tres (n)
```

```
  (* 3 n))

(mult-tres 7)
(mult-tres "abacaxi")
```

Exemplo 7. 2 - Verificação Dinâmica de Tipos em LISP

O exemplo 7.2 mostra inicialmente a função *mult-tres*, que tem um parâmetro formal *n*. Como o tipo de *n* não é conhecido previamente, essa função pode ser chamada com argumentos de tipos diferentes. Para cada chamada da função *mult-tres* é necessário verificar o tipo de *n* em tempo de execução para assegurar que seja um número (único tipo válido como argumento da operação *). Assim, na primeira chamada a *mult-tres*, o parâmetro *n* é associado ao valor *7* e a função retorna o resultado de sua multiplicação por três. Note que somente quando a função tentar efetuar a multiplicação é que será feita a verificação do tipo de *n*. Na segunda chamada, ocorrerá um erro de execução, uma vez que o tipo de *n* não é um valor legal para a operação de multiplicação.

LPs dinamicamente tipadas oferecem maior flexibilidade para produzir código reutilizável. O exemplo 7.3 mostra a função *segundo* em LISP. Essa função é usada para retornar o segundo elemento de uma lista. O corpo de *segundo* utiliza as funções *car* (retorna o primeiro elemento da lista) e *cdr* (retorna a lista sem o primeiro elemento) de LISP para atingir o seu objetivo.

```
(defun segundo (l)
  (car(cdr l)))

(segundo (1 2 3))
(segundo ((1 2 3) (4 5 6)))
(segundo ("manga" "abacaxi" 5 6))
```

Exemplo 7. 3 - Vantagem de Tipagem Dinâmica

Note as três chamadas de *segundo* no exemplo 7.3. O fato de *l* não ser previamente amarrada a um tipo específico de lista, possibilita que, em cada chamada, a lista *l* assuma valores de tipos distintos. Na primeira chamada, *l* é uma lista de números, na segunda, uma lista de listas e, na terceira, uma lista de `strings` e números.

Por outro lado, LPs dinamicamente tipadas perdem eficiência computacional devido à inclusão da verificação de tipos em tempo de execução e ao aumento de espaço de memória ocupada, uma vez que é necessário manter o tipo dos valores durante a execução do programa. Além disso, é necessário manter em memória o código necessário para a verificação dos tipos em tempo de execução e para tratar qualquer tipo legal que possa ser utilizado pelas funções.

LPs estaticamente tipadas evitam os problemas de eficiência e espaço citados no parágrafo anterior e proporcionam confiabilidade. Erros de tipos são detectados em tempo de compilação, ao invés de terem de serem detectados por testes (procedimento pouco confiável). Por outro lado, elas impõem mais restrições à redigibilidade e à reutilização de código.

LPs fortemente tipadas que seguem a recomendação de Cardelli, citada anteriormente nessa seção, oferecem a melhor combinação de eficiência e flexibilidade, além de aumentarem a confiabilidade, uma vez que erros de tipo serão sempre detectados.

7.1.2 Inferência de Tipos

LPs estaticamente tipadas não devem necessariamente exigir a declaração explícita de tipos. Um sistema de inferência de tipos pode ser usado para identificar os tipos de cada entidade e expressão do programa. Para ter clareza sobre essa propriedade, suponha que na função *par* do exemplo 7.1 o tipo do parâmetro *n* não tivesse sido declarado. Ainda assim, um compilador poderia deduzir, pelo fato da operação % requerer operandos do tipo inteiro, que o parâmetro *n* tem de ser inteiro.

Exemplos de LPs cujos compiladores realizam inferência de tipos são HASKELL e ML. Cabe ressaltar que esses compiladores são bem mais exigidos que um compilador de uma LP com declaração explícita de tipos. Cabe dizer ainda que ML é liberal quanto a declaração de tipos, isto é, o programador pode definir explicitamente o tipo da entidade declarada ou deixar o compilador inferir o tipo.

Não declarar explicitamente o tipo das entidades certamente aumenta a redigibilidade dos programas. Contudo, isso também provoca uma redução de legibilidade pois pode ser difícil descobrir o tipo de uma função ou entidade. Muitas vezes é necessário percorrer várias outras funções para poder realizar a identificação. Outra desvantagem ocorre quando surge um erro de programação. Isso pode fazer com que o compilador produza mensagens de erro obscuras.

7.1.3 Equivalência de Tipos

LPs podem adotar posturas distintas em situações nas quais se aplica um operando de tipo diferente do esperado por uma operação. Algumas LPs, como C, são extremamente liberais permitindo que isso seja feito livremente. Nessas LPs o operando é convertido implicitamente e tratado como se fosse do tipo esperado pela operação. Outras LPs, como MODULA-2, são extremamente rigorosas e, por isso, não admitem isso ser feito.

Assim, o programador é forçado a converter o valor explicitamente para o tipo esperado da operação ou não realizá-la.

Algumas LPs, tal como PASCAL, adotam uma postura intermediária, fazendo conversões implícitas sempre que possível e acusando erro quando isso não é válido. LPs estabelecem regras definindo quais conversões implícitas são válidas. Essas regras podem ser específicas para cada tipo de conversão, baseadas no conceito de inclusão de tipos ou no conceito de equivalência de tipos.

Segundo o conceito de inclusão de tipos, a conversão é permitida se os valores do tipo do operando também são valores do tipo esperado pela operação. Por exemplo, a conversão implícita de um valor do tipo *int* para um *long* em JAVA é permitida porque os valores de *int* também são valores de *long*.

Regras também podem ser baseadas na equivalência estrutural ou nominal de tipos. No caso da equivalência estrutural, a conversão é permitida se o conjunto de valores do tipo do operando é o mesmo do tipo esperado pela operação. Essa equivalência é chamada de estrutural porque a verificação da igualdade do conjunto de valores é realizada comparando-se as estruturas dos dois tipos, uma vez que a comparação por enumeração de valores é inviável. Watt [WATT, 1990] lista as seguintes regras para definir a equivalência estrutural de dois tipos T e T'.

Se T e T' são primitivos, então T e T' devem ser idênticos
    Por exemplo, 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'
    Por exemplo, 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'
    Por exemplo, inteiro + booleano ≡ booelano + inteiro
Se T e T'são mapeamentos e T = A → B e T' = A' → B';
    então A ≡ A' e B ≡ B'.
    Por exemplo, inteiro → booleano ≡ inteiro → booleano

Figura 7. 1 - Equivalência Estrutural de Tipos Segundo Watt [WATT, 1990]

No caso da equivalência nominal, dois tipos só são equivalentes se e somente se possuem o mesmo nome. Essa é uma abordagem mais restritiva pois sua aplicação implica na inexistência de possibilidade de conversão implícita de tipos por equivalência. De fato, em LPs que adotam esse modo de equivalência, a única forma de fazer conversões é através de regras específicas ou baseadas no conceito de inclusão. Com o advento da pro-

gramação baseada em TADs, a equivalência nominal passou a ser adotada pela maioria das LPs pois ela sempre garante a consistência entre os operandos e as operações.

O exemplo 7.3, escrito em C, pode ser usado para ilustrar as diferenças entre a equivalência estrutural e a nominal. No exemplo são definidos dois tipos a partir do tipo *float*.

```
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
}
```

Exemplo 7. 4 - Equivalência de Tipos

Em caso de equivalência estrutural, todo o código do exemplo é válido. No caso de equivalência nominal, o retorno da função *converte*, a atribuição e a chamada da última linha do código são inválidas. C adota equivalência estrutural nesse caso. No entanto, se em vez de tipos primitivos, *quilometros* e *milhas* fossem associados a estruturas anônimas idênticas, a equivalência seria nominal.

7.1.4 Sistemas de Tipos Monomórficos e Polimórficos

PASCAL e MODULA-2 têm um sistema de tipos no qual todas constantes, variáveis e subprogramas devem ser definidos com um tipo específico. Um sistema de tipos como esse é chamado de monomórfico. Entretanto, a simplicidade imposta por um sistema de tipos monomórfico pode ser insatisfatória quando se deseja escrever códigos reutilizáveis. Muitos algoritmos e estruturas de dados são inerentemente genéricos, no sentido de que eles são praticamente independentes do tipo dos valores manipulados. Contudo, uma LP monomórfica não permite defini-los e tratá-los genericamente.

Considere, por exemplo, as operações de pertinência, contingência, união e interseção de conjuntos. Conceitualmente, essas operações são genéricas pois podem ser realizadas independentemente dos tipos dos elementos dos conjuntos. De fato, nada impede aos conjuntos de possuírem até elementos de tipos distintos e ainda assim se poder realizar essas operações sobre eles.

Contudo, linguagens monomórficas não proporcionam flexibilidade para se fazer nada disso. Nessas LPs é necessário criar um tipo conjunto para cada tipo de elemento e operações específicas para cada um dos tipos conjunto. Isso provoca grande redundância no código, reduzindo significativamente a redigibilidade, pois as operações são efetivamente as mesmas, somente variando o tipo do elemento dos conjuntos. Além disso, linguagens monomórficas não possibilitam que os conjuntos sejam compostos por elementos de tipos diferentes.

Embora basicamente monomórficas, PASCAL E MODULA-2 não são puramente monomórficas. PASCAL possui os subprogramas pré-definidos *read, readln, write, writeln* e *eof*, os quais aceitam valores de vários tipos na sua chamada. MODULA-2 e PASCAL possuem operadores (por exemplo, o operador +) os quais atuam sobre diversos tipos.

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

C, por exemplo, possui o tipo *void**, o qual possibilita a criação de variáveis e parâmetros cujos valores podem ser ponteiros para tipos quaisquer. Esse tipo de C possibilita a construção de estruturas de dados genéricas e algoritmos genéricos para manipulá-las. C também permite a criação de funções polimórficas através do uso de uma lista de parâmetros variável. Outros exemplos de linguagens polimórficas são ADA, ML, C++ e JAVA.

7.2 Tipos de Polimorfismo

Polimorfismo em LPs se refere a possibilidade de se criar código capaz de operar (ou, pelo menos, aparentar operar) sobre valores de tipos distintos. Existem diferentes tipos de polimorfismo. Uma classificação [CARDELLI & WEGNER, 1985] dos tipos de polimorfismo é apresentada na tabela7.1.

Polimorfismo	Ad-hoc	Coerção
		Sobrecarga
	Universal	Paramétrico
		Inclusão

Tabela 7. 1 - Tipos de Polimorfismo [CARDELLLI & WEGNER, 1985]

A diferença entre polimorfismo ad-hoc e universal diz respeito ao uso e à capacidade de proporcionar reuso de código. O polimorfismo ad-hoc se

aplica apenas a abstrações de controle e somente proporciona reuso do código no qual é feita a chamada da abstração de controle. Já o polimorfismo universal se aplica tanto às abstrações de dados quanto às de controle. Além disso, ele proporciona reuso de código tanto na chamada quanto na implementação das abstrações de controle.

O polimorfismo ad-hoc aparentemente proporciona reuso do código de implementação da abstração, mas na realidade não o faz. Tipicamente, são criadas abstrações de controle específicas para operar sobre cada tipo admissível.

De fato, o polimorfismo ad-hoc ocorre quando um mesmo símbolo ou identificador é associado a diferentes trechos de código que atuam sobre diferentes tipos. Para quem lê o código, pode parecer que o símbolo ou identificador denota um único trecho de código polimórfico, atuando sobre elementos de tipos diferentes. Contudo, isso é apenas aparente, uma vez que, através dos argumentos associados ao identificador ou símbolo, se faz a identificação do trecho de código específico a ser executado.

O polimorfismo universal ocorre quando uma estrutura de dados pode ser criada incorporando elementos de tipos diversos ou quando um mesmo código pode ser executado e atuar sobre elementos de diferentes tipos.

Por isso, o polimorfismo universal é tido como o "verdadeiro" polimorfismo enquanto que o polimorfismo ad-doc é tido como um polimorfismo "aparente".

A tabela 7.1 também mostra que o polimorfismo ad-hoc pode ser dividido em polimorfismo de coerção e polimorfismo de sobrecarga. Por sua vez, o polimorfismo universal se subdivide em polimorfismo paramétrico e polimorfismo por inclusão. Cada um desses tipos de polimorfismos serão detalhados nas próximas seções desse capítulo.

7.2.1 Coerção

Coerção significa conversão implícita de tipos. Por exemplo, em C, ao se atribuir um valor de um tipo *char* para uma variável do tipo *int*, haverá uma conversão implícita do valor do tipo *char* para um valor do tipo *int*.

Para fazer isso, o compilador de C necessita ter uma tabela embutida de conversões permitidas. Assim, quando uma operação é aplicada sobre um operando de tipo diferente do esperado, o compilador verifica se existe uma conversão na tabela que seja aplicável e, em caso positivo, inclui código para realizar a conversão do tipo do operando para o tipo esperado pela operação. O exemplo 7.5 mostra como isso é feito em um programa C.

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

Exemplo 7. 5 - Coerção em C

No exemplo 7.5, apesar de a função receber um *float* como parâmetro, ela é chamada passando um *long*. Aparentemente, a função *f* seria capaz de tratar tanto valores do tipo *float* quanto do tipo *long*. Contudo, essa função só lida efetivamente com valores do tipo *float*. De fato, o próprio compilador se encarrega de embutir código para fazer a conversão.

Frequentemente, coerções são ampliações ou estreitamentos. A ampliação ocorre quando um valor de um tipo com menor conjunto de valores é convertido em um valor de um tipo cujo conjunto de valores é mais amplo. No estreitamento ocorre o inverso, o que implica no risco de perda de informação, uma vez que o valor do tipo mais amplo pode não possuir representação no tipo mais estreito.

Em alguns casos, coerções podem não ser ampliações nem estreitamentos. Por exemplo, quando, em C, se converte um *int* em um *unsigned*, embora os dois tipos possuam o mesmo número de valores, os seus conjuntos de valores são diferenciados. Nesses casos, embora não se possa dizer que houve ampliação ou estreitamento, também existe a possibilidade de perda de informação.

Uma LP frequentemente fornece um conjunto de regras para definir as conversões implícitas de tipos válidas. Por exemplo, algumas das regras de C são:

1. Valores de tipo mais estreito podem ser promovidos para tipos mais amplos. Por exemplo: *char* para *int*, *int* para *float*, *float* para *double*.
2. Quando um valor do tipo ponto flutuante é convertido para um tipo inteiro, a parte fracionária é eliminada; se o valor resultante não puder ser representado no tipo inteiro, o comportamento é definido pela implementação do compilador.

Coerções dão a entender que determinada operação ou subprograma pode ser realizada com operandos de tipos diferentes. Contudo, não é isso o que ocorre. Veja o exemplo 7.6 em C.

```
main() {
  int w = 3;
  float x;
  float y = 5.2;
  x = x + y;          // x = somafloat (x, y)
```

```
  x = x + w;        // x = somafloat (x, intToFloat (w) );
}
```

Exemplo 7. 6 - Expressão com Coerção em C

Embora a coerção na última linha do exemplo 7.6 possa dar a impressão, para quem cria ou lê esse programa, que o operador + corresponde a uma função capaz de realizar tanto a operação de somar dois valores do tipo *float* (*float* x *float* $\rightarrow$ *float*) quanto a operação de somar um *float* e um *int* (*float* x *int* $\rightarrow$ *float*), ela só realiza mesmo a primeira dessas operações. Como pode ser observado no exemplo, antes da chamada da operação de soma, no segundo uso do operador +, ocorre uma chamada implícita a uma função capaz de converter valores do tipo *int* em valores do tipo *float*.

Em C, coerções ocorrem com grande frequência em expressões e atribuições. O exemplo 7.7 mostra várias coerções ocorrendo em atribuições.

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

Exemplo 7. 7 - Coerções em Atribuições em C

Na primeira atribuição do exemplo 7.7 o valor *char* de *c* é convertido em um *int*. Na segunda atribuição, o valor *int* de *i* é convertido em um *char*. Nesse caso, pode haver perda de informação pois os bits mais significativos de *i* são descartados. Na penúltima atribuição, o valor *int* de *i* é convertido em um *float*. Por sua vez, na última atribuição, o valor *float* de *x* é convertido em um *int*, através do truncamento do valor de *x.* Novamente nesse caso pode haver perda de informação pois a parte fracionária é descartada e o valor inteiro de *x* pode não possuir correspondente no intervalo definido por *int*.

Existem posições controversas a respeito da inclusão de coerções em LPs. Enquanto elas tornam a LP mais redigível (uma vez que não é necessário chamar explicitamente as funções de conversão), elas podem impedir a detecção de certos tipos de erros por parte do compilador, o que pode reduzir a confiabilidade dos programas criados na linguagem. Considere o exemplo 7.8 em C.

```
main() {
```

```
  int a, b = 2, c = 3;
  float d;
  d = a * b;        // d = (float) (a * b) ;
  a = b * d;        // a = b * c;
}
```

Exemplo 7. 8 - Diferentes Perspectivas sobre Coerções

No exemplo 7.8, caso não houvesse coerção de *int* para *float*, na primeira atribuição, o programador teria de converter explicitamente (através do uso do operador de `cast`) o resultado da multiplicação para *float*. Isso reduziria a redigibilidade da linguagem e tornaria a programação enfadonha. Por outro lado, se na segunda atribuição o programador desejasse multiplicar *b* e *c*, mas tivesse digitado equivocadamente *d*, o compilador C não poderia fornecer qualquer auxílio, uma vez que a conversão implícita de tipos é legal.

Por causa dos problemas de confiabilidade ADA e MODULA-2 não admitem coerções, sacrificando assim a redigibilidade. C adota uma postura bastante permissiva em relação a coerções. JAVA busca um meio termo só admitindo a realização de coerções para tipos mais amplos. O exemplo 7.9 ilustra a postura adotada por JAVA.

```
byte a, b = 10, c = 10;
int d;
d = b;
c = (byte) d;
a = (byte) (b + c);
```

Exemplo 7. 9 – Coerção e Conversão Explícita em JAVA

No exemplo 7.9 a coerção do valor *byte* de *b* para o valor *int* de *d* na primeira atribuição é válida porque ocorre uma ampliação. As conversões explícitas nas demais atribuições são obrigatórias pois há estreitamento. Observe, na última linha do exemplo, a existência de coerções dos valores de *b* e *c* para valores *int* antes da realização da operação de soma. O resultado dessa operação será do tipo *int* e, portanto, deve ser convertido explicitamente em um *byte*.

7.2.2 Sobrecarga

Um identificador ou operador é sobrecarregado quando pode ser usado para designar duas ou mais operações distintas. Em geral, sobrecarga (ou `overloading`, como é mais conhecida) somente é aceitável quando o uso do operador ou identificador não é ambíguo, isto é, quando a operação apropriada pode ser identificada usando-se apenas as informações disponíveis a respeito dos tipos dos operandos. Veja, por exemplo, o caso

do operador – de C. Ele pode ser usado para denotar um número significativamente elevado de operações, tais como:

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*)

O conjunto de operações associadas ao operador – pode ser muito maior que o listado acima, uma vez que se pode associar a ele, para cada tipo distinto de dados numérico, uma operação de negação e uma de subtração.

Tal como a coerção, a sobrecarga também sugere que determinada operação ou subprograma pode ser realizada com operandos de tipos diferentes. Contudo, não é isso que ocorre. O exemplo 7.10 em C ilustra esse fato.

```
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);
}
```

Exemplo 7. 10 - Sobrecarga de Operador + em C

Embora a sobrecarga do operador + no exemplo 7.10 possa dar a impressão, para quem cria ou lê esse trecho de programa, que esse operador representa uma função capaz de realizar tanto a operação de somar dois valores do tipo *int* (*int* x *int* → *int*), quanto a operação de somar dois valores do tipo *float* (*float* x *float* → *float*), na realidade, cada ocorrência de + invoca funções específicas para cada uma dessas operações.

MODULA-2 e C embutem sobrecarga em seus operadores, mas os programadores só podem usar essa sobrecarga, sem poder implementar novas sobrecargas. Além disso, não existe qualquer sobrecarga de subprogramas. PASCAL adota postura similar. No entanto, também existem subprogramas sobrecarregados na biblioteca padrão, tais como *read* e *write*. JAVA embute sobrecarga em operadores e em subprogramas de suas bibliotecas, mas somente subprogramas podem ser sobrecarregados pelo programador. ADA e C++ adotam a postura mais ampla e ortogonal realizando e permitindo que programadores realizem sobrecarga de subprogramas e operadores.

ADA e C++ permitem aos programadores sobrecarregar os operadores da linguagem, mas não admitem a criação de novos operadores, isto é, operadores diferentes dos existentes na LP. Além disso, quando é feita a so-

brecarga dos operadores, sua sintaxe e precedência não pode ser alterada. O motivo para essas restrições é evitar a criação de ambigüidades.

Nem todos operadores de C++ podem ser sobrecarregados. Por exemplo, os operadores *::* (resolução de escopo), . (seleção de membro), *sizeof* (tamanho do objeto/tipo) não podem ser sobrecarregados. Os motivos para não permitir a sobrecarga desses operadores são evitar a quebra de mecanismos de segurança, o surgimento de confusões e dificuldades no entendimento do código. O exemplo 7.11 ilustra a sobrecarga de subprogramas e operadores 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;
  }
  const umValor& operator++() { // prefixado
    v++;
    return *this;
  }
  const umValor operator++(int) { // posfixado
    umValor antes(v);
    v++;
    return antes;
  }
};
main() {
  int a = 1, b = 2, c = 3;
  c += a +b;
  umValor r(1), s(2), t;
  t += r + s;
  r = ++s;
  s = t++;
}
```

Exemplo 7. 11 - Sobrecarga em C++ (adaptado de [Eckel, 2000])

O exemplo 7.11 mostra a sobrecarga da função construtora *umValor* e dos operadores +, += e ++ na classe *umValor*. Sobrecarga de subprogramas

é muito útil em linguagens orientadas a objeto para permitir a criação de objetos segundo diferentes contextos. Observe, na terceira linha da função *main* do exemplo, que *t* é criada com o construtor `default` de *umValor* e que *r* e *s* são criados usando a outra função *umValor*, a qual é parametrizada.

O exemplo 7.11 também mostra a sintaxe utilizada por C++ para o programador sobrecarregar operadores. Ela envolve o uso da palavra *operator@*, onde @ é o operador a ser sobrecarregado. Note como esses operadores são usados em *main* da mesma maneira que seus equivalentes já pré-definidos. Note também que as funções de sobrecarga dos operadores binários só requerem um parâmetro, uma vez que o outro é o próprio objeto, o qual será o argumento da esquerda nas expressões que aplicam o operador.

O exemplo 7.11 mostra ainda a notação sintática adotada em C++ para permitir a sobrecarga dos operadores unários ++ e --. Como esses operadores podem ser usados de maneira prefixada ou posfixada, é necessário haver um meio de definir qual modo a ser usado. C++ usa uma notação sintática no mínimo esquisita para resolver isso. Se a lista de parâmetros da função de sobrecarga for vazia, adota-se a forma prefixada. Já se a lista conter um parâmetro fictício do tipo *int*, adota-se a forma posfixada. Esse é claramente um exemplo de solução contra-intuitiva uma vez que não fica claro porque o operador posfixado tem um parâmetro fictício e o prefixado não.

C++ e JAVA usam sobrecarga independente do contexto. Em outras palavras, o tipo de retorno não pode ser usado para diferenciar uma função de outra. Assim, todo e qualquer operador, ou identificador de subprograma, sobrecarregado deve possuir uma lista de parâmetros diferenciada em número ou tipo dos parâmetros.

O exemplo 7.12, em C++, ajuda entender a sobrecarga independente de contexto. Todas as funções *f* definidas no exemplo 7.12 possuem listas de parâmetros distintas, exceto aquela colocada na linha comentada. Caso não fosse assim, essa linha causaria ambiguidade na primeira chamada de função em *main*, impedindo o compilador de determinar se a função a ser chamada é a que retorna *void* ou a que retorna *int*. A sobrecarga independente de contexto é adotada exatamente para evitar esses problemas. Assim, os compiladores sempre reportam um erro quando o programador cria funções sobrecarregadas cuja diferenciação é feita pelo tipo de retorno.

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

```
void f(float, float) { }
// int f(void) { }
main() {
  f();
  f(2.3);
  f(4, 5);
  f(2.2f, 7.3f);
  // f(3, 5.1f);
  // f(1l, 2l);
}
```

Exemplo 7. 12 - Sobrecarga Independente de Contexto em C++

No exemplo 7.12 as funções chamadas são aquelas cuja lista de parâmetros corresponde exatamente ao número e tipo dos parâmetros passados na chamada, com exceção da segunda chamada e das duas últimas colocadas em linhas comentadas. No caso da segunda chamada de *f*, o argumento passado é um valor do tipo *double*. Como não existe qualquer declaração de *f* cuja lista de parâmetros seja composta por um valor desse tipo, o compilador de C++ verifica se é possível realizar uma coerção desse valor para um tipo de modo a fazer um casamento com alguma definição de *f*. Dessa maneira, a função *f(float)* é chamada. Por outro lado, esse mesmo processo não é bem sucedido nas duas chamadas feitas em linhas comentadas, pois há ambigüidade causada pela combinação de sobrecarga com coerção. Em ambas o compilador não consegue determinar se deve chamar a função *f(int, int)* ou *f(float, float)*. Observe ainda que esse problema também ocorreria caso o especificador *f* de literal *float* fosse omitido na chamada *f(2.2f, 7.3f)*.

ADA adota sobrecarga dependente de contexto. Isso significa que cada definição de função sobrecarregada deve possuir uma assinatura diferenciada. Tal exigência se contrasta com o outro tipo de sobrecarga, o qual requer uma lista de parâmetros distinta. Assim, na sobrecarga dependente de contexto, as funções podem ter o mesmo número e tipos de parâmetros e se distingüirem apenas pelo tipo do valor retornado pela função. A sobrecarga dependente de contexto exige mais esforço dos compiladores e pode provocar erros de ambigüidade mesmo em LPs nas quais não há coerção.

Existe controvérsia quanto a possibilidade dos programadores sobrecarregarem os operadores de uma LP. Nem todos consideram uma característica desejável. Enquanto alguns consideram que programas podem ficar mais fáceis de serem lidos e redigidos com a sobrecarga, outros consideram que, além de aumentar a complexidade da LP, esse recurso frequentemente é mal utilizado e acaba levando a elaboração de programas mais difíceis de serem lidos e entendidos. Os criadores de JAVA resolveram

não incluir a sobrecarga de operadores por considerá-la capaz de gerar confusões e aumentar a complexidade da LP.

7.2.3 Paramétrico

Nesse tipo de polimorfismo, pode-se construir abstrações de dados e controle que atuam uniformemente sobre valores de vários tipos. A principal característica desse polimorfismo é a parametrização das estruturas de dados e subprogramas com relação ao tipo do elemento sobre o qual operam. Pode-se dizer, então, que essas abstrações recebem um parâmetro implícito adicional especificando o tipo sobre o qual elas agem. O exemplo 7.13 mostra a implementação de uma função em C que aceita um inteiro e simplesmente retorna ele mesmo.

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

Exemplo 7. 13 - Função Específica para Tipo Inteiro

A chamada *identidade (12)* produziria *12*. Já a chamada *identidade (7.5)* retornaria *7*, um valor incorreto, visto que C converteria implicitamente o argumento em um inteiro. Em outras LPs menos permissivas com relação a coerções (MODULA-2, por exemplo) essa chamada seria ilegal, gerando erro de compilação, porque o argumento não é do tipo inteiro.

Contudo, essa postura não parece ser muito adequada. Semanticamente, a função *identidade* pode ser aplicada a valores de qualquer tipo. De fato, não existe qualquer operação específica de inteiros no corpo dessa função. Seria mais razoável, então, poder escrever a função *identidade* tal como no exemplo 7.14.

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

Exemplo 7. 14 - Função Genérica

Essa função é do tipo $T \rightarrow T$ no qual T é uma variável tipo pois pode designar um tipo qualquer. Agora a chamada *identidade (7.5)* produziria o valor esperado. Mais que isso, em uma LP na qual existisse o tipo *boolean*, por exemplo, a chamada *identidade(true)* também seria válida e legal.

Chamadas da função *identidade* podem assumir tipos diferentes. Esse tipo pode ser determinado combinando o tipo dos valores dos argumentos com os tipos declarados no cabeçalho da função. Por exemplo, na chamada *identidade(true)* combina-se o tipo do argumento *boolean*, com o tipo da função $T \rightarrow T$, substituindo sistematicamente o T pelo *boolean*. Portanto,

o tipo do resultado será *boolean* (e seu valor será *true*) e o tipo dessa chamada de *identidade* é *boolean* → *boolean*.

Um tipo como $T \rightarrow T$ é chamado de politipo porque a partir dele se pode derivar uma família de muitos tipos. Não existe qualquer impedimento em se usar mais de uma variável tipo em um politipo. Por exemplo, o politipo $U \times T \rightarrow T$ possui duas variáveis tipo e pode ser associado com funções de diversas assinaturas, tais como, $int \times float \rightarrow float$, $float \times int \rightarrow int$ e $int \times int \rightarrow int$.

C++ usa o mecanismo de *template* para incorporar o polimorfismo paramétrico. Observe como a função *identidade* pode ser implementada em C++ no exemplo 7.15.

```
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);
  // y = identidade (d2);
}
```

Exemplo 7. 15 - Função Genérica em C++

A função *identidade* foi usada na função *main* para denotar funções do tipo (*int* → *int*), (*float* → *float*) e (*data* → *data*). A chamada de *identidade* na linha comentada provocaria erro de compilação. Isso ocorreria porque ela retorna um valor do tipo *data*, mas a atribuição demanda um *float*.

Nem todas funções baseadas em polimorfismo paramétrico são tão genéricas quanto a função *identidade*. Muitas vezes, os tipos aplicáveis a essas funções são limitados por operações realizadas no corpo da função. Nesses casos, o programador deve levar isso em conta durante o uso da função para evitar a ocorrência de erros. O exemplo 7.16 mostra a função *maior*, em C++, a qual usa o mecanismo *template* para retornar o maior entre dois valores de um tipo. Para poder ser usada, o tipo do valor dos argumentos deve ter uma operação associada ao operador >, uma vez que *maior* vai usá-la para identificar o maior dos dois valores.

```
template <class T>
T maior (T x, T 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);
}
```

Exemplo 7. 16 – Função Genérica Restrita a Certos Tipos em C++

As duas primeiras chamadas de *maior* no exemplo 7.16 são válidas porque o operador > é pré-definido para valores dos tipos *int* e *float*. Contudo, a chamada na linha comentada provocaria erro de compilação porque o operador > não está definido para a classe *tData*. Esse erro não ocorreria caso existisse uma sobrecarga do operador > para a classe *tData*.

Além de subprogramas com polimorfismo paramétrico, é também possível criar estruturas de dados parametrizadas em relação ao tipo dos seus elementos. Para criar essas estruturas é fundamental o conceito de tipo parametrizado, isto é, um tipo cuja definição é parametrizada em relação a outros tipos. Tipos parametrizados frequentemente existem em LPs cujo sistema de tipos inclui tipos compostos. Com ele se pode especificar propriedades de arquivos, vetores e conjuntos sem se preocupar com os tipos reais de seus componentes. Os tipos *file*, *array* e *set* de PASCAL e *array* de ADA são parametrizados porque é possível criar a partir deles tipos cujos elementos são inteiros, reais, caracteres, etc.

Contudo, em uma linguagem basicamente monomórfica, somente existem tipos parametrizados pré-definidos. O programador não pode definir novos tipos parametrizados. C++ também usa o mecanismo *template* para permitir a criação de tipos parametrizados pelos programadores. O exemplo 7.17 mostra a definição e uso do tipo parametrizado *tPilha*.

```
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);
};
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];
}
class tData {
  int d, m, a;
};
void 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();
  }
}
```

Exemplo 7. 17 - Classe Genérica em C++

O mecanismo de *template* usado no exemplo 7.17 parametriza a classe *tPilha* pelo tipo *T* do seu elemento e também pelo valor *tam* do número máximo de elementos que podem ser colocados nas pilhas. A variável tipo *T* também é usada na declaração e definição das operações *empilha* e *obtemTopo* de *tPilha*. Em *main*, o tipo *tPilha* é usado para criar uma pilha capaz de armazenar até 3 inteiros e outra pilha capaz de armazenar até dois objetos da classe *tData*.

A forma de implementação do mecanismo *template* em C++ é curiosa. Ao contrário do que seria mais desejado (o reuso de código fonte e objeto), esse mecanismo só possibilita o reuso de código fonte. Isso significa que não é possível compilar o código usuário das funções ou classes definidas com polimorfismo paramétrico separadamente do código de implementação dessas funções ou classes. De fato, para compilar funções ou classes paramétricas, o compilador necessita saber quais tipos serão associados a elas. A partir de uma varredura do código usuário, o compilador identifica os tipos associados a essas funções e classes e replica todo o código de implementação para cada tipo utilizado, criando assim um código objeto específico para cada tipo diferente utilizado.

A razão para isso tudo é a necessidade do compilador C++ saber o tamanho e forma do tipo para gerar código que aloque memória na pilha. Além disso, para o compilador garantir que qualquer uso inadequado do tipo será detectado em compilação, ele necessita verificar se o tipo passado é compatível com as operações definidas nas implementações das funções paramétricas.

ADA também incorpora o polimorfismo paramétrico permitindo a definição de pacotes genéricos. Embora JAVA tenha reservado a palavra *generic* para uma futura inclusão de polimorfismo paramétrico na LP, esse conceito ainda não foi implementado.

7.2.4 Inclusão

O polimorfismo por inclusão é característico de linguagens orientadas a objetos. Ele se baseia no uso de uma hierarquia de tipos para criar abstrações de dados e controle polimórficas. A idéia fundamental do polimorfismo por inclusão é que elementos dos subtipos são também elementos do supertipo (daí o nome inclusão). Assim, abstrações formadas a partir do supertipo podem também envolver elementos dos seus subtipos.

Um subtipo S de um tipo T é formado por um sub-conjunto dos valores de T. Assim, todo valor de S deve ser também um valor de T. Adicionalmente, cada uma das operações associadas ao tipo T também deve ser aplicável ao subtipo S. Pode-se pensar, portanto, que S herda todas as operações do tipo T.

7.2.4.1 Herança

O conceito de tipo é implementado através de classes em LPs orientadas a objetos. Uma classe define uma representação dos dados (chamados de atributos) e um conjunto de operações (chamadas de métodos). Objetos são instâncias de uma classe e correspondem aos valores do tipo definido

por ela. Subclasses herdam os atributos e métodos de uma classe e, portanto, implementam subtipos do tipo definido por essa classe.

Por exemplo, um objeto *x* tem um ou mais atributos e um ou mais métodos definidos pela sua classe *X*. Um objeto *y*, declarado como sendo uma instância de uma subclasse *Y* de *X*, também possui todos os atributos e métodos de *x*. Além disso, *y* pode ter alguns atributos e métodos adicionais.

Pode-se concluir, portanto, que o mecanismo de herança associa à classe herdeira uma representação inicial para os objetos dessa classe (os atributos herdados) e um conjunto inicial de métodos aplicáveis aos objetos dessa classe (os métodos herdados). A classe herdeira pode conter atributos e métodos adicionais, especializando assim o estado e o comportamento dos novos objetos a serem criados. O exemplo 7.18 mostra o uso de herança em JAVA.

```
// Pessoa.java
public class Pessoa {
  private String nome;
  private int idade;
  public Pessoa (String n, int i) {
    nome = n;
    idade = i;
  }
  public void aumentarIdade () {
    idade++;
  }
}
// Empregado.java
public class Empregado extends Pessoa {
  private float salario;
  public Empregado (String n, int i, float s) {
    super(n, i);
    salario = s;
  }
  public void mudarSalario (float s) {
    salario = s;
  }
}
// Cliente.java
public class Cliente {
  public static void main(String[] args) {
    Pessoa p = new Pessoa ("Denise", 34);
    p.aumentarIdade();
```

```
        Empregado e1 = new Empregado ("Rogerio", 28, 1000.00);
        e1.mudarSalario(2000.00);
        e1.aumentarIdade();
    }
}
```

Exemplo 7. 18 - Herança em JAVA

No exemplo 7.18 são definidas três classes chamadas *Pessoa*, *Empregado* e *Cliente*, cada uma em seu arquivo .JAVA correspondente[7.2]. Uma vez que todo empregado é uma pessoa, a classe *Empregado* é definida por herança a partir da classe *Pessoa*. Como pode ser observado no exemplo, além dos atributos e métodos herdados da classe *Pessoa*, a classe *Empregado* ainda possui um novo atributo *salario*, um método específico para a construção de objetos *Empregado* e outro para a mudança de seu salário. A classe *Cliente* é usada apenas para definir a função *main*, a qual é um programa para criar objetos *Pessoa* e *Empregado* e usar seus métodos.

Uma das vantagens do mecanismo de herança é aumentar a reusabilidade do código. No exemplo 7.18, a herança torna desnecessário redefinir os atributos e métodos da classe *Pessoa* na classe *Empregado*. Destaque-se, em particular, o método *aumentarIdade()*, usado para mudar a idade de uma *Pessoa*, e também aplicado para mudar a idade de um *Empregado*. Isso mostra porque herança é um exemplo de polimorfismo universal. Por fim, cabe esclarecer que a chamada *super(n,i);* na primeira linha do construtor da classe *Empregado*, é a forma adotada por JAVA para chamar o construtor da classe herdada quando o mesmo possui parâmetros.

7.2.4.1.1 Especificador de Acesso para Classes Herdeiras

A herança traz consigo uma demanda por um novo tipo de especificador de acesso usado para possibilitar às classes herdeiras terem acesso livre aos atributos da classe herdada sem que para isso seja necessário tornar esses atributos públicos ou criar métodos públicos de acesso na classe herdada.

Considere ainda o exemplo 7.18. Caso se desejasse criar uma função em *Empregado* para responder se um empregado já pode se aposentar, seria necessário acessar o campo *idade* definido em *Pessoa*. Contudo, isso não é possível pois esse atributo foi declarado como *private* e, portanto, só pode ser acessado diretamente pelos métodos de *Pessoa*. A alternativa de se criar métodos públicos de acesso a esse atributo em *Pessoa* pode não

[7.2] Classes públicas em JAVA devem ser definidas em um arquivo .JAVA de mesmo nome da classe. Os próximos exemplos deixarão subentendido que cada classe pública é escrita em seu arquivo .JAVA correspondente.

ser satisfatória porque isso significaria tornar esses métodos disponíveis para métodos de qualquer outra classe.

A solução adotada por JAVA e C++ para esse problema foi criar um novo especificador de acesso para os atributos da classe. Esse especificador foi denominado como *protected*. Enquanto ele libera o acesso ao atributo da classe para os métodos das suas subclasses, ele ao mesmo tempo impede o acesso para os métodos de outras classes. O exemplo 7.19 ilustra o uso desse especificador de acesso em JAVA.

```
public class Pessoa {
  protected int idade;
}
public class Empregado extends Pessoa {
  public Empregado (int i) { idade = i; }
  public boolean aposentavel() {
    if (idade >= 65) return true;
    return false;
  }
}
public class Cliente {
  public static void main(String[] args) {
    Empregado e = new Empregado (32);
    if (e.aposentavel()) System.out.println("Chega de trabalho!");
    // e.idade = 70;
  }
}
```

Exemplo 7. 19 - Especificador de Acesso para Classes Herdeiras

No exemplo 7.19 a classe *Pessoa* possui um único atributo *idade* declarado como sendo *protected*. Os dois métodos da subclasse *Empregado* agora têm livre acesso ao atributo de *Pessoa*. No entanto, caso se tentasse compilar a linha comentada, haveria erro de compilação uma vez que a classe *Cliente* não é uma subclasse[7.3] de *Pessoa* e, portanto, não pode acessar o seu atributo protegido.

7.2.4.1.2 Inicialização de Atributos com Herança

Quando se cria um objeto de uma classe é necessário inicializar os seus atributos e, em seguida, chamar o construtor dessa classe. Contudo, quando essa classe é uma subclasse, ele herda atributos que também precisam ser inicializados. Isso precisa ser feito antes da chamada do contrutor da

[7.3] Considera-se a classe Cliente como pertencente a outro pacote pois métodos de classes de um pacote também têm livre acesso aos atributos protegidos das demais classes do pacote.

classe. Dessa forma, o procedimento adotado na criação de um objeto deve ser realizado na superclasse antes de ser feito na classe. Esse processo deve ser aplicado recorrentemente para cada nível de herança existente. O exemplo 7.20 mostra uma situação na qual isso ocorre em JAVA.

```
class Estado {
  Estado(String s) {
    System.out.println(s);
  }
}
class Pessoa {
  Estado p = new Estado("Ativo");
  Pessoa () {
    System.out.println("Pessoa");
  }
}
class Idoso extends Pessoa {
  Estado i = new Estado("Sabio");
  Idoso () {
    System.out.println("Idoso ");
  }
}
class Avo extends Idoso {
  Estado a1 = new Estado ("Alegre");
  Avo() {
    System.out.println("Avo");
    a3 = new Estado ("Orgulhoso");
  }
  Estado a2 = new Estado ("Amigo");
  void fim() {
    System.out.println("Fim");
  }
  Estado a3 = new Estado ("Satisfeito");
}
public class Inicializacao {
  public static void main(String[] args) {
    Avo a = new Avo();
    a.fim();
  }
}
```

Exemplo 7. 20 - Inialização com Herança

O resultado obtido no exemplo 7.20 será a impressão linha a linha da seqüência *Ativo Pessoa Sabio Idoso Alegre Amigo Satisfeito Avo Orgulhoso*

Fim. Um detalhe importante diz respeito à ordem de inicialização dos atributos da classe *Avo*. Em JAVA (e C++) eles são inicializados na ordem na qual foram declarados. Embora o processo de inicialização do exemplo 7.20 tenha sido simples e fácil de entender, isso nem sempre ocorre assim. Outros fatores podem ser complicadores, tais como a existência de atributos de classe e a necessidade de passar argumentos para os construtores.

7.2.4.1.3 Sobrescrição

Em algumas situações o método herdado de uma classe pode não ser o mais adequado para realizar a mesma operação nos objetos de suas subclasses. Para lidar com essas situações, LPs orientadas a objetos permitem que os métodos de uma classe sejam sobrescritos em suas subclasses. Assim, pode-se criar, em uma subclasse, um novo método com a mesma interface (nome, parâmetros e resultado) do método na classe herdada. Se um método é sobrescrito em uma subclasse, então o novo método é visível para os usuários dessa classe. O exemplo 7.21 mostra o uso do mecanismo de sobrescrita em JAVA.

```
class XY {
  protected int x = 3, y = 5;
  public int soma () {
    return x + y;
  }
}
class XYZ extends XY {
  int z = 17;
  public int soma () {
    return x + y + z;
  }
}
public class Sobrescrita {
  public static void main(String[] args) {
    XYZ xyz = new XYZ();
    System.out.println(xyz.soma());
  }
}
```

Exemplo 7. 21 - Método Sobrescrito em JAVA

O método *soma* da classe *XY* é sobrescrito na classe *XYZ*. Assim, quando objetos da classe *XYZ* (tal como *xyz* no método *main* da classe *Sobrescrita*) chamam o método *soma*, o método da classe *XYZ* será chamado ao invés do método homônimo que seria herdado.

Muitas vezes, só se necessita estender a funcionalidade produzida pelo método sobrescrito de modo a complementar a funcionalidade com o que é específico da classe herdeira. Para evitar repetição de código, é possível invocar o método sobrescrito de dentro do método da subclasse. O exemplo 7.22 mostra como isso pode ser feito no método *soma* da classe *XYZ*, apresentada no exemplo 7.21.

```
class XYZ extends XY {
  int z = 17;
  public int soma () {
    return super.soma() + z;
  }
}
```

Exemplo 7. 22 - Extensão de Método

Para permitir a chamada do método sobrescrito, é necessário utilizar o nome *super* no método da subclasse. Caso contrário, o compilador entenderia como sendo uma chamada recursiva do próprio método que está sendo definido. C++ usa uma sintaxe baseada no operador de resolução de escopo *::* para oferecer essa facilidade. Portanto, a chamada equivalente ao *super.soma()* de JAVA seria *XY::soma()* em C++

A sobrescrição de métodos é uma outra maneira de se fazer polimorfismo de sobrecarga, uma vez que existe uma implementação específica do método para cada classe. O fato da lista de parâmetros dos métodos sobrescritos ser aparentemente a mesma nas diferentes implementações pode ser fonte de confusão. Contudo, não se pode esquecer a existência de um parâmetro implícito passado para os métodos. Esse parâmetro é o próprio objeto sobre o qual o método vai ser aplicado. Desse modo, a lista de parâmetros de cada método é diferenciada, caracterizando ainda mais o polimorfismo de sobrecarga.

7.2.4.2 Identificação Dinâmica de Tipos

O polimorfismo por inclusão permite ainda a elaboração de trechos de código polimórficos nos quais métodos invocados por um mesmo referenciador de objetos se comportam de maneira diferenciada de acordo com o tipo verdadeiro do objeto sendo manipulado naqueles trechos. Isso pode ser útil em situações nas quais o programador desconhece o tipo verdadeiro de um objeto.

Nessas situações, objetos diferentes podem ser tratados de modo semelhante. Tal característica confere grande versatilidade as classes e programas que lhes são beneficiários. No entanto, para oferecer isso, a LP deve possuir um meio de identificar o tipo do objeto em tempo de execu-

ção, ou seja, realizar identificação dinâmica de tipos. Essa seção discute como isso pode ser feito e utilizado.

7.2.4.2.1 Ampliação

Uma distinção comum realizada em orientação a objetos é entre membros e instâncias de uma classe. Instâncias de uma classe são os objetos declarados exclusivamente como sendo daquela classe. Membros de uma classe são todas as instâncias da classe e de suas subclasses. Considere, por exemplo, um objeto *x*, instância da classe *X*, e um objeto *y,* instância da classe *Y*, subclasse de *X.* Enquanto o único membro da classe *Y* é o objeto *y,* a classe *X* possui *x* e *y* como membros.

Em orientação a objetos, é sempre legal atribuir todos os membros de uma classe a objetos declarados como sendo dessa classe. Assim, é possível atribuir a objetos da classe *X* tanto objetos dessa classe (como *x*) quanto da classe *Y* (como *y).* Por outro lado, não é sempre possível atribuir a objetos da classe *Y*, objetos da classe *X* (por exemplo, *x*) uma vez que eles não são membros dessa classe.

Ampliação (`upcasting`) é o termo usado para descrever o movimento de objetos na sua linha de ancestrais no sentido da subclasse para as superclasses. Em outras palavras, é a capacidade de uma instância de uma subclasse poder aparecer quando um membro de uma superclasse é solicitado. O exemplo 7.23 usa as classes *Pessoa* e *Empregado*, definidas no exemplo 7.18, para ilustrar a realização de ampliação em JAVA.

```
public class Cliente {
  public void paga (Pessoa pes) {}
  public void contrata (Empregado emp) {}
  public static void main(String[ ] args) {
    Pessoa p = new Pessoa ("Lucas", 30);
    Empregado e = new Empregado ("Luis", 23, 1500.00);
    p = e;
    // e = p;
    Cliente c = new Cliente();
    c.paga(e);
    // c.contrata(p);
```

Exemplo 7. 23 - Ampliação em JAVA

A ampliação de um *Empregado* para uma *Pessoa* é sempre legal e ocorre tanto na atribuição quanto na chamada do método *paga*. Já as duas linhas comentadas são ilegais porque uma *Pessoa* não pode aparecer quando se espera um *Empregado*. O compilador de JAVA geraria erros de compilação caso elas não estivessem como comentários.

A ampliação é uma operação sempre válida pois garante a segurança do sistema de tipos. Ao se mover para cima na linha de ancestrais se converge para uma classe mais geral. A única perda possível é não se poder mais usar os métodos específicos da subclasse. O objeto só poderá usar os métodos da superclasse, os quais são sempre legalmente aplicáveis para os objetos das subclasses. Por causa dessa segurança, os compiladores de LPs orientadas a objetos permitem a realização de ampliação sem a necessidade de uma operação explícita de conversão de tipos.

C++ também permite a realização de ampliação, mas só quando se usa ponteiros ou referências para os objetos. O exemplo 7.24 mostra um trecho de código em C++ no qual ocorre ampliação.

```
Pessoa p, *q;
Empregado e, *r;
q = r;
// r = q;
// p = e;
// e = p;
```

Exemplo 7. 24 - Ampliação em C++

A atribuição do ponteiro *r* para o ponteiro *q* é uma ampliação válida. Já as três linhas comentadas no exemplo 7.24 são ilegais. Na primeira delas, embora se use ponteiros, o objeto apontado por *q* não é membro de *Empregado* e, portanto, não pode ser apontado por *r*. Na linha seguinte, embora seja uma operação de ampliação, ela não é válida pois não se trata de uma atribuição de ponteiros. Na última linha, a atribuição não é válida porque não se tem ampliação e também porque não se trata de uma operação sobre ponteiros.

O motivo dessa limitação em C++ é o mecanismo de cópia de objetos utilizado pela operação de atribuição e para passagem de parâmetros por valor. Se fosse permitido fazer esse tipo de cópia em uma operação de ampliação haveria perda de informação quando se atribuísse um objeto da subclasse a um da superclasse (os campos adicionais da subclasse seriam perdidos).

7.2.4.2.2 Amarração Tardia de Tipos

Considere uma situação, durante a execução de um programa, na qual já tenha havido uma operação de ampliação de um objeto *y* de uma classe *Y* para um objeto *x* da superclasse *X*. Considere também que um método da superclasse *X* tenha sido sobrescrito em *Y*. A chamada desse método através de *x* invoca a execução do método definido em *Y*, ao invés do definido em *X*. O mecanismo pelo qual esse processo é realizado se chama a-

marração tardia de tipos. O exemplo 7.25 mostra um trecho de código em JAVA no qual isso ocorre.

```
class Pessoa {
  String nome;
  int idade;
  public Pessoa (String n, int i) {
    nome = n;
    idade = i;
  }
  public void aumentaIdade () { idade++; }
  public void imprime(){
    System.out.print(nome + " , " + idade);
  }
}
class Empregado extends Pessoa {
  float salario;
  Empregado (String n, int i, float s) {
    super(n, i);
    salario = s;
  }
  public void imprime(){
    super.imprime();
    System.out.print(" , " + salario);
  }
}
public class Cliente {
  public static void main(String[] args) {
    Pessoa p = new Empregado ("Rogerio", 28, 1000.00);
    p.aumentaIdade();
    p.imprime();
  }
}
```

Exemplo 7. 25 - Amarração Tardia de Tipos em JAVA

O programa do exemplo 7.25 cria um objeto da classe *Empregado* e o amplia imediatamente para a superclasse *Pessoa*. Então, o método *aumentarIdade*, definido em *Pessoa* e herdado por *Empregado*, é chamado. Em seguida, o método *imprime* de *Empregado* é invocado, muito embora isso seja feito através de *p*, o qual foi declarado como sendo da classe *Pessoa*.

Note que a decisão tomada a respeito de qual método executar, quando *imprime* é chamado, só pode ser tomada em tempo de execução. Se essa decisão tivesse de ter tomada em tempo de compilação, como *p* foi decla-

rado como da classe *Pessoa*, o método invocado teria de ser o dessa classe pois o compilador não teria como saber para qual tipo *p* estaria referenciando no momento da execução.

É importante entender como funciona o mecanismo de amarração tardia de tipos. A figura 7.2 ilustra como esse mecanismo pode ser implementado[7.4].

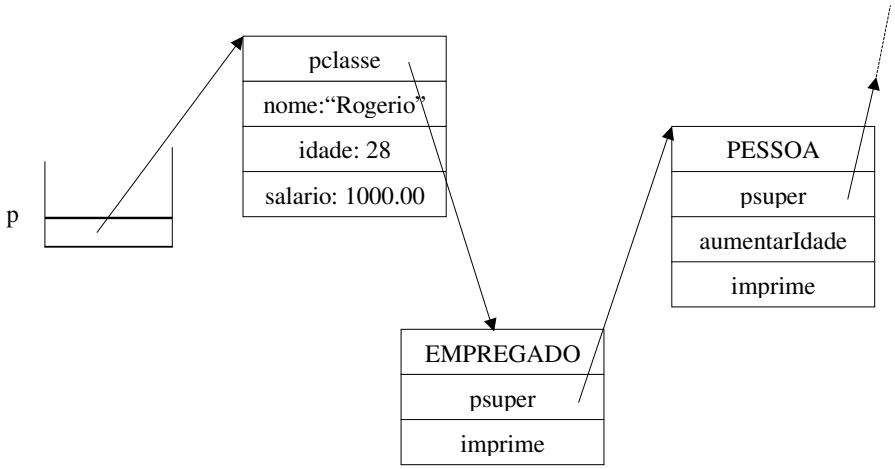

Figura 7. 2 - Mecanismo de Amarração Tardia de Tipos

A figura 7.2 retrata o estado do programa do exemplo 7.25 após a execução da linha na qual o objeto da classe *Empregado* é criado. A variável *p*, declarada como *Pessoa*, é alocada na base da pilha de registros de ativação e aponta para o objeto recém-criado da classe *Empregado*. Além dos atributos de sua classe, esse objeto possui uma referência (indicada na figura pelo nome *pclasse*) para a tabela de métodos de sua classe (no caso, *Empregado*). Essa tabela, por sua vez, também possui uma referência (indicada na figura pelo nome *psuper*) para a tabela de métodos de sua superclasse. Isso se repete sucessivamente, até que se chegue a uma classe sem superclasse. Quando um método é invocado por *p*, essa cadeia de ponteiros é seguida até chegar a primeira tabela de métodos da classe na qual esse método foi definido. Assim, na chamada do método *aumentaIdade*, o método invocado é o de *Pessoa* e na chamada do método *imprime*, o método invocado é o de *Empregado*.

O mecanismo de amarração tardia de tipos oferece maior flexibilidade para a escrita de código reutilizável. Contudo, a eficiência computacional se reduz quando comparada com a obtida com amarração estática. Na amarração tardia é necessário manter essa cadeia de ponteiros e segui-la em todas as chamadas de métodos. Nada disso é necessário quando o subprograma a ser chamado é definido em tempo de compilação.

[7.4] Essa não é a única forma possível de implementar o mecanismo de amarração tardia de tipos.

Por não desejar comprometer a eficiência, C++ adota uma postura diferente da de JAVA. Em C++, o implementador da classe pode decidir se deseja o uso de amarração tardia ou não para cada método da classe. Para o método ter amarração tardia, a sua declaração deve vir precedida da palavra *virtual*. Se essa palavra é omitida, a amarração é estática. Em C++, portanto, o programador deve decidir caso a caso se deseja versatilidade ou eficiência. O exemplo 7.26 mostra a definição de uma classe com métodos que utilizam amarração estática e amarração tardia em C++.

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

Exemplo 7. 26 - Amarração Estática e Amarração Tardia em C++

No exemplo 7.26, o método *ler* é amarrado estaticamente e o método imprimir é amarrado tardiamente. Sempre que um ponteiro para a classe *Pessoa* invocar *ler* é executada a função definida em *Pessoa*, independentemente do tipo do objeto que está sendo apontado. Por outro lado, sempre que *imprimir* for invocado, o método executado será o definido na classe do objeto sendo apontado.

A amarração tardia de tipos possibilita a criação de código usuário com polimorfismo universal, isto é, código usuário capaz de operar uniformemente sobre objetos de tipos diferentes. O exemplo 7.27 ilustra essa possibilidade.

```
class Militar {
  void operacao(){}
}
class Exercito extends Militar {
  void operacao(){System.out.println("Marchar");}
}
class Marinha extends Militar {
  void operacao(){System.out.println("Navegar");}
}
class Aeronautica extends Militar {
  void operacao(){System.out.println("Voar");}
}
public class Treinamento {
  public static void treinar(Militar[] m) {
    for (int i = 0; i < m.length; i++) {
      m[i].operacao();
    }
```

```
  }
  public static void main (String[] args) {
    Militar[] m = new Militar[] {
      new Exercito(), new Marinha(),
      new Aeronautica(), new Militar()
    }
    treinar(m);
  }
}
```

Exemplo 7. 27 - Amarração Tardia e Polimorfismo Universal

O método de classe *treinar* da classe *Treinamento* é polimórfico universal pois opera sobre qualquer elemento do vetor *m* independentemente desse elemento ser um objeto da classe *Exercito*, *Marinha*, *Aeronautica* ou mesmo *Militar*.

7.2.4.2.3 Classes Abstratas

Classes abstratas possuem membros, mas não possuem instâncias. Embora não se possa criar instâncias de classes abstratas, elas possuem como membros as instâncias de suas subclasses não abstratas (também chamadas de classes concretas). Portanto, uma classe abstrata deve ser necessariamente estendida, ou seja, deve ser uma classe herdada por outra, mais específica, contendo os detalhes não incluídos na classe abstrata.

Classes abstratas são especialmente 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. O exemplo 7.27 é um caso típico no qual uma classe abstrata seria útil. A classe *Militar* só existe para especificar as características comuns de suas subclasses *Exercito*, *Marinha* e *Aeronautica*. Qualquer membro de *Militar* deve necessariamente ser uma instância dessas classes pois só existem militares nessas três forças armadas. O exemplo 7.28 mostra como a classe *Militar* poderia ter sido definida como abstrata no exemplo 7.27. Para tornar uma classe abstrata em JAVA basta incluir a palavra *abstract* como prefixo da definição.

```
abstract class Militar {
  void operacao(){}
}
```

Exemplo 7. 28 - Classe Abstrata em JAVA

Com essa definição de *Militar* não é mais possível criar instâncias dessa classe. Assim, ocorreria erro ao se tentar compilar a classe *Treinamento*

do exemplo 7.27, uma vez que a função *main* dessa classe tentaria criar uma instância de *Militar*.

Classes abstratas normalmente possuem um ou mais métodos abstratos, isto é, métodos declarados, mas não implementados. A implementação desses métodos é deixada para as suas subclasses. Por exemplo, o método *operacao* de *Militar* deveria ser abstrato, uma vez que não possui qualquer instrução. O exemplo 7.29 mostra como isso pode ser feito em JAVA.

```
abstract class Militar {
  abstract void operacao();
}
```

Exemplo 7. 29 - Classe Abstrata com Método Abstrato em JAVA

O fato de classes abstratas poderem possuir métodos abstratos não significa que classes abstratas não podem possuir métodos definidos e atributos próprios. De fato, classes abstratas podem até possuir construtores, embora eles nunca possam ser chamados para criar instâncias dessa classe. Eles só podem ser chamados no momento da construção das instâncias das subclasses da classe abstrata. O exemplo 7.30 redefine a classe abstrata *Militar* para incluir o atributo *patente* e o método concreto (não abstrato) *toString*. Observe também a inclusão do método construtor (não abstrato) nessa nova definição.

```
abstract class Militar {
  String patente;
  Militar(String p) { patente = p; }
  String toString() { return patente; }
  abstract void operacao();
}
```

Exemplo 7. 30 - Classe Abstrata com Atributo, Método Abstrato e Método Concreto

Em JAVA, é possível, mas não é comum, criar classes abstratas nas quais nenhum método é abstrato. Já em C++, para uma classe ser abstrata, é obrigatório possuir pelo menos um método abstrato. De fato, a presença de um método abstrato caracteriza a classe como abstrata. A sintaxe adotada por C++ para definir um método abstrato também é estranha, como pode ser vista no exemplo 7.31.

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

Exemplo 7. 31 - Classe Abstrata em C++

A classe *Militar* do exemplo 7.31 é abstrata porque possui o método abstrato *operacao*. A indicação de que *operacao* é um método abstrato se dá pela inclusão da terminação *=0*. Métodos abstratos em C++ devem necessariamente ser declarados como virtuais, uma vez que o seu comportamento terá de ser definido nas subclasses da classe abstrata. A classe *Militar* do exemplo 7.31 também possui o método concreto, e não virtual, *imprime*.

Classes abstratas são um meio conveniente para disciplinar a construção de classes. Para cada método abstrato em uma classe abstrata, todas as suas subclasses devem implementar esse método ou também devem ser abstratas.

Afim de exemplificar a conveniência das operações abstratas, considere uma aplicação em JAVA envolvendo um conjunto de formas geométricas que devem ser mostradas na tela do vídeo. Diferentes algoritmos são necessários para mover as diferentes formas. De modo a impor a necessidade de operações distintas para cada tipo de forma, uma classe abstrata *Forma* pode ser criada, tal como no exemplo 7.32.

```
abstract class Forma {
  abstract void mover(int dx, int dy);
  abstract void desenhar();
}
```

Exemplo 7. 32 – Especificação por Classe Abstrata

As várias formas geométricas da aplicação (por exemplo, *Circulo*, *Retangulo*, *Triangulo*, etc.) podem ser implementadas a partir da classe *Forma*. O exemplo 7.33 mostra a implementação de uma dessas subclasses.

```
class Circulo extends Forma {
  int x, y, raio;
  Circulo (int x, int y, int r) {
    this.x = x;
    this.y = y;
    raio = r;
  }
  void mover(int dx, int dy) {
    x += dx;
    y += dy;
  }
  void desenhar() {
    System.out.println ("Circulo:");
    System.out.println ("  Origem: (" + x + ", " + y +")");
    System.out.println ("  Raio: " + raio);
  }
```

```
    }
}
```

Exemplo 7. 33 - Implementação de Especificação por Classe Abstrata

Caso os métodos *mover* e *desenhar* não fossem definidos na classe *Circulo*, haveria erro de compilação, uma vez que uma subclasse de uma classe abstrata deve necessariamente implementar os métodos abstratos (no caso de ser concreta) ou adiar a sua implementação (declarando-se, e declarando os métodos abstratos não implementados, como *abstract*).

A conveniência de disciplinar a construção de classes através de operações abstratas é tão significativa que os projetistas de JAVA incluíram o conceito de *interface*[7.5] para designar classes abstratas nas quais todos os métodos são abstratos. Interfaces também não podem definir atributos de objetos.

Pode-se resumir, portanto, como principais vantagens da utilização de classes abstratas a melhoria da organização hierárquica de classes, através do encapsulamento de atributos e métodos na raiz da estrutura; a promoção de uma maior disciplina na programação, visto que força o comportamento necessário nas suas subclasses; e o incentivo ao uso de amarração tardia, permitindo um comportamento mais abstrato e genérico para os objetos.
Deve-se ressaltar que nem todas as superclasses devem ser classes abstratas. Por exemplo, poder-se-ia implementar uma classe *Arco* como uma subclasse da classe *Circulo*. Instâncias da classe *Circulo* são ainda úteis, apesar de se utilizar essa classe como superclasse da classe *Arco*. Além disso, também é possível criar subclasses abstratas de classes concretas. Isso ilustra como uma hierarquia de classes pode crescer através do processo de especialização das classes existentes. É preciso atentar, contudo, para o projeto dessa hierarquia para evitar torná-la mais complexa do que o realmente necessário.

7.2.4.2.4 Estreitamento

Estreitamento (`downcasting`) é o termo usado para descrever a conversão de tipos de objetos no sentido da superclasse para as subclasses. A operação de estreitamento não é completamente segura 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. Em outras palavras, o objeto sob o qual se fará estreitamento pode ser uma instância de uma subclasse diferente daquela para a qual será feito a conversão. Nesse caso,

[7.5] Outro motivo para a inclusão desse conceito em JAVA será visto ainda nesse capítulo na seção de herança múltipla.

as operações específicas da nova classe do objeto serão incompatíveis com a classe original do objeto.

Por exemplo, se for feito o estreitamento de uma *Forma*, ela pode ser um círculo, um quadrado, um triângulo, ou qualquer outra subclasse de *Forma*. Se a *Forma* for um *Circulo* e o estreitamento for feito para a classe *Quadrado*, na hora de calcular a área do quadrado, a operação não poderia ser feita consistentemente visto que um círculo não possui lado. É por esses problemas de violação do sistema de tipos que algumas LPs não permitem que se faça estreitamento e outras exigem que ele seja feito através de uma operação de conversão explícita (`cast`).

JAVA somente permite a realização de estreitamento através do uso de uma 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 realmente pertença a outra linha de descendência, ocorrerá uma exceção[7.6] em tempo de execução. Para que o programador possa evitar esse último tipo de erro, JAVA oferece o operador *instanceof*, o qual permite testar dinamicamente se o objeto designado pela superclasse realmente é da classe para a qual se deseja fazer a conversão. O exemplo 7.34 ilustra como isso é feito 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;
  }
}
```

Exemplo 7. 34 - Estreitamento em JAVA

A primeira linha da função *main* de *Estreitamento* cria um objeto de *umaSubclasse*, o qual é ampliado para a referência *uc* de *umaClasse*. Na linha seguinte é feito o estreitamento de *uc* para *us*, declarado como *umaSubclasse*. Essa operação é válida pois *umaClasse* e *umaSubclasse* estão

[7.6] Exceções são discutidas com maior profundidade no capítulo 8 desse livro.

na mesma linha de descendência e porque *uc* está se referindo realmente a uma instância de *umaSubclasse*.

As duas linhas comentadas de *main* causariam erros caso não fossem comentários. Na primeira linha comentada, o erro seria de compilação pois *umaSubclasse* e *outraSubclasse* não se encontram na mesma linha de descendência. Esse tipo de erro é verificado pelo compilador da mesma maneira como na verificação da validade da operação de ampliação. Isso implica na necessidade do compilador poder acessar, durante a compilação, as informações sobre todas as classes da linha de ascendência de uma classe usada no programa.

Na segunda linha comentada, o erro seria detectado em tempo de execução, o que provocaria o disparo de uma exceção. Embora *umaClasse* e *outraSubclasse* se encontrem na mesma linha de descendência, a referência a *umaClasse* está designando realmente um objeto de *umaSubclasse*, o que provocaria uma conversão inválida. Note que esse tipo de erro só pode ser detectado em tempo de execução porque, de modo geral, só se sabe o tipo do objeto designado por uma referência ao longo da execução.

Erros tais como o mencionado no parágrafo anterior podem ser evitados usando-se previamente o operador *instanceof* para testar se a conversão é válida, isto é, se o objeto a ser convertido realmente é membro da classe para a qual será feita a conversão. O uso de *instanceof* é ilustrado no comando *if* na última linha de *main*.

É recomendável usar *instanceof* antes de um estreitamento especialmente quando não se tem outra informação indicando o tipo de objeto sobre o qual será feita a operação. Caso isso não seja feito e a conversão for indevida, ocorrerá a exceção *ClassCastException*.

Cabe ressaltar que a conversão de um objeto pode ser feita ou testada para cada classe da qual ele é membro. Isso significa que o estreitamento de um objeto não precisa necessariamente ser feito para a classe da qual ele é instância. O estreitamento também pode ser feito para as suas superclasses.

Para fazer a verificação dinâmica do estreitamento ou usar *instanceof*, é necessário seguir a cadeia de ponteiros ilustrada na figura 7.2. Caso a classe para a qual se deve fazer a conversão for encontrada em algum momento, a operação é válida.

C++ oferece três maneiras diferentes para permitir a realização de estreitamento: o mecanismo usual de conversão explícita (`cast`), o mecanismo de conversão explícita estática (*static_cast*) e o mecanismo de conversão explícita dinâmica (*dynamic_cast*). C++ também oferece o meca-

nismo de verificação dinâmica de tipos (*typeid*). Considere o exemplo 7.35.

```
#include <typeinfo>
class UmaClasse {
public:
  virtual void temVirtual () {};
};
class UmaSubclasse: public UmaClasse {};
class OutraSubclasse: public UmaClasse {};
class OutraClasse {};
main () {
  // primeira parte do exemplo
  UmaClasse* pc = new UmaSubclasse;
  OutraSubclasse* pos = dynamic_cast <OutraSubclasse*> (pc);
  UmaSubclasse* ps = dynamic_cast <UmaSubclasse*> (pc);
  // 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);
}
```

Exemplo 7. 35- Estreitamento em C++

Na primeira parte do exemplo 7.35 é mostrado o uso do mecanismo de *dynamic_cast*. Essa é uma forma de fazer estreitamento de modo seguro. Ao usar *dynamic_cast* o que se está tentando fazer é um estreitamento para um tipo particular. O valor de retorno dessa operação será um ponteiro para o tipo desejado, no caso do estreitamento ser apropriado. De outra forma, o valor retornado será zero (*null*) para indicar que o tipo não era o esperado.

Ao se usar *dynamic_cast* trabalha-se com a verdadeira hierarquia do polimorfismo. Somente se pode usar o *dynamic_cast* em classes com funções virtuais. Isso ocorre porque o *dynamic_cast* usa a informação armazenada em uma tabela de métodos virtuais para determinar o tipo atual. No exemplo, o ponteiro *pos* receberá zero pois o estreitamento para *OutraSubclasse** é incorreto. É responsabilidade do programador verificar se

o resultado do estreitamento por *dynamic_cast* é diferente de zero. Tal como a verificação dinâmica em JAVA, a operação de *dynamic_cast* sobrecarrega um pouco a execução do programa. Portanto, se um programa usa muito *dynamic_cast*, isso poderá diminuir a eficiência de execução. Nesse caso é aconselhável reavaliar o projeto do programa.

Em alguns casos é possível saber durante a própria redação do programa com qual tipo se está lidando no local do estreitamento. Nessas situações, a sobrecarga adicional de se usar *dynamic_cast* é desnecessária. É melhor fazer um *static_cast*, pois assim a verificação é feita em tempo de compilação.

Usar *static_cast* para realizar estreitamento também é melhor do que o mecanismo de conversão explícita tradicional (`cast`) pois o primeiro não permite fazer conversões fora da hierarquia de classes, o que é permitido pelo segundo. Como a eficiência é mesma para códigos gerados usando ambos mecanismos, *static_cast* deve ser preferido pois é mais seguro.

A segunda parte do exemplo mostra o uso do mecanismo de *static_cast*. Nessa parte, um objeto (*us*) de *UmaSubclasse* é criado e é feita uma ampliação para um ponteiro para *UmaClasse*. Essa mesma operação é repetida usando *static_cast*. Como se trata de uma ampliação, não existe uma obrigatoriedade de usar *static_cast*, mas seu uso pode ser conveniente para tornar mais explícita a ampliação e para evitar a realização equivocada de conversões fora da hierarquia de classes.

Após as operações de ampliação, o exemplo mostra a diferença entre se fazer estreitamento com *static_cast* e o mecanismo tradicional. Com o mecanismo tradicional é possível fazer a conversão fora da hierarquia de classes entre os ponteiros para *UmaClasse* e para *OutraClasse*. Isso não é permitido no caso do *static_cast*. Caso a linha de código na qual essa operação é feita não estivesse comentada, ocorreria um erro de compilação.

Na terceira parte do exemplo, o identificador especial *typeid* é usado para detectar dinamicamente os tipos dos ponteiros e o *static_cast* é usado para fazer o estreitamento. Contudo, esse uso de *typeid* no exemplo produz efetivamente o mesmo efeito do mecanismo usando *dynamic_cast* pois o programador deve fazer algum teste para descobrir se a conversão obteve sucesso. Além disso, o uso de *typeid* torna o código com *dynamic_cast* tão eficiente quanto esse.

Resumindo, é boa política usar preferencialmente os mecanismos de *dynamic_cast* e *typeid*. Embora seja mais rápido fazer estreitamento estaticamente, a conversão dinâmica é mais segura pois os mecanismos estáticos de conversão podem produzir conversões inapropriadas.

Existem controvérsias a respeito do uso de estreitamento. Para alguns, o uso de estreitamento indica um projeto de sistema inadequado. Esses acreditam que operações de estreitamento vão contra os princípios do projeto orientado a objetos, uma vez que a verdadeira origem do objeto necessita ser conhecida ou presumida para que o programa funcione apropriadamente.

No caso da origem ser presumida, isso pode provocar violação no sistema de tipos (caso a pressuposição seja equivocada). No caso da origem ser conhecida através de alguma espécie de teste dinâmico, perde-se bastante a modificabilidade proporcionada pela orientação a objetos, uma vez que o código usuário da hierarquia de classes necessita incluir operações para testar o tipo do objeto sendo manipulado. Nesse caso, quando uma nova subclasse é incluída na hierarquia, ao contrário do apregoado pela orientação a objetos, o código usuário terá de ser modificado para levar em conta os objetos dessa nova classe.

Para outros, a ausência de estreitamento é considerada uma limitação significativa no poder expressivo da linguagem. Segundo eles, existem situações nas quais a solução através de estreitamento é muito mais fácil de ser redigida e manipulada.

Considere novamente, por exemplo, a situação na qual temos uma lista de formas (círculos, triângulos, etc.) apresentadas na tela. Se a operação de `highlight` de um tipo de forma (por exemplo, triângulos) necessita ser realizada, é necessário percorrer a lista identificando o tipo do objeto e realizando a operação de `highlight` sobre ele caso seja um triângulo.

A solução puramente orientada a objetos seria criar um método `highlight` na classe base com um parâmetro indicando qual tipo deveria ser submetido a essa operação. Cada subclasse teria de sobrescrever esse método. Essa solução não é boa porque nem sempre essa operação de `highlight` deve ser feita em todas as subclasses, o que acaba retirando a naturalidade do projeto orientado a objetos, isso sem falar no aumento de métodos na classe base e de métodos desnecessários nas subclasses.

7.2.4.3 Herança Múltipla

A herança pode ser simples ou múltipla. Se uma classe herda de uma única classe temos herança simples. Se ela herda de mais de uma classe temos herança múltipla. A figura 7.3 mostra diagramas de classes com exemplos de herança simples e herança múltipla.

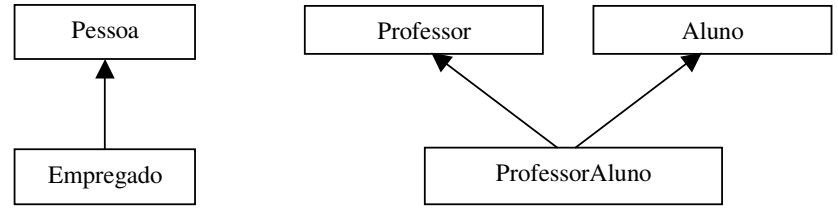

Figura 7. 3 - Herança Simples x Herança Múltipla

Herança múltipla é considerada importante em orientação a objetos porque alguns objetos no mundo real pertencem a várias classes. Contudo, nem todas linguagens orientadas a objetos permitem a realização de herança múltipla. Esse é o caso de SMALLTALK. Outras, como C++, suportam herança múltipla. Por sua vez, JAVA adota uma postura intermediária, suportando herança simples e uma forma restrita de herança múltipla.

Em linguagens nas quais não existe herança múltipla, quando uma classe necessita herdar de mais de uma classe, ela deve incluir objetos das classes extras como atributos da classe, provocando a inserção de uma significativa quantidade de métodos envoltórios (isto é, métodos usados apenas para chamar métodos de outras classes) ou mesmo a duplicação de código. O exemplo 7.36 apresenta uma possível implementação em JAVA para a classe *AlunoProfessor* mostrada na figura 7.3.

```
class Professor {
        String n = "Marcos";
        int matr = 53023;
        public String nome() { return n; };
        public int matricula() { return matr; };
}
class Aluno {
        String n = "Marcos";
        int matr = 127890023;
        float coef = 8.3;
        public String nome() { return n; };
        public int matricula() { return matr; };
        public float coeficiente() { return coef; };
}
class ProfessorAluno extends Professor {
        Aluno aluno = new Aluno();
        public float coeficiente() { return aluno.coeficiente(); };
        public String matriculaAluno() { return aluno.matricula(); };
}
```

Exemplo 7. 36 - Herança Simples em JAVA

Observe a necessidade de se incluir um atributo *aluno* e os métodos envoltórios *coeficiente* e *matriculaAluno* na classe *ProfessorAluno* para também fazê-la se comportar como um *Aluno*. Observe ainda a duplicação do atributo *nome* em *ProfessorAluno*. Tal duplicação é desnecessária uma vez que somente o nome herdado de *Professor* é utilizado em *ProfessorAluno*. Isso faz sentido visto que, em geral, uma pessoa só possui um nome, mas implica em desperdício de espaço de memória.

Esse desperdício poderia ser evitado criando-se atributos para abrigar o coeficiente e a matrícula do aluno em *ProfessorAluno*. Nesse caso, o atributo *aluno* não seria mais necessário. No entanto, essa alternativa implicaria em não poder reutilizar os métodos de *Aluno* em *ProfessorAluno*, o que acarretaria a necessidade de uma implementação específica desses métodos em *ProfessorAluno*.

Um problema mais sério com essa solução é impedir a subtipagem múltipla, isto é, impedir um objeto do tipo *ProfessorAluno* participar de uma estrutura de dados contendo objetos do tipo *Aluno*, uma vez que *ProfessorAluno* deixa de ser subtipo de *Aluno*.

Linguagens com herança múltipla resolvem o problema da subtipagem múltipla e não necessitam de métodos envoltórios ou da reimplementação de métodos de uma das classes, mas ainda precisam lidar com problemas de colisão de nomes e herança repetida.

A colisão de nomes ocorre quando existem nomes idênticos (homônimos) de atributos ou métodos nas classes herdadas. Nesses casos, a linguagem deve especificar uma regra para permitir aos métodos da classe herdeira identificar a classe do atributo ou método ao qual o nome está se referindo.

Existem diferentes abordagens para tratar o problema de colisão de nomes. EIFFEL força as classes derivadas a renomear os membros das classes base que conflitam. SELF define uma lista de prioridades entre as classes base. CLOS unifica em um único membro todos os membros que conflitam. C++ detecta um erro de ambigüidade em tempo de compilação se há conflito, mas permite o uso do operador de resolução de escopo para resolver o conflito explicitamente. O exemplo 7.37 ilustra o problema de colisão de nomes em C++.

```
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();
}
```

Exemplo 7. 37 - Colisão de Nomes na Herança Múltipla em C++

Caso a linha de código comentada fosse compilada, na função *main* do exemplo 7.37, na qual um objeto da classe *ProfessorAluno* é criado e tenta utilizar a função *imprime*, seria detectado um erro de ambigüidade. Para evitar esse erro, é preciso especificar qual classe herdada está sendo referenciada pelo objeto da classe herdeira. No exemplo 7.37, poder-se-ia eliminar a ambigüidade sobrescrevendo-se a operação de impressão na classe *ProfessorAluno*. Na sobrescrição, é possível utilizar o operador de resolução de escopo *::* para acessar as operações desejadas nas classes herdadas. O exemplo 7.38 mostra uma maneira como a sobrescrição poderia ser feita.

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

Exemplo 7. 38 - Sobrescrição para Resolver Conflito de Nomes em C++

Com a sobrescrição de *imprime* em *ProfessorAluno* não mais ocorre o erro de ambigüidade em *main*. Além disso, o operador de resolução de escopo pode ser usado dentro da função sobrescrita para chamar os métodos homônimos das classes herdadas. Em caso do conflito de nomes ocorrer em um atributo, é preciso usar o operador de resolução de escopo para indicar a classe do atributo referenciado.

O problema de herança repetida ocorre quando uma classe faz herança múltipla de classes descendentes de uma mesma classe. Os atributos dessa classe comum são repetidos na classe na qual é feita a herança múltipla. Por isso, esse tipo de problema é conhecido como herança repetida.

Além de poder desperdiçar memória, esse tipo de situação pode causar também conflito de nomes. A figura 7.4 ilustra uma situação na qual ocorre herança repetida.

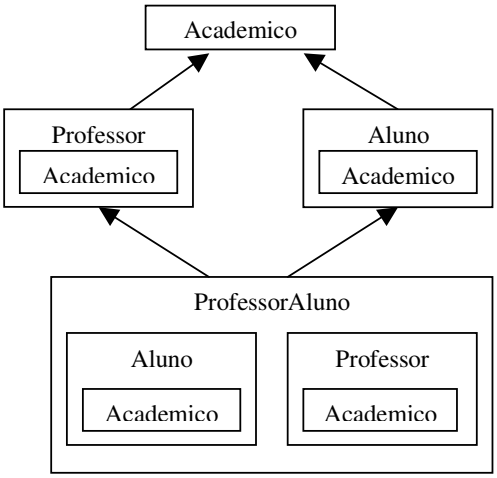

Figura 7. 4 - Herança Repetida

Os atributos da classe *Pessoa* serão duplicados em *ProfessorAluno*. Além da duplicação de memória, pode haver colisão de nomes quando os métodos de *ProfessorAluno* se referem aos atributos de *Academico*. Nesse caso, é preciso saber se o atributo referenciado é o herdado pela classe *Aluno* ou pela classe *Professor*. O conflito de nomes também pode ocorrer quando um método de *Academico* é sobrescrito em *Professor* ou *Aluno*. Novamente, será necessário saber de qual classe é o método a ser chamado.

No caso de herança repetida as alternativas também são muitas. Em CLOS os atributos compartilhados são obrigatoriamente unificados em um único atributo. EIFFEL também unifica os atributos por `default`, com exceção daqueles que são renomeados ao longo do caminho de herança. C++ fornece um mecanismo especial, através do uso da palavra *virtual*, para resolver esse problema. Se a especificação da classe herdada é precedida por *virtual* somente um objeto daquela classe comporá o objeto das classes herdeiras, independentemente de a classe ser herdada múltiplas vezes ou não. O exemplo 7.39 mostra como esse mecanismo pode ser usado.

```
class Academico {
  int i;
  int 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 {};
```

Exemplo 7. 39 - Uso de *virtual* para Evitar Herança Repetida em C++

Se a palavra *virtual* não fosse usada na especificação das classes *Aluno* e *Professor*, os atributos de idade (*i*) e matricula (*m)*de *Academico* serão repetidos na classe *ProfessorAluno*. Isso não ocorre na implementação do exemplo 7.39.

Contudo, a abordagem de C++ apresenta dois problemas. Primeiro, a especificação da palavra *virtual* não é feita na classe na qual ocorre a herança repetida. Em princípio, ao se criar as classes *Professor* e *Aluno*, não se sabe se elas serão herdadas futuramente por uma mesma subclasse e muito menos se essa subclasse precisará ou não compartilhar os atributos comuns.

O segundo problema é relacionado com esse e reflete um problema de expressividade da linguagem. No exemplo do *ProfessorAluno*, um objeto dessa classe deveria possuir um único atributo de idade (é a idade da mesma pessoa). Contudo, ele deveria possuir dois atributos matricula (uma vez que a matrícula de um professor é normalmente diferente da de um aluno).

Em C++ não é possível especificar que um atributo deve ser repetido e outro não. O mecanismo de EIFFEL permite fazer isso, mas requer mais trabalho para especificar herança múltipla pois se deve especificar individualmente os atributos que compartilhados e os repetidos.

JAVA não permite herança múltipla de classes, mas usa o conceito de interfaces para permitir subtipagem múltipla. Como dito anteriormente, interfaces são classes abstratas nas quais são definidos apenas protótipos de métodos, mas não fornecem a implementação dos métodos. Métodos da interface são considerados automaticamente públicos. Interfaces podem conter valores de tipos primitivos, mas estes são implicitamente definidos como *static* e *final*, isto é, são constantes de classe. Uma classe implementa uma interface quando ela implementa os métodos definidos

na interface. O exemplo 7.40 mostra a definição e uso de uma interface em JAVA.

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

Exemplo 7. 40 - Interface em JAVA

No exemplo 7.40 a interface *Aluno* declara os métodos *estuda* e *estagia* e é implementada pela classe *Graduando*. Embora *Aluno* seja de fato uma classe abstrata, a herança em *Graduando* é estabelecida usando a palavra *implements*. Note ainda que a declaração de métodos na interface não utiliza a palavra *public*, mas na classe *Graduando* é obrigatório utilizar essa palavra na definição dos métodos especificados em *Aluno*. Isso ocorre porque os métodos das interfaces são sempre implicitamente declarados como públicos. Se algum método da interface herdada não for implementado na classe herdeira, essa classe deve ser declarada como abstrata para não ocorrer erro de compilação.

Embora uma classe só possa herdar apenas de uma outra classe, ela pode implementar um número qualquer de interfaces. Além disso, interfaces também podem herdar de outras interfaces. A combinação de herança simples de classes com herança múltipla de interfaces é utilizada em JAVA para permitir a um objeto ser subtipo de vários tipos. Assim, um objeto de um determinado tipo pode ser usado em situações nas quais se esperam objetos de seus supertipos. O exemplo 7.41 mostra como isso pode ser feito.

```
interface Cirurgiao { void opera(); }
interface Neurologista { void consulta(); }
class Medico { public void consulta() {} }
class NeuroCirurgiao extends Medico
  implements Cirurgiao, Neurologista {
  public void opera() { }
}
public class Hospital {
  static void plantãoCirurgico (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();
    plantãoCirurgico(doutor);
    atendimentoGeral(doutor);
    neuroAtendimento(doutor);
    neuroCirurgia(doutor);
  }
}
```

Exemplo 7. 41 - Subtipagem Múltipla em JAVA

O exemplo 7.41 mostra como um objeto da classe *NeuroCirurgiao* pode ser usado nos locais onde se espera um objeto de *Cirurgiao*, *Neurologista*, *Medico* ou *NeuroCirurgiao*. Note que a classe concreta *NeuroCirurgiao* não precisa necessariamente definir o método *consulta* da interface *Neurologista*. Isso não é preciso porque a definição do método é herdada da classe *Medico*. Já o método *opera* tem de ser implementado em *NeuroCirurgiao* para que essa classe não seja considerada abstrata.

Como interfaces não podem conter valores (a menos de constantes de classe, que são únicas para a classe e não se repetem nos objetos) e não implementam métodos, os problemas de conflito de nomes e herança repetida também são evitados. Portanto, o mecanismo de interfaces oferece uma solução elegante para esses problemas e ao mesmo tempo possibilita a ocorrência de subtipagem múltipla.

Por outro lado, esse mecanismo demanda das classes concretas que implementam interfaces ficarem repletas de implementações de métodos necessários para satisfazer todas as interfaces relacionadas. Mais que isso, eventualmente, o mesmo código terá de ser repetido em diferentes classes que implementam a mesma interface. De fato, se a herança múltipla fosse irrestrita, métodos desse tipo poderiam ser implementados uma única vez em uma superclasse e herdados pelas subclasses sem necessidade de reescrevê-los.

Por fim, cabe comentar a existência de herança dinâmica em algumas LPs. Herança dinâmica é a habilidade de adicionar, excluir ou modificar superclasses dos objetos (ou classes) em tempo de execução. Assim, ela possibilita que objetos possam se modificar e evoluir com o tempo. CLOS e SMALLTALK oferecem herança dinâmica. C++ e JAVA não oferecem.

7.2.4.4 Metaclasses

Uma metaclasse é uma classe de classes, isto é, uma classe cujas instâncias são outras classes. Em LPs com metaclasses, classes também são ob-

jetos. Tal como classes normais, metaclasses são descritas através de atributos e métodos. Tipicamente, os valores dos atributos são os métodos, instâncias e as superclasses das classes que são instâncias da metaclasse. Os métodos da metaclasse 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*. Esta classe fornece serviços aos programas de aplicação, tais como, a recuperação de métodos, atributos, classes internas, a criação de novas instâncias e a invocação de métodos de um objeto. O exemplo 7.42 mostra isso sendo feito 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);
  }
}
```

Exemplo 7. 42 - Metaclasse em JAVA

No exemplo 7.42 *info* é um objeto da metaclasse *Class* designando a classe *Info*. O objeto *info* é usado para se obter os métodos da classe *Info*, para criar uma instância dessa classe e para chamar seu método *informa* (aquele sem parâmetros). Cada LP possui um modo específico de lidar com metaclasses. Os diversos modos se enquadram em um dos seguintes sistemas:

- **Nível Único:** Todos objetos são vistos como classes e todas as classes são vistas como objetos. Não há necessidade de metaclasses porque os objetos se autodescrevem. Só existe um tipo de objeto. Um exemplo de LP que adota este sistema é SELF.
- **Dois Níveis:** Todos objetos são instâncias de uma classe, mas classes não são acessíveis por programas. Existem dois tipos de objetos: objetos e classes. Um exemplo de LP que adota este sistema é C++.

- **Três Níveis:** Todos objetos são instâncias de uma classe e todas as classes são instâncias de uma única metaclasse. A metaclasse é uma classe e, portanto, uma instância de si mesma. Isto possibilita que classes sejam manipuladas diretamente pelos programas de aplicação. Existem dois tipos de objetos (objetos e classes), com uma classe atípica (a metaclasse). Um exemplo de LP que adota este sistema é JAVA.
- **Vários Níveis:** Semelhante ao sistema de três níveis, mas com níveis adicionais que permitem a criação de metaclasses especializadas para um conjunto de classes. Existem três tipos de objetos (objetos, classes e metaclasses). Um exemplo de LP que adota este sistema é SMALLTALK.

7.3 Considerações Finais

Nesse capítulo foram abordados conceitos relacionados com sistemas de tipos polimórficos, os quais aumentam a possibilidade de reutilização de código. Mostrou-se que o polimorfismo pode ser ad-hoc (coerção ou sobrecarga) ou universal (paramétrico ou inclusão). Discutiu-se com profundidade cada um desses tipos de polimorfismo, com especial destaque para o polimorfismo de inclusão, fundamental em linguagens orientadas a objetos.

Uma discussão interessante relativa ao polimorfismo de inclusão diz respeito às indicações de uso de composição ou herança para a reutilização de código. A composição ocorre quando um atributo de uma classe é um objeto de outra classe. Isso significa que o código da classe do objeto pode ser reutilizado pela classe que o tem como atributo. Herança também pode ser vista como a colocação implícita de um objeto da classe herdada dentro de uma classe herdeira. Portanto, tanto composição como herança permitem a colocação de objetos dentro de uma nova classe e a reutilização do código da classe desse objeto.

Composição é geralmente utilizada quando se quer as características de uma classe existente dentro de uma nova classe, mas não sua interface. Isto é, a inserção de um objeto ocorre de modo a utilizá-lo para implementar a funcionalidade da nova classe, mas o usuário acessa a nova classe somente através da sua própria interface, não usando a interface do objeto inserido. Para alcançar esse efeito se deve incluir os objetos de classes existentes como privados na nova classe.

A herança parte de uma classe existente e cria uma nova versão dessa classe. Em geral, isso significa que se está tomando uma classe de propósito mais geral e especializando-a para uma necessidade particular. Além

de usar as características, a classe herdeira também usa a interface da classe herdada.

Uma regra geral para tomar a decisão sobre herança ou composição é a seguinte: um relacionamento entre classes do tipo **é-um** é expresso como herança e um relacionamento do tipo **tem-um** é expresso como composição. Por exemplo. não faz sentido compor um automovel a partir de um objeto veículo pois um automovel não contém um veiculo. De fato, um automovel é um veículo. Logo, nesse caso, a regra mencionada sugere que se deve usar herança.

Outro ponto interessante diz respeito aos tipos de polimorfismo envolvidos com o polimorfismo de inclusão. Os métodos herdados da superclasse são polimórficos universais pois seu código é usado tanto para as instâncias da superclasse quanto das subclasses. Estruturas de dados cujos elementos são da superclasse também são polimórficas universais pois podem abrigar instâncias da própria superclasse e de qualquer uma de suas subclasses. Já o polimorfismo puramente decorrente da sobrescrição de métodos é um polimorfismo ad-hoc de sobrecarga pois as chamadas desses métodos invocam códigos diferentes para cada tipo de objeto. Existe ainda o polimorfismo decorrente da amarração tardia de tipos no código usuário. Esse polimorfismo também é universal pois o código usuário pode ser usado com instâncias de diferentes classes.

Um último assunto para ser discutido são as diferentes possibilidades oferecidas por C++ e JAVA para criar código polimórfico na implementação de estruturas de dados genéricas, tanto homogêneas quanto heterogêneas.

Estruturas de dados genéricas são capazes de armazenar e operar sobre elementos de tipos diferentes. Estruturas genéricas podem ser preenchidas com elementos de um mesmo tipo (nesse caso são chamadas de estruturas homogêneas) ou de tipos diferentes (nesse caso são chamadas de estruturas heterogêneas).

C++ utiliza o polimorfismo paramétrico proporcionado pelo mecanismo de *template* para a criação de estruturas de dados genéricas homogêneas e o mecanismo de polimorfismo por inclusão para a criação de estruturas de dados heterogêneas. É possível ainda combinar o mecanismo de *template* com o polimorfismo de inclusão para criar estruturas de dados genéricas heterogêneas.

JAVA utiliza o polimorfismo por inclusão para permitir a criação de estruturas de dados genéricas heterogêneas. Para isso, JAVA considera todas as classes existentes como subclasses (diretas ou indiretas) da classe *Object*.

Assim, estruturas de dados cujos elementos são do tipo *Object* podem abrigar elementos de qualquer classe em JAVA. Para ter uma estrutura de dados homogênea o programador deve garantir que os elementos inseridos são sempre de um mesmo tipo. Isso é pior do que a solução de C++, na qual o compilador garante a homogeneidade da estrutura. Além disso, a solução de JAVA obriga a realização de estreitamento sempre que um elemento deve ser acessado a partir da lista. Por outro lado, a solução de JAVA simplifica a linguagem pois não têm sido necessário incluir o polimorfismo paramétrico, o qual é imprescindível para C++.

7.4 Exercícios

1. Segundo a classificação de Cardelli e Wegner, existem quatro tipos de polimorfismo. Quais desses tipos de polimorfismo existem em C, C++ e JAVA? Mostre exemplos desses tipos de polimorfismo com trechos de código em C, C++ ou JAVA. Identifique o tipo de polimorfismo que ocorre em cada exemplo e explique porque cada um dos trechos de código é polimórfico. Indique ainda se o polimorfismo é ad-hoc ou universal e justifique.

2. O que são classes abstratas? Quando devem ser usadas e quais as suas vantagens? Quais diferenças existem na definição de classes abstratas em C++ e JAVA?

3. Enquanto em C++ somente os métodos precedidos pela palavra *virtual* utilizam o mecanismo de amarração tardia de tipos, em JAVA todos os métodos empregam este mecanismo. Justifique esta decisão dos criadores dessas linguagens.

4. Linguagens de programação orientadas a objetos podem adotar herança simples ou múltipla. C++, por exemplo, adota herança múltipla. Quais os dois problemas que podem ocorrer quando se adota herança múltipla? Explique-os usando exemplos em C++. Mostre de que forma C++ permite contornar esses problemas. Apresente um exemplo de situação na qual os mecanismos de C++ são inadequados para tratá-los.

5. C++ oferece três mecanismos distintos para permitir a realização de estreitamento. Mostre exemplos do uso desses três mecanismos e os compare em termos de confiabilidade e eficiência.

6. Amarração tardia de tipos é o processo de identificação em tempo de execução do tipo real de um objeto. Esse processo pode ser utilizado para a identificação dinâmica do método a ser executado (quando ele é

sobrescrito) e para a verificação das operações de estreitamento. Explique como é implementado o mecanismo de amarração tardia de tipos em linguagens como C++ e JAVA e como ele é usado para realizar as operações mencionadas na frase anterior.

7. Tanto C++ quanto JAVA oferecem bibliotecas de classes que disponibilizam estruturas de dados genéricas, tais como listas, árvores e tabelas `hash`. Ambas utilizam polimorfismo para a implementação dessas estruturas, embora sejam formas diferentes de polimorfismo. Explique como essas linguagens usam o polimorfismo para a implementação dessas estruturas. Discuta as vantagens e desvantagens de cada abordagem. Justifique também porque os criadores dessas linguagens adotaram essa postura diferenciada.

8. Implemente o tipo abstrato de dados lista genérica em C, C++ e JAVA. É suficiente apresentar a definição do tipo e o cabeçalho dos métodos de construção, ordenação, destruição, verificação de lista vazia, inclusão e exclusão de elemento (não é preciso codificar os métodos da lista). Atente para o fato de que a operação de ordenação na lista deve ser única, mas deve permitir que a lista seja ordenada por critérios distintos. Justifique a sua implementação, enfocando o modo como se obtém a generalidade da lista e o funcionamento da operação de ordenação.

9. Em alguns problemas pode ser conveniente permitir a um mesmo objeto participar de duas ou mais listas cujos campos de informação são de tipos diferentes. Por exemplo, em um problema no qual é necessário armazenar em listas os diversos tipos de dependências de um apartamento haveria uma lista para quartos e outra para salas. Contudo, é possível haver uma mesma dependência usada como quarto e como sala. Nesse caso, essa dependência participaria tanto da lista de quartos quanto da lista de salas.

   a) Como você resolveria esse problema em uma linguagem que não possui mecanismos para a realização de subtipagem múltipla, ou seja, que permita a um mesmo objeto fazer parte de listas de dados cujos tipos de informação sejam de tipos distintos. Existe alguma desvantagem na sua solução.

   b) Sabendo que C também não permite a realização de subtipagem múltipla, responda se é possível resolver esse mesmo problema de outra maneira? Se a resposta for positiva, explique como seria essa solução e faça uma análise de suas vantagens e desvantagens.

c) Mostre através da implementação desse exemplo como C++ e JAVA permitem a realização de subtipagem múltipla.

Compare ainda os mecanismos oferecidos por C++ e JAVA para realização de subtipagem múltipla apresentando vantagens e desvantagens de cada um.

10. Considere as seguintes definições de funções e classes em C++:

```
template <class T >
T xpto (T x, T y) {
  return y;
}
template <class T, class U>
U  ypto (T x, U y) {
  return y;
}
template <class T, class U>
T zpto (T x, U y) {
  return ((T) y);
}
class tdata {
  int d, m, a;
};
class thorario {
  int h, m, s;
};
class tdimensao {
  int h, l, w;
};
```

Indique quais das linhas de código de *main* abaixo são legais e explique as que não são.

```
main () {
  tdata a;
  thorario b;
  tdimensao c;
  a = xpto (a, a);
  b = xpto (a, a);
  c = xpto (a, b);
  a = ypto (a, a);
  b = ypto (a, a);
  b = ypto (a, b);
  c = ypto (a, b);
  a = zpto (a, a);
  b = zpto (a, a);
  a = zpto (a, b);
```

```
    c = zpto (a, b);
}
```

Explique ainda como o compilador C++ implementa o mecanismo de polimorfismo paramétrico. Discuta essa solução em termos de reusabilidade de código.

11. Cada um dos programas seguintes, escritos em C++, utiliza um tipo de polimorfismo visto nesse capítulo. Defina o que é polimorfismo. Descreva as características desses tipos de polimorfismo, indicando o tipo empregado por cada programa. Execute os programas passo a passo, mostrando o resultado apresentado, indicando onde ocorre polimorfismo e explicando sua execução.

```
// programa 1
#include <iostream>

class base {
   public:
      virtual void mostra1() {
         cout << "base 1\n";
      }
      void mostra2 () {
         cout << "base 2 \n";
      }
};
class derivada1: public base {
   public:
      void mostra1() {
         cout << "derivada 1\n";
      }
};
class derivada2: public base {
   public:
      void mostra2 () {
         cout << "derivada 2 \n";
      }
};
void prt(base *q) {
   q->mostra1();
   q->mostra2();
}
void main( ) {
   base b;
   base *p;
   derivada1 dv1;
   derivada2 dv2;

   p = &b;
   prt(p);

   dv1.mostra1();
   p = &dv1;
   prt(p);

   dv2.mostra2();
   p = &dv2;
   prt(p);
}
```

```
// programa 2
#include <iostream>

class teste {
   int d;
 public:
   teste () {
      d = 0;
      cout << "default \n";
   }
   teste (int p, int q = 0) {
      d = p + q;
      cout << "soma \n";
   }
   teste (teste & p) {
      d = p.d;
      cout << "copia \n";
   }
   teste & operator = (teste & p)
   {
      cout << "atribuicao 1\n";
      d = p.d; return *this;
   }
   teste & operator = (int i)  {
      cout << "atribuicao 2\n";
      d = i; return *this;
   }
   void mostra ( ) {
      cout <<  d << " \n";
   }
};
void main ( ) {
   teste e1 (2, 6);  e1.mostra ( );
   teste e2;           e2.mostra ( );
   teste e3 (73);    e3.mostra ( );
   teste e4;
   e4 = e1;  e4.mostra();
   teste e5 (e2);    e5.mostra ( );
   e5 = 55;           e5.mostra ( );
   teste e6 = e3;   e6.mostra ( );
   teste e7 (21,3);  e7.mostra ( );
   e1 = e3 = e5 = e7;
   e1.mostra();     e5.mostra();
}
```

```
// programa 3
#include <iostream>

template <class T>
class pilha {
   T* v;
   T* p;
public:
   pilha (int i) {
      cout << "cria "<< i << "\n";
      v = p = new T[i];
   }
   ~pilha () {
      delete[ ] v;
       cout << "tchau \n";
   }
   void empilha (T a) {
      cout << "emp "<< a << "\n";
      *p++ = a;
   }
   T desempilha () {
      return *--p;
   }
   int vazia() {
    if (v == p) return 1;
    else  return 0;
   }
};
void main ( ) {
   pilha<int> p(40);
   pilha<char> q(30);
   p.empilha(11);
   q.empilha('x');
   p.empilha(22);
   q.empilha('y');
   p.empilha(33);
   do {
      cout << p.desempilha() <<
      "\n";
   } while (!p.vazia());
   do {
      cout << q.desempilha() <<
      "\n";
   } while (!q.vazia());
}
```

12. Qual a diferença entre as posturas adotadas por JAVA e C++ em relação ao polimorfismo de sobrecarga? Qual dessas posturas você acha melhor? Apresente argumentos justificando sua posição.

13. Uma loja especializada em produtos de arte vende livros, discos e fitas de vídeo. Ela necessita montar um catálogo com as informações sobre cada produto. Todo produto possui um número único de registro, um preço e uma quantidade em estoque. Além disso, deve-se saber o número de páginas dos livros, o número de músicas dos discos e a duração da fita de vídeo. Outro aspecto importante a ser considerado é a existência de um tipo de produto de venda combinada (uma fita de vídeo combinada com um disco). Utilize C, C++ e JAVA para:

    a) Especificar tipos abstratos de dados para cada um dos produtos da loja (basta apresentar a definição do tipo e os cabeçalhos de suas operações de criação, leitura, obtenção de dados, escrita e destruição).

    b) Supondo que os produtos da loja estão armazenados em uma lista genérica, fazer uma função/método para receber a lista dos produtos e uma quantidade mínima de produtos a serem mantidos em estoque, e listar quais produtos necessitam de reposição.

    c) Fazer uma função/método para receber a lista dos produtos, uma quantidade de músicas e uma duração mínima de filme, e listar quais discos possuem menos músicas que a quantidade especificada e quais filmes possuem maior duração que a especificada.

Compare os mecanismos oferecidos por cada uma das linguagens (C, C++ e JAVA) para lidar com o produto combinado. Enfoque na sua comparação os aspectos de facilidade de reutilização de código, facilidade para a realização do ítem c desta questão e o potencial para indução de erros de programação.

14. O Ministério da Defesa te contratou para desenvolver um protótipo de um sistema de informação em C++ que cadastre os militares existentes nas Forças Armadas Brasileiras e gere duas listagens.

a. Crie uma classe abstrata *Militar* com um atributo inteiro representando sua matrícula e outro atributo representando sua patente. Garanta que todas subclasses concretas de *Militar* implementem obrigatoriamente os métodos de leitura de dados de um militar, impressão de dados de um militar e verificação se o militar está habilitado para progredir na carreira.

    Utilize a classe seguinte para representar a patente do militar.

```
class Patente {
  private:
    string titulo;
    int tempo;  // na patente
  public:
    le() {
      cin >> titulo;
      cin >> tempo;
    }
    string retornaPatente() {
      return titulo;
    }
    int retornaTempo() {
      return tempo;
    }
    void incrementa (int t) {
       tempo+=t;
    }
    void imprime() {
      cout << titulo << " – "
      << tempo;
    }
}
```

b. Implemente uma subclasse concreta de *Militar*, denominada *MilitarAeronautica*, que será usada para representar os militares dessa divisão das forças armadas. Essa classe possui como atributo adicional o número de horas de vôo efetuadas pelo militar. Note que um militar da aeronáutica está em condições de progredir se tem mais de 100 horas de vôo e está a mais de 2 anos naquela patente.

c. Considerando a existência de duas outras subclasses de *Militar* semelhantes a *MilitarAeronáutica*, denominadas *MilitarExercito* e *MilitarMarinha*, utilize uma classe *ListaMilitares* (consistindo de uma lista de ponteiros para militares) para implementar um programa que:
   - Solicite ao usuário o número total de militares das Forças Armadas
   - Solicite ao usuário a corporação e os dados de cada militar das Forças Armadas
   - apresente os dados de todos os militares em condições de progredir na carreira.
   - apresente os dados de todos os militares da Aeronáutica.

Observação: Você não deve implementar a classe *ListaMilitares*. Considere que, além de possuir o método construtor default e o destrutor, ela também possui os seguintes métodos:

```
void incluir(Militar* m);        // inclui um militar ao final da lista
Militar* retornar(int i);        // retorna o militar na i-ésima posição
```